Scalable Load-Distance Balancing

Edward Bortnikov   Israel Cidon   Idit Keidar
ebortnik@techunix.technion.ac.il   {cidon,idish}@ee.technion.ac.il

Department of Electrical Engineering
The Technion
Haifa 32000

Abstract. We introduce the problem of *load-distance balancing* 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 service delay among all users.
We address the challenge of finding a near-optimal 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 hosts to Internet gateways in an urban wireless mesh network (WMN).

Keywords— Local Computation, Distributed Algorithms, Load-Distance Balancing, Wireless Networks

1 Introduction

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) [12] and massively multiplayer online gaming (MMOG) grids [8]. Another example can be found in wireless mesh networks (WMNs) [2]. 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. To this end, gateway functionality is anticipated to expand, and to deploy application server logic [2].

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 [8], 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 [10].

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 traffic of a WMN user is typically routed via a single gateway [2].

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 paper, we present a novel approach to service assignment, which is based on both metrics. We call the new problem, which seeks to minimize the maximum service delay among all users, *load-distance balancing* (Section 3).

Resource management problems in which the assignment of every user to the closest server leads to unsatisfactory 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 that, however, 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. In this paper, we address this challenge, drawing inspiration from workload-adaptive distributed algorithms [6, 14].

In Section 4, 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 the full version of this paper [7], 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 Appendix A, we discuss how they can be extended to cope with dynamic workloads.

We note that even as a centralized optimization problem, load-distance balancing is NP-hard, as we show in the full version of this paper [7]. Therefore, `Tree` and `Ripple` employ a centralized polynomial 2-approximation algorithm, `BFlow`, within each cluster. For space limitations, the presentation of `BFlow` is also deferred to the full paper.

Finally, we empirically evaluate our algorithms using a case study in an urban WMN environment (Section 5). 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.

2 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., [4]). However, in large-scale networks, simple load balancing is insufficient because servers are not co-located. While some prior work [8, 12] 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. Moreover, in contrast with distributed algorithms for traditional load balancing (e.g., [11]), 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., [5, 9]), 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., [12]. 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.

Local solutions of network optimization problems have been addressed starting from [16] ,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 [15], Minimum Dominating Set and Maximum Independent Set [13]. While some papers explore the tradeoff between the allowed running time and the approximation ratio (e.g., [15]), 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 [14], and local distributed aggregation [6].

3 Definitions and System Model

Consider a set of k servers S and a set of n user sessions U, such that $k \ll n$. The users and the servers reside in some metric space, in which the *network delay* function, $D : (U \times S) \to \mathbb{R}^+$, captures the network distance between a user and a server.

Consider an assignment $\lambda : U \to S$ that maps every user to a single server. Each server s has a monotonic non-decreasing *congestion delay* function, $\delta_s : \mathbb{N} \to \mathbb{R}^+$, 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 λ 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 [12], 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 [10], 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 [2].

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

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

The `LDB` (load-distance balancing) assignment problem is to find an assignment λ^* such that $\Delta^M(\lambda^*(U))$ is minimized. An assignment that yields the minimum cost is called *optimal*. The `LDB` problem is NP-hard. Our optimization goal is therefore to find a constant approximation algorithm for this problem. We denote the problem of computing an α-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 network delay function D and the set of server congestion functions $\{\delta_s\}$ are known to all servers. 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., [3]).

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 α.

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).

4 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` [7]), which computes an $r_{\mathtt{ALG}}$-approximation for a given instance of the LDB problem. They are also parametrized by the *required* approximation ratio α, which is greater or equal to $r_{\mathtt{ALG}}$. 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 [17]), 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 α: the larger α is, the smaller the constructed clusters are.

We call a value ε, such that $\alpha = (1+\varepsilon)r_{\mathtt{ALG}}$, the algorithm's *slack factor*. A cluster is called *ε-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. ε-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 ε-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).

4.1 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 ε-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.

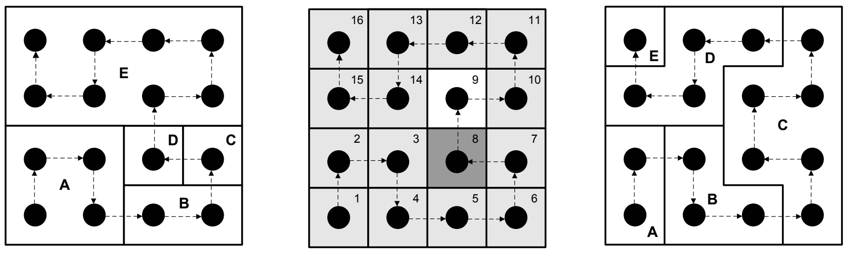
A single round consists of three synchronous phases: the first phase initiates the process with a *"merge"* message (from a child to its parent), the second disseminates the *"query"* message (from a leader to all its descendants), and the third collects the *"reply"* messages (from all descendants back 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.

Fig. 1(a) depicts a sample clustering of `Tree` where 16 servers reside on a 4×4 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 ε-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 the full paper [7]).

Theorem 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 ε-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 α-approximation of the optimal cost.*



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

Fig. 1. Example workloads for the algorithms and clusters formed by them in a 4×4 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`.

`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. Fig. 1(b) shows an example workload on the network in Fig. 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 ε, 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.

4.2 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 ε-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., Fig. 1(c)). The elimination of the hierarchy also introduces some challenges and race conditions between requests from different neighbors.

We first make some formal definitions and present `Ripple` at a high level. Following this, we provide the algorithm's technical details. Finally, we claim `Ripple`'s properties; their formal proofs appear in the full version of this paper [7].

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 $C_i.imp$,

Message	Semantics	Size
$\langle$*"probe",id,cost,imp*$\rangle$	Assignment summary (cost and ε-improvability)	small, fixed
$\langle$*"propose",id*$\rangle$	Proposal to join	small, fixed
$\langle$*"accept",id,*λ*,nid*$\rangle$	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
`LdrId`	the cluster leader's id	`Id`
Λ	the internal assignment	`NearestServer`
`Cost`	the cluster's cost	Δ^M(`NearestServer`)
`NbrId`[2]	the L/R neighbor cluster leader's id	$\{\texttt{Id}-1, \texttt{Id}+1\}$
`ProbeS`[2]	*"probe"* to L/R neighbor sent?	$\{\texttt{false}, \texttt{false}\}$
`ProbeR`[2]	*"probe"* from the L/R neighbor received?	$\{\texttt{false}, \texttt{false}\}$
`PropR`[2]	*"propose"* from L/R neighbor received?	$\{\texttt{false}, \texttt{false}\}$
`ProbeFwd`[2]	need to forward *"probe"* to L/R?	$\{\texttt{false}, \texttt{false}\}$
`Probe`[2]	need to send *"probe"* to L/R in the next round?	$\{\texttt{true}, \texttt{true}\}$
`Prop`[2]	need to send *"propose"* to L/R?	$\{\texttt{false}, \texttt{false}\}$
`Acc`[2]	need to send *"accept"* to L/R?	$\{\texttt{false}, \texttt{false}\}$

Fig. 2. **`Ripple`'s messages, constants, and state variables.**

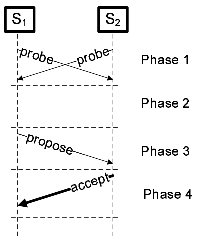
respectively. Two clusters C_i and C_j ($1 \leq i < j \leq k$) are called *neighbors* if there exists an l such that server s_l belongs to cluster C_i and server s_{l+1} belongs to cluster C_j. Cluster C_i is said to *dominate* cluster C_j if:

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

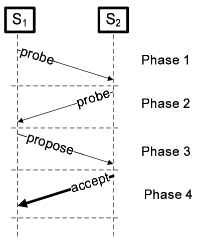
`Ripple` proceeds in rounds. During a round, a cluster that dominates some (left or right) neighbor tries to reduce its cost by inviting this neighbor to merge with it. A cluster that dominates two neighbors can merge with both in the same round. A dominated cluster can only merge with a single neighbor 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, communication stops, and the assignment remains stable.

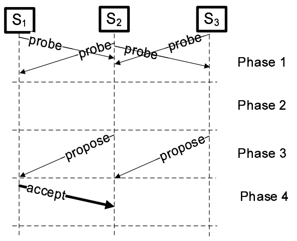
Detailed Description Fig. 2 provides a summary of the protocol's messages, constants, and state variables. See Fig. 4 for the pseudo-code. We assume the existence of local functions `ALG` : $(U, S) \rightarrow \lambda$, $\Delta^M : \lambda \rightarrow \mathbb{R}^+$, and `improvable` : $(\lambda, \varepsilon) \rightarrow \{\texttt{true}, \texttt{false}\}$, which compute the assignment, its cost, and the improvability flag.



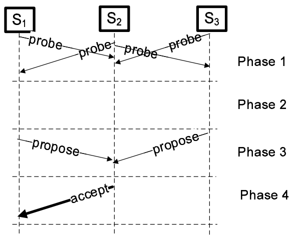
(a) Simultaneous probe: s_1 and s_2 send messages in Phase 1.



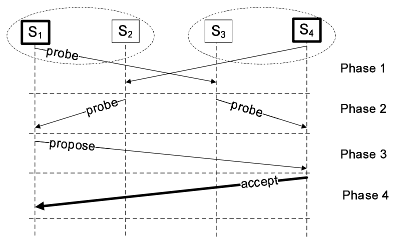
(b) Late probe: s_2 sends message in Phase 2.



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



(d) $\Leftarrow\Rightarrow$ 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.

Fig. 3. `Ripple`'s scenarios. Nodes in solid frames are cluster leaders. Dashed ovals encircle servers in the same cluster.

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` (ll. 4–6). Upon receiving a probe from a neighbor, if the cluster dominates this neighbor, the cluster's leader schedules a proposal to merge (line 53), and also decides to send a probe to the neighbor in this direction in the next round (line 55). If the neighbor dominates the cluster, the cluster's leader decides to accept the neighbor's proposal to merge, should it later arrive (line 54). Fig. 3(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 (ll. 14–16). Fig. 3(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 (ll. 19–20). Fig. 3(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 (ll. 58–68). Figures 3(c) and 3(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 (ll. 28–29).

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 37). 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 Fig. 3(e). At the end of the round, a consuming cluster's leader re-computes the assignment within its cluster (ll. 70–72). Note that a merge 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 (ll. 73–74).

Ripple's Properties We now discuss `Ripple`'s properties. Their proofs appear in the full version of this paper [7].

Theorem 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 α-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 [7]). However, the worst-case scenario is not representative. Our case study (Section 5) 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 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-ε-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:    forall $d \in \{L, R\}$ do
3:      initState($d$)
4:      if (LdrId $=$ Id $\wedge$ Probe[$d$]) then
5:        $i \leftarrow$ improvable($\Lambda, \varepsilon$)
6:        send ⟨"probe", Id, Cost, $i$⟩
              to NbrId[$d$]
7:        ProbeS[$d$] $\leftarrow$ true
8:        Probe[$d$] $\leftarrow$ false
9:    forall recv ⟨"probe", $id$, $cost$, $imp$⟩ do
10:     handleProbe($id$, $cost$, $imp$)

11: Phase 2 {Probe completion} :
12:   if (LdrId $=$ Id) then
13:     forall $d \in \{L, R\}$ do
14:       if ($\neg$ProbeS[$d$] $\wedge$ ProbeR[$d$]) then
15:         $i \leftarrow$ improvable($\Lambda, \varepsilon$)
16:         send ⟨"probe", Id, Cost, $i$⟩
                to NbrId[$d$]
17:   else
18:     forall $d \in \{L, R\}$ do
19:       if (ProbeFwd[$d$]) then
20:         send the latest "probe" to LdrId
21:   forall recv ⟨"probe", $id$, $cost$, $imp$⟩ do
22:     handleProbe($id$, $cost$, $imp$)

23: Phase 3 {Conflict resolution & proposal} :
24:   if (LdrId $=$ Id) then
25:     resolveConflicts()
26:   {Send proposals to merge}
27:   forall $d \in \{L, R\}$ do
28:     if (Prop[$d$]) then
29:       send ⟨"propose", Id⟩ to NbrId[$d$]
30:   forall recv ⟨"propose", $id$⟩ do
31:     PropR[dir($id$)] $\leftarrow$ true

32: Phase 4 {Acceptance or rejection} :
33:   forall $d \in \{L, R\}$ do
34:     if (PropR($d$) $\wedge$ Acc[$d$]) then
35:       {I do not object joining}
36:       LdrId $\leftarrow$ NbrId[$d$]
37:       send ⟨"accept", Id, $\Lambda$, NbrId[$\overline{d}$]⟩
              to LdrId
38:   forall recv ⟨"accept", $id$, $\lambda$, $nid$⟩ do
39:     $\Lambda \leftarrow \Lambda \cup \lambda$; Cost $\leftarrow \Delta^M(\Lambda)$
40:     NbrId[dir($id$)] $\leftarrow nid$
41:   if (LdrId $=$ Id) then
42:     computeAssignment()

43: procedure initState($d$)
44:   ProbeS[$d$] $\leftarrow$ ProbeR[$d$] $\leftarrow$ false
45:   Prop[$d$] $\leftarrow$ Acc[$d$] $\leftarrow$ false
46:   ProbeFwd[$d$] $\leftarrow$ false

47: procedure handleProbe($id$, $cost$, $imp$)
48:   $d \leftarrow$ dir($id$)
49:   ProbeR[$d$] $\leftarrow$ true
50:   NbrId[$d$] $\leftarrow id$
51:   $i \leftarrow$ improvable($\Lambda, \varepsilon$)
52:   if (LdrId $=$ Id) then
53:     Prop[$d$] $\leftarrow$
          dom(Id, Cost, $i$, $id$, $cost$, $imp$)
54:     Acc[$d$] $\leftarrow$
          dom($id$, $cost$, $imp$, Id, Cost, $i$)
55:     Probe[$d$] $\leftarrow$ Prop[$d$]
56:   else
57:     ProbeFwd[$d$] $\leftarrow$ true

58: procedure resolveConflicts()
59:   { Resolve $\Leftarrow\Leftarrow$ or $\Rightarrow\Rightarrow$ conflicts}
60:   forall $d \in \{L, R\}$ do
61:     if (Prop[$d$] $\wedge$ Acc[$\overline{d}$]) then
62:       if (randomBit() $= 0$) then
63:         Prop[$d$] $\leftarrow$ false
64:       else
65:         Acc[$\overline{d}$] $\leftarrow$ false
66:   {Resolve $\Rightarrow\Leftarrow$ conflict}
67:   if (Acc[$L$] $\wedge$ Acc[$R$]) conflicts then
68:     Acc[randomBit()] $\leftarrow$ false

69: procedure computeAssignment()
70:   $\Lambda' \leftarrow$ ALG(Users($\Lambda$), Servers($\Lambda$))
71:   if ($\Delta^M(\Lambda') < \Delta^M(\Lambda)$) then
72:     $\Lambda \leftarrow \Lambda'$; Cost $\leftarrow \Delta^M(\Lambda')$
73:     forall $d \in \{L, R\}$ do
74:       Probe[$d$] $\leftarrow$ true

75: function dom($id_1$, $cost_1$, $imp_1$,
                  $id_2$, $cost_2$, $imp_2$)
76:   return ($imp_1$ $\wedge$
              ($imp_1$, $cost_1$, $id_1$)
               $>$
              ($imp_2$, $cost_2$, $id_2$))

77: function dir($id$)
78:   return ($id <$ Id) ? $L$ : $R$
```

Fig. 4. Ripple**'s pseudo-code: single round.**

5 Numerical Evaluation

In this section, we employ `Tree` and `Ripple` for gateway assignment in an urban WMN, using the `BFlow` centralized algorithm [7] for local assignment. We compare our algorithms with `NearestServer`.

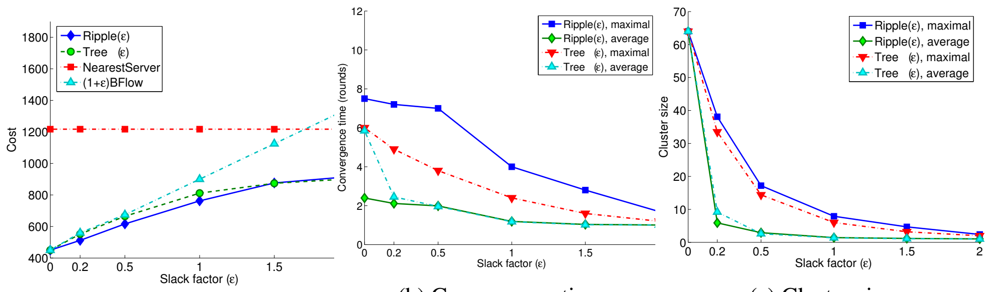
The WMN provides access to a real-time service (e.g., a network game). The mesh gateways, which are also application servers, form a rectangular grid. This topology induces a partitioning of the space into cells. The wireless backbone within each cell is an 16×16 grid of mesh routers, which route the traffic either to the gateway, or to the neighboring cells. The routers apply flow aggregation [10], 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, e.g., 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 ε ranging from 0 to 2. The results show that both `Tree` and `Ripple` significantly improve the cost achieved by `NearestServer` (Fig. 5(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.

Fig. 5(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 ε is increased.

Fig. 5(c) depicts how the algorithms' average and maximal cluster sizes depend on ε. The average cluster size does not exceed 2.5 servers for $\varepsilon \geq 0.5$. The maximal size drops fast as ε increases. Note that for the same value of ε, `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



(a) Cost

(b) Convergence time (maximal/average)

(c) Cluster size (maximal/average)

Fig. 5. Sensitivity of `Tree`(ε)'s and `Ripple`(ε)'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).

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.

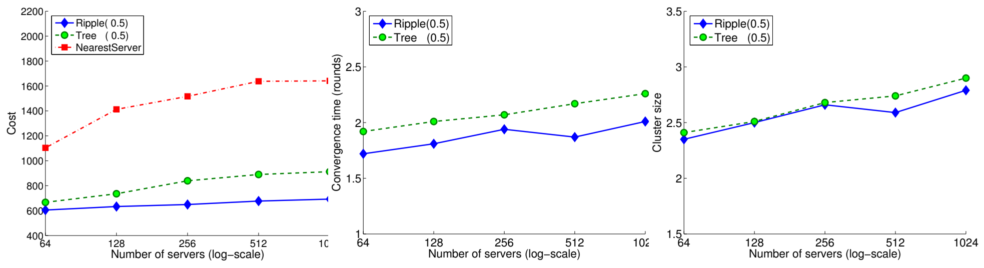
Sensitivity to network size: Next, we explore `Tree`'s and `Ripple`'s scalability with the network size, for $\varepsilon = 0.5$ and the same workload as in the previous section. We gradually increase the number of gateways from 64 to 1024. Fig. 6 depicts the results in logarithmic scale. We see that thanks to `Ripple`'s flexibility, its cost scales better than `Tree`'s, remaining almost constant with the network growth (Fig. 6(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.

Fig. 6(b) and Fig. 6(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: In the full paper [7], we also study the algorithms' sensitivity to varying workload parameters, like congestion skew and the size of congested areas. We demonstrate that whereas our algorithms perform well on all workloads, their advantage for peaky distributions is most clear. Here too, `Ripple` achieves a lower cost than `Tree`. The algorithms' maximal convergence times and cluster sizes are high only when the workload is skewed.

6 Conclusions

We defined 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 α-approximation of the optimal solution.



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

Fig. 6. 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).**

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.

References

1. Cisco Wireless Control System. `http://www.cisco.com/univercd/cc/td/doc/product/wireless/wcs`.
2. I.F. Akylidiz, X. Wang, and W. Wang. Wireless Mesh Networks: a Survey. *Computer Networks Journal (Elsevier)*, March 2005.
3. B. Awerbuch. On the Complexity of Network Synchronization. *JACM*, 32:804–823, 1985.
4. A. Barak, S. Guday, and R. Wheeler. The MOSIX Distributed Operating System, Load Balancing for UNIX. *LNCS, Springer Verlag*, 672, 1993.
5. Y. Bejerano and S.-J. Han. Cell Breathing Techniques for Balancing the Access Point Load in Wireless LANs. *IEEE INFOCOM*, 2006.
6. Y. Birk, I. Keidar, L. Liss, A. Schuster, and R. Wolff. Veracity Radius – Capturing the Locality of Distributed Computations. *ACM PODC*, 2006.
7. E. Bortnikov, I. Cidon, and I. Keidar. Scalable Load-Distance Balancing in Large Networks. Technical Report 587, CCIT, EE Department Pub No. 1539, Technion IIT, May 2006. `http://comnet.technion.ac.il/magma/ftp/LDBalance_tr.pdf`.
8. J. Chen, B. Knutsson, B. Wu, H. Lu, M. Delap, and C. Amza. Locality Aware Dynamic Load Management form Massively Multiplayer Games. *PPoPP*, 2005.
9. L. Du, J.Bigham, and L. Cuthbert. A Bubble Oscillation Algorithm for Distributed Geographic Load Balancing in Mobile Networks. *IEEE INFOCOM*, 2004.
10. 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(11), 2006.

11. 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.
12. K. M. Hanna, N. N. Nandini, and B. N. Levine. Evaluation of a Novel Two-Step Server Selection Metric. *IEEE ICNP*, 2001.
13. F. Kuhn, T. Moscibroda, T. Nieberg, and R. Wattenhoffer. Local Approximation Schemes for Ad Hoc and Sensor Networks. *ACM DIALM-POMC*, 2005.
14. S. Kutten and D. Peleg. Fault-Local Distributed Mending. *J. Algorithms*, 1999.
15. T. Moscibroda and R. Wattenhoffer. Facility Location: Distributed Approximation. *ACM PODC*, 2005.
16. M. Naor and L. Stockmeyer. What can be Computed Locally? *ACM STOC*, 1993.
17. R. Niedermeyer, K. Reinhardt, and P. Sanders. Towards Optimal Locality in Mesh Indexings. *LNCS Springer-Verlag*, 1279:364–375, 1997.

A 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 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 4.2.