CCIT Report #587 May 2006

Scalable Load-Distance Balancing in Large Networks

Edward Bortnikov Israel Cidon Idit Keidar

Department of Electrical Engineering

The Technion, Haifa 32000

Israel

ebortnik@techunix.technion.ac.il {cidon,idish}@ee.technion.ac.il

June 20, 2006

Abstract

We focus on a setting where users of a real-time networked application need to be assigned to servers, e.g., assignment of hosts to Internet gateways in a wireless mesh network (WMN). The service delay experienced by a user is a sum of the network-incurred delay, which depends on its network distance from the server, and a server-incurred delay, stemming from the load on the server. We introduce the problem of load-distance balancing, which seeks to minimize the maximum service delay among all users. We address the challenge of finding a near-optimal assignment in a distributed manner, without global communication, in a large network. We present a scalable algorithm for doing so, and evaluate our solution with a case study of its application in an urban WMN.

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, ranging from wireless mesh networks (WMNs) [4] to content delivery networks (CDNs) and massively multiplayer online gaming (MMOG) grids [10]. In such settings, every application session is typically mapped to a single server. For example, WMNs provide Internet

access to residential areas with limited wired infrastructure. A mesh network is a collection of ubiquitously deployed wireless routers. A few of them, called gateways, are wired to the Internet. The mesh access protocol typically routes the traffic from a wireless host to a single gateway.

Employing distributed servers instead of centralized server farms enables location-dependent QoS optimizations, which enhance the users' real-time experience. Service responsiveness is one of the most important QoS parameters. For example, in the first-person shooter (FPS) online game [10], the system must provide an end-to-end delay guarantee of below 100ms. This guarantee is nontrivial to implement in mesh networks, due to multiple wireless hops and a limited number of gateways.

The service delay incurred to a session typically consists of two parts: 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 this twofold nature of the overall delay, simple heuristics that either greedily map every session to the closest server, or spread the load evenly between the servers 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 term this problem, which seeks to minimize the maximum delay among all users, *load-distance balancing* (Section 3).

Resource management problems are often solved centrally because purely local solutions lead to poor results. For example, Cisco Airespace wireless LAN controllers [2] perform global optimization in assigning wireless hosts to access points (APs), after collecting user 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 wide-area networks like an urban WMN. For large-scale real-time network management, a protocol that restricts itself to local communication is required.

Naturally, the degree of locality exhibited by a distributed resource management algorithm must be context-sensitive. In a non-congested area, every user can be assigned to the nearest server, without any inter-server communication. On the other hand, 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. The main challenge is therefore in providing an *adaptive* solution that adjusts itself to the congestion within the network and performs communication to a distance proportional to that required for handling the load. In this paper, we address this challenge.

Even as an optimization problem, load-distance balancing is NP-hard. We present a two-approximation centralized algorithm, termed `BFlow` (Section 4). Our search for a scalable distributed solution goes through applying the centralized solution within a bounded network area. In Section 5, we present an adaptive distributed algorithm for load-distance balancing, termed `Ripple`. The algorithm employs `BFlow` as a building block, and adjusts its communication requirements to congestion distribution. `Ripple` produces a constant approximation of the optimal cost.

We conduct an extensive case study of our algorithm in an urban WMN environment (Section 6). Our simulation results show that `Ripple` achieves a consistently better cost than a naïve local heuristic, while communicating to small distances and converging in short time on average. The solution's cost, as well as the algorithm's convergence time and communication distance, are both scalable and congestion-sensitive, that is, they depend on the distribution of workload rather than the network size.

2 Related Work

Load-distance balancing is an extension of a well-known load balancing problem, which seeks to evenly spread the workload among multiple servers. Load balancing has been extensively studied in the context of tightly coupled systems like multiprocessors, compute clusters etc (e.g., [6]). In large-scale networks, however, simple load balancing is insufficient because servers are not co-located. Moreover, centralized load balancing solution are inappropriate in large geographically distributed systems, for scalability reasons. While some prior work (e.g., [10]) indicated the importance of joint handling of distance and load in these environments, we are not aware of any study that provides a cost model which combines these two metrics and can be rigorously analyzed.

Recently, a number of papers addressed the issue of geographic load-balancing for throughput maximization in cellular networks [12] and wireless LANs [7], and proposed natural local solutions through dynamic adjustment of cell size. While the motivation of these works is similar to ours,

their model is constrained by a rigid requirement that a user can only be assigned to some 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 multihop networks like WMNs. In addition, dealing with locality in this setting is more challenging because the potential assignment space is very large.

Prior WMN research addressed resource management problems that are specific to wireless, e.g., exploiting multiple radio interfaces to increase throughput [5]. However, the wireless part of the mesh is not necessarily a bottleneck if gateways are scarce and resource-constrained.

Local solutions of network optimization problems have been addressed by the theoretical community, starting from [15], in which the question "what can be computed locally?" was first asked. Our work is inspired in part by Kutten and Peleg's algorithm for self-stabilizing consensus [13], in which only a close neighborhood of the compromised nodes participates in the self-stabilization process. While some papers (e.g., [14]) explore the tradeoff between the allowed running time and the approximation ratio, our paper takes a different approach, also adopted by [8] – the algorithm achieves a given approximation ratio, while adapting its running time to congestion distribution.

In our previous work [9], we studied the problem of online assignment of mobile users to service points, which balances between the desire of always being connected to the closest server and the cost of migration. Unlike the current paper, that work completely ignored the issue of load.

3 Problem Definition and System Model

3.1 The Load-Distance Min-Max Delay Assignment Problem

Consider a set of servers $S = \{S_1, \ldots, S_k\}$ and a set of user sessions $U = \{u_1, \ldots, u_n\}$, so 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.

Consider an assignment $\lambda : U \rightarrow S$ that maps every user to a single server. We assume that each session u assigned to server s incurs a unit of *load* on s. We denote the load on s as $\mathcal{L}(s) \triangleq |\{u : \lambda_u = s\}|$. A monotonous non-decreasing *congestion delay* function, $\delta_s : \mathbb{N} \rightarrow \mathbb{R}^+$,

captures the delay incurred by server s to every processed session. 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:

$$\Delta(u, \lambda) \triangleq D(u, \lambda(u)) + \delta_{\lambda(u)}(\mathcal{L}(\lambda(u))).$$

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. The instance of `LDB` that seeks to compute an α-approximation for a specific value of α is termed $\alpha-$`LDB`.

3.2 Distributed System Model

We solve the $\alpha-$`LDB` problem in a distributed setting. Users and servers reside at fixed locations on a plane. The network delay grows with the Euclidean (L_2) distance between the client and the server. Each server's location and congestion delay function are known to all servers. At startup time, each user reports its location information to the closest server.

Every pair of servers can communicate directly, using a reliable channel. The algorithm's *locality* is measured by the number of servers that each server communicates with.

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.

Throughout the protocol, every server knows which users are assigned to it. At startup, every user is assigned to the closest server (this is termed a `NearestServer` assignment). Servers can then communicate and change this initial assignment. Eventually, the following conditions must hold:

1. The assignment stops changing;

2. all inter-server communication stops; and

3. the assignment's cost approximates the optimal solution with a constant factor α.

We define the *local convergence time* of a server as the number of phases that this server is engaged in inter-server communication. The *global convergence time* is defined as the maximal convergence time among all servers, i.e., the number of phases until all communication ceases.

4 Centralized Min-Max Delay Assignment

We first address the `LDB` assignment problem in a centralized setting, in which complete information about users and servers is available. The `LDB` problem is NP-hard. Its hardness can be proved through a reduction from the *exact set cover* problem [3]. The proof appears in Appendix A. In this section, we present the `BFlow` algorithm, which computes a 2-approximate solution.

`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 Δ^*, and hence, its cost is bounded by $2\Delta^*$. `BFlow` performs a binary search on the value of Δ^*. 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 client. 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 polynomial max-flow min-cut algorithm (e.g., [11]) in a bipartite user-server graph with auxiliary source and sink vertices. Figure 1 depicts an example of such a graph.

Theorem 1 `BFlow` *computes a 2-approximation of an optimal assignment for* `LDB`*.*

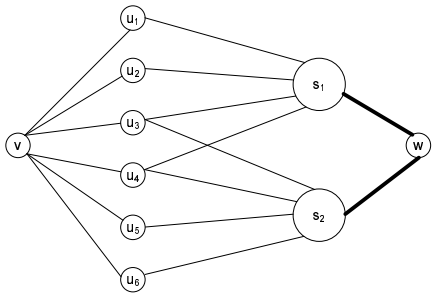


Figure 1: **The bipartite graph for a single phase of `BFlow`.**

Proof : Consider an optimal assignment λ^* with cost Δ^*. 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 δ_s. Hence, `BFlow` is a polynomial algorithm.

5 Ripple: an Adaptive Distributed Algorithm

In this section, we present a synchronous distributed algorithm, called `Ripple`, for `LDB` assignment. This algorithm is parametrized by the local assignment procedure `ALG` with approximation factor $r_{\texttt{ALG}}$ (e.g., `BFlow`) and the *desired* approximation ratio α, which is greater or equal to $r_{\texttt{ALG}}$. In the appendix, we formally prove the algorithm's correctness, and analyze its worst-case convergence time.

5.1 Overview

`Ripple` partitions the network into non-overlapping zones called *clusters*, and restricts user assignments servers residing in the same cluster (we call these *internal* assignments).

Initially, every cluster consists of a single server. Subsequently, clusters can grow through merging. The clusters' growth is congestion-sensitive, that is, loaded areas are surrounded by large clusters. This clustering approach balances between a centralized assignment, which requires

collecting all the user location 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 size also depends on α: the larger α is, the smaller the constructed clusters are.

Within each cluster, a designated *leader* server collects full information, and computes the internal assignment. A cluster's *cost* is defined as the maximum service delay among the users in this cluster. Only the leaders of neighboring clusters engage in inter-cluster communication, using small fixed-size messages. When two clusters merge, the leader of the cluster with the higher cost becomes the leader of the union.

`Ripple` enumerates the servers using a *locality-preserving* indexing. In this context, servers with close ids are also close to each other on the plane. Every cluster contains a contiguous range of servers with respect to this indexing. Two clusters C_i and C_j are called *neighbors* if there exists a k such that server s_k belongs to cluster C_i whereas server s_{k+1} belongs to cluster C_j. `Ripple` employs Hilbert's space-filling curve ((e.g., [16]), which is known for tight locality preservation. Figure 2 depicts a Hilbert indexing of 16 servers on a 4×4 grid, and a sample clustering that may be constructed by the algorithm.

We term a value ε, such that $\alpha = (1+\varepsilon)r_{\texttt{ALG}}$, as the algorithm's *slack factor*. A cluster is called ε-*improvable* with respect to `ALG` (denoted: *impr*) 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 an ε-approximation of the centralized assignment with `ALG`. Cluster C_i is said to *dominate* cluster C_j if:

1. $C_i.impr = \texttt{true}$, and
2. $(C_i.cost, C_i.impr, i) > (C_j.cost, C_j.impr, j)$, in lexicographic order.

`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.

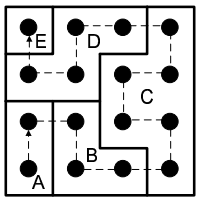


Figure 2: **Hilbert ordering of 16 servers on a 4×4 grid, and a sample clustering.**

Message	**Semantics**	**Size**
⟨ *"probe",id,cost,impr*⟩	Assignment summary (cost and ε-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`
Λ	the internal assignment	`NearestServer`
`Cost`	the cluster's cost	Δ^M(`NearestServer`)
`NbrId`[2]	the L/R neighbor cluster leader's id	$\{\texttt{Id}-1, \texttt{Id}+1\}$
`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 1: `Ripple`'s messages, constants, and state variables

A dominated cluster can only merge with a single neighbor and cannot split. 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 (type A conflict), or to be dominated by both neighbors (type B conflict). 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.

5.2 Detailed Description

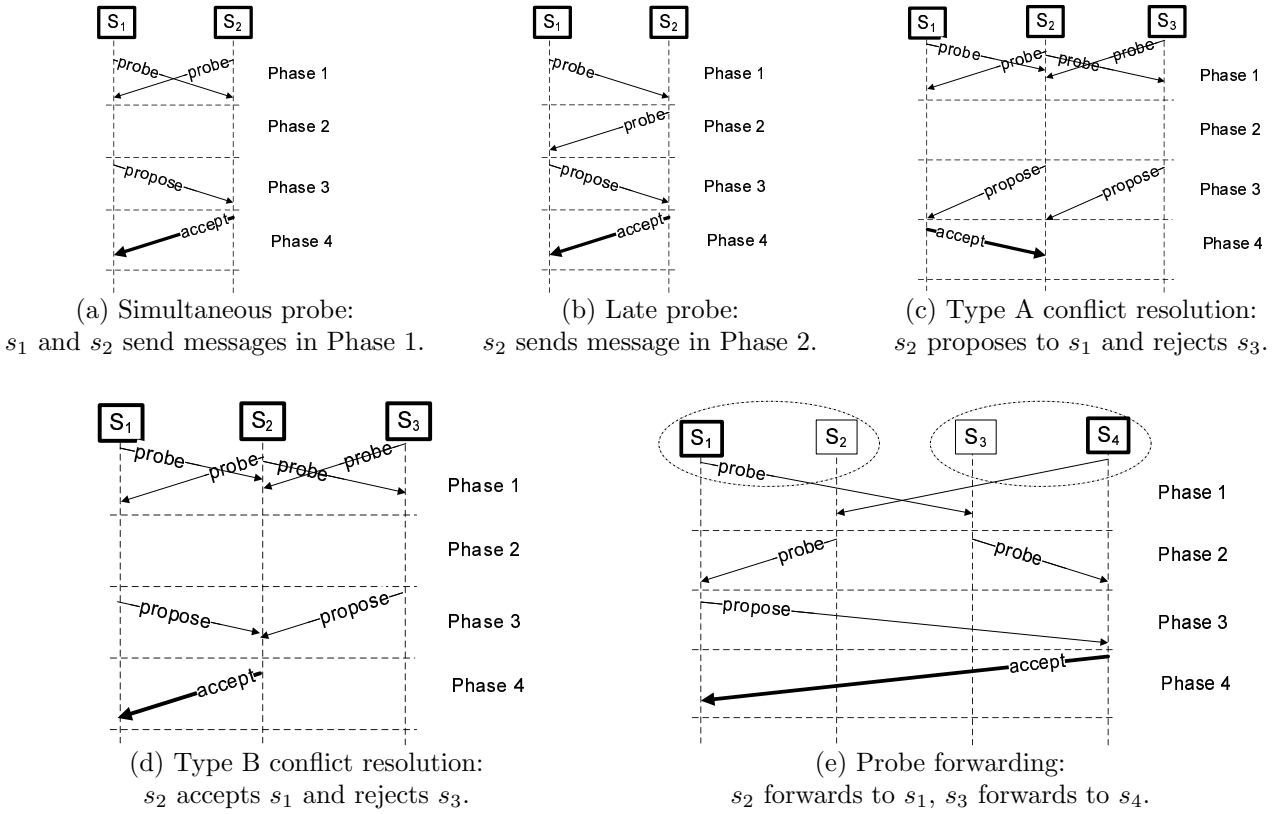
In this section, we present `Ripple`'s technical details. Table 1 provides a summary of the protocol's messages, constants, and state variables. See Figure 4 for the algorithm's pseudo-code. The code assumes the existence of local functions $\texttt{ALG} : (U, S) \to \lambda$, $\Delta^M : \lambda \to \mathbb{R}^+$, 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 info exchange *"probe"* messages with this info. 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 3(a) depicts a simultaneous mutual probe scenario. 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 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 (Lines 17–18). Figure 3(e) depicts this scenario: server s_2 forwards a message from s_1 to s_4, and server s_3 forwards the 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 3(c) and 3(d) illustrate the resolution



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

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

(c) Type A conflict resolution: s_2 proposes to s_1 and rejects s_3.

(d) Type B 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 3: **`Ripple`'s execution scenarios. Nodes in solid frames are cluster leaders. Dashed ovals encircle servers in the same cluster.**

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 will be the consuming cluster's neighbor unless it is itself consumed in the current round. The latter situation is addressed by the forwarding mechanism in Phase 2, as illustrated by Figure 3(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 it decides to send a *"probe"* message to both neighbors in the next round (Lines 70–71).

In the appendix, we prove that `Ripple`'s global convergence time is at most $k-1$ rounds. This theoretical upper bound bound 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. However, this scenario is very unrealistic. Indeed, our case study (Section 6) shows that in practice, `Ripple`'s *average* convergence time and cluster size remain flat as the network grows, whereas the growth rate of the respective *maximal* metrics is approximately logarithmic with k.

6 Simulation Results

In this section, we employ `Ripple` for gateway assignment in an urban WMN environment, using `BFlow` as a local assignment procedure. In most experiments, the simulated network spans a square area of size 16×16 km^2. This area is partitioned into 8×8 square cells of size 2×2km^2 each. There is an Internet gateway in the center of each cell. The delay is the following linear function of Euclidean distance: $D(u,s) = \frac{100\text{ms}}{\sqrt{2}\text{km}} d_2(u,s)$, that is, the delay between a user in the corner of a cell and the cell's gateway is 100 ms. The congestion delay of every gateway is equal to the load: $\delta_s(\mathcal{L}(s)) = \mathcal{L}(s)$. For example, consider a workload of 6400 uniformly distributed users in this network (e.g., 100 users in a cell on average). With high probability, there is some user close to the corner of each cell. Hence, the `NearestServer` heuristic yields an expected maximum delay which is close to $100 + 100 = 200$ms (i.e., the two delay types have equal contribution). While `NearestServer` is good for a uniform distribution, it is grossly suboptimal for skewed workloads. In our simulations, we test `Ripple` with varying distributions of heavy user load.

We term a normal distribution with variance R around a randomly chosen point on a plain as *congestion peak* $p(R)$. R is termed the *effective radius* of this peak. Every experiment employs a superposition of two workloads: $U(n_1)$, consisting of n_1 users uniformly distributed in the grid, and

```
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, impr⟩ do
9:     handleProbe(id, cost, impr)

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, impr⟩ do
20:     handleProbe(id, cost, impr)

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
      end:
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, impr)
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, impr)

51:     Accept[dir] ←
          dominates(id, cost, impr, Id, Cost, improvable(Λ, ε))

52:     Probe[dir] ← Propose[dir]
53:   else
54:     ProbeFwd[dir] ← true

55: procedure resolveConflicts()
56:   { Resolve type A conflicts: ⇐⇐ or ⇒⇒}
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 type B conflicts: ⇒⇐}
64:   if (Accept[L] ∧ Accept[R]) 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(id_1, cost_1, impr_1,
                       id_2, cost_2, impr_2)
73:   return (impr_1 ∧
          (impr_1, cost_1, id_1) > (impr_2, cost_2, id_2))

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

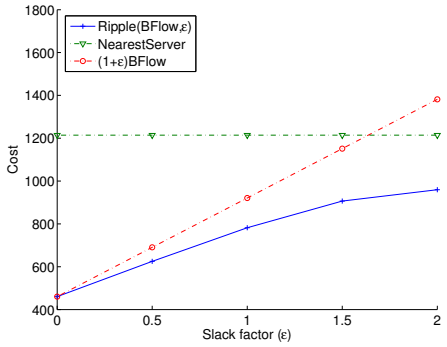
Figure 4: Ripple's **pseudo-code: single round.**

$P(n_2, k, R)$, consisting of n_2 users uniformly distributed among k congestion peaks $p(R)$.

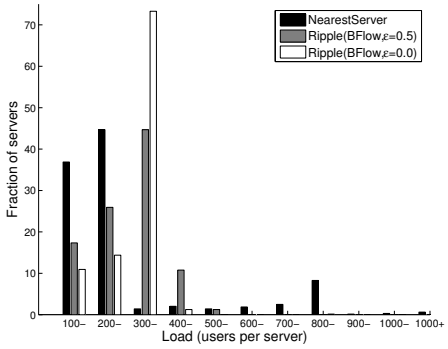
Sensitivity to slack factor We evaluate how `Ripple`'s cost, convergence time and locality depend on the slack factor's value. The workload is $\{U(6400), P(6400, 10, 200\text{m})\}$, i.e., a mix of a uniform distribution of 6400 users with 6400 users spread among ten peaks of effective radius 200m. We consider the values $0 \leq \varepsilon \leq 2$. The results show that `Ripple` significantly improves the cost achieved by `NearestServer`, and its cost is also well below the theoretical upper bound of $(1 + \varepsilon)$ times the cost of `BFlow` (Figure 5(a)). Figure 5(b) depicts the density of load distribution among the servers after running `NearestServer`, `Ripple`(`BFlow`, 0), and `Ripple`(`BFlow`, 0.5). The distribution after `NearestServer` (which is also the initial distribution for `Ripple`) is bimodal, that is, the majority of servers are lightly loaded, whereas some of them are congested. Both instances of `Ripple` "compress" this distribution, i.e., the variance of load values drops. The resulting assignment is not perfectly load-balanced because distances are a constraint. We can also see that the for $\varepsilon = 0.5$, the load distribution is more stretched, because the algorithm achieves its target cost earlier than for $\varepsilon = 0$.

Figure 6 demonstrates the dependency between the slack factor and `Ripple`'s convergence speed. The global (maximal) and local (average) convergence times are measured in rounds. We see that although theoretically `Ripple` may require a linear number of rounds to converge, in practice it completes much faster. On average, servers do not communicate for more than two rounds for all $\varepsilon \geq 0.5$. As observed before, the whole system converges faster as ε is increased. The algorithm's locality, i.e., the number of servers that each server communicates with, is expressed by the cluster size upon the algorithm's completion. Figure 6(b) depicts how the maximal and average cluster sizes depend on ε. The average size does not exceed 2.5 servers for $\varepsilon \geq 0.5$, and the maximal size rapidly drops as ε increases, for the same reason as the convergence time.

Sensitivity to user distribution We study `Ripple`'s sensitivity to varying workload parameters, like congestion skew and the size of congested areas, for $\varepsilon = 0.5$. We first compare the cost of `Ripple` and `NearestServer` on a workload $\{U((1 - p) \times 12800), P(p \times 12800, 10, 200)\}$, for $0 \leq p \leq 1$, i.e., different partitions of 12800 users between the uniform and peaky distributions, the latter consisting

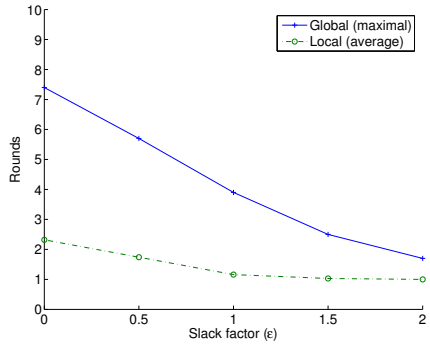


(a) Comparison of assignment cost

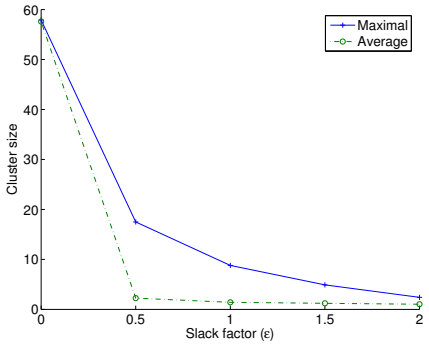


(b) Load distribution density

Figure 5: **Performance of** $\texttt{Ripple}(\texttt{BFlow}, \varepsilon)$**, for mixed workload: 50% uniform/50% peaky (10 peaks of effective radius 200m). (a)** `Ripple`**'s cost compared to** `NearestServer`**'s and the upper bound of** $(1+\varepsilon)$ **times** `BFlow`**'s cost, for** $0 \leq \varepsilon \leq 2$**. (b) Density of load distribution on servers after running** `NearestServer`**,** $\texttt{Ripple}(\texttt{BFlow}, 0)$**, and** $\texttt{Ripple}(\texttt{BFlow}, 0.5)$**.**

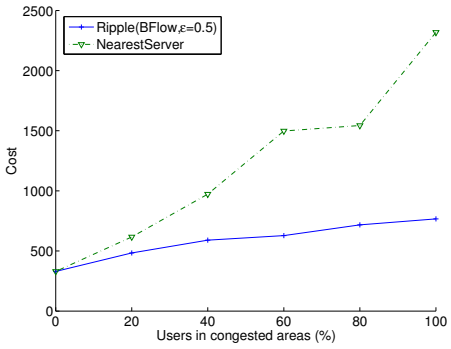


(a) `Ripple`'s convergence time (local and global)

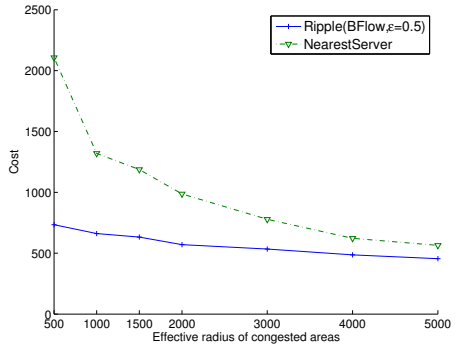


(b) `Ripple`'s cluster size (maximal and average)

Figure 6: **Convergence time (in rounds) and locality (cluster size) achieved by** $\texttt{Ripple}(\texttt{BFlow}, 0.5)$**, for mixed workload: 50% uniform/50% peaky (10 peaks of effective radius 200m).**



(a) Sensitivity to the number of users in congestion peaks



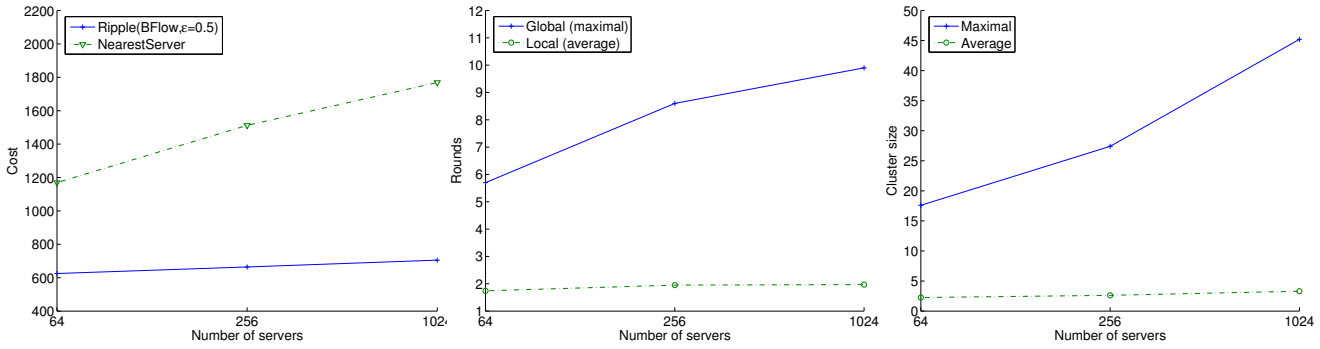
(b) Sensitivity to the radius of congestion peaks

Figure 7: **Sensitivity of the cost achieved by** `Ripple`$(\texttt{BFlow}, 0.5)$ **and** `NearestServer` **to user workload. (a) mixed workload: (100-p)% uniform/p% peaky (10 peaks of effective radius 200m), for** $0 \leq p \leq 100$**. (b) peaky workload (10 peaks of varying effective radius** $500\text{m} \leq R \leq 5000\text{m}$**).**

of ten peaks of effective radius 200m each. For $p = 0$ (100% uniform distribution), the algorithms achieve equal cost, because `Ripple` starts from the nearest-server assignment and cannot improve its cost. For larger values of p, `Ripple`'s cost remains almost flat, while `NearestServer` cannot adapt to increasingly peaky workloads. Following this, we compare the the two algorithms on a workload $\{P(12800, 10, R)\}$, for $500\text{m} \leq R \leq 5000\text{m}$, i.e., ten peaks of varying radius. For large values of R, this workload approaches to the uniform one, and consequently, `NearestServer` achieves a better cost than for more peaky distributions.

Sensitivity to network size We explore `Ripple`'s scalability, i.e., how the achieved cost, convergence and locality are affected as the network's size grows, for $\varepsilon = 0.5$ and a mixed 50%/50% workload. In this context, we gradually increase the network's size from 64 cells to 1024. Figure 8 depicts the results in log-scale. Similarly to the previous simulations, we first compare `Ripple`'s cost with the one achieved by `NearestServer` (Figure 8(a)). `NearestServer`'s cost grows logarithmically with the system's size although the workload remains the same. The reason for this is that the cost function is *maximum* delay. As the network grows, the expected maximum load among all cells also grows, which affects `NearestServer`'s cost. Since `Ripple` is more flexible than `NearestServer`, it adapts better to the network's growth.

Figure 8(b) and Figure 8(c) depict the dependency of `Ripple`'s convergence time and local-



(a) Comparison of assignment cost (b) `Ripple`'s convergence time (c) `Ripple`'s cluster size

Figure 8: **Scalability of `Ripple`(`BFlow`, 0.5) with the network's size (log-scale), for mixed workload: 50% uniform/50% peaky (10 peaks of effective radius 200m). (a) `Ripple`'s cost compared to `NearestServer`'s. (b) Convergence time (in rounds). (c) Locality (cluster size).**

ity metrics on the network's size. The average convergence time remains almost flat (about two rounds) as the network scales, as well as the average cluster size, which does not exceed 3.3 servers. The respective maximal metrics exhibit approximately logarithmic growth with the network's size, stemming from the increase in the expected maximum load.

7 Conclusion

We defined a novel load-distance min-max delay assignment problem, which is important for service access networks with multiple servers. In such settings, the service delay consists of a network-incurred delay, which depends on network distance, in addition to server-incurred delay, which arises from server load. While this problem is NP-hard, we presented a centralized 2-approximation algorithm for it, called `BFlow`. We then presented a scalable distributed algorithm, named `Ripple`, which computes a load-distance-balanced assignment with local information. `Ripple` employs `BFlow` as a subroutine. The algorithm's convergence time and communication requirements are congestion-sensitive, that is, they depend on the skew of congestion within the network and the size of congested areas, rather than the entire network size. We have studied `Ripple`'s practical performance in a large-scale WMN environment, which showed its significant advantage compared to naïve nearest-server assignment, as well as scalability with the network size.

Acknowledgements

We thank Seffi Naor for his contribution to proving the problem's NP-hardness and to the approximation algorithm. We also thank Ziv Bar-Yossef, Uri Feige, Isaac Keslassy and Yishay Mansour for fruitful discussions. We used the `boost` software package [1] for the max-flow algorithm implementation.

References

[1] Boost C++ Libraries. `http://www.boost.com`.

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

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

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

[5] M. Alicherry, R. Bhatia, and Li (Erran) Li. Joint Channel Assignment and Routing for Throughput Optimization in Multi-Radio Wireless Mesh Networks. *ACM Mobicom*, 2005.

[6] 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.

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

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

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

[10] J. Chen, B. Knutsson, B. Wu, H. Lu, M. Delap, and C. Amza. Locality Aware Dynamic Load Management form Massively Multiplayer Games. *Practices and Principles of Parallel Programming (PPoPP)*, 2005.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

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

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

[14] T. Moscibroda and R. Wattenhoffer. Facility Location: Distributed Approximation. *ACM Symposium on Principles of Distributed Computing (PODC)*, 2005.

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

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

A NP-Hardness of Load-Distance Balancing

In this section, we prove that the `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 Δ^* is feasible, i.e., $\Delta^M(\lambda(U)) \leq \Delta^*$.

In what follows, we prove the show a reduction from the *exact set cover* (`XSC`) problem [3]. 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.

Theorem 2 *The* `LDB−D` *problem is NP-hard.*

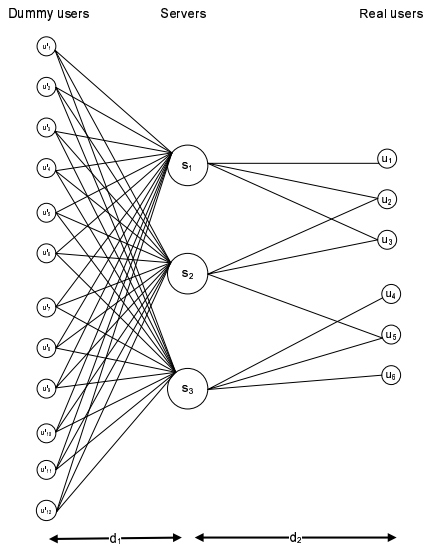


Figure 9: **Reduction from exact set cover to LDB-D.**

Proof : Consider an instance of XSC in which $|U| = n$, $|S| = k$, and each set contains exactly m elements. 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 9), in which the left side contains the elements in $U \bigcup U'$ (the users), and the right 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 ∞ 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 $\delta_s^{-1}(\Delta^* - d_2) = m$. It is easy to see that under a feasible assignment, no user's delay exceeds Δ^*.

Each server can cover either M dummy users, or any combination of $0 < m' \leq m$ original users and $m - m'$ dummy users. If both real and dummy users are assigned to at least one server, 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. The total number of assigned dummy users is therefore 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. □

B Correctness and Worst-Case 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.

Lemma 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 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 $\texttt{Probe}[L] = \texttt{Probe}[R] =$ `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 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 ε-improvable. This property cannot change in future rounds.
2. One cluster (e.g., C) is ε-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 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 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 2 (correctness and performance)

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

2. *The final (stable) assignment's cost of is an α-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 2, all communication would cease starting from the first round in which no messages are sent. By Lemma 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 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 ε-improvable, or it has a neighbor of equal cost that is not ε-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 $\alpha = r_{\texttt{ALG}}(1+\varepsilon)$. □