New Distributed Constraint Reasoning Algorithms for Load Balancing in Edge Computing

Khoi D. Hoang[1], Christabel Wayllace[1], William Yeoh[1], Jacob Beal[2], Soura Dasgupta[3], Yuanqiu Mo[3], Aaron Paulos[2], and Jon Schewe[2]

[1] Department of Computer Science and Engineering, Washington University in St. Louis
{khoi.hoang,cwayllace,wyeoh}@wustl.edu

[2] Raytheon BBN Technologies
{jake.beal,aaron.paulos,jon.schewe}@raytheon.com

[3] Department of Electrical and Computer Engineering, University of Iowa
{soura-dasgupta,yuanqiu-mo}@uiowa.edu

Abstract. Edge computing is a paradigm for improving the performance of cloud computing systems by performing data processing at the edge of the network, closer to the users and sources of data. As data processing is traditionally done in large data centers, typically located far from their users, the edge computing paradigm will reduce the communication bottleneck between the user and the location of data processing, thereby improving overall performance. This becomes more important as the number of Internet-of-Things (IoT) devices and other mobile or embedded devices continues to increase. In this paper, we investigate the use of distributed constraint reasoning (DCR) techniques to model and solve the distributed load balancing problem in edge computing problems. Specifically, we *(i)* provide a mapping of the distributed load balancing problem in edge computing to a distributed constraint satisfaction and optimization problem; *(ii)* propose two DCR algorithms to solve such problems; and *(iii)* empirically evaluate our algorithms against a state-of-the-art DCR algorithm on random and scale-free networks.

Keywords: DisCSPs · DCOPs · Edge Computing · Multi-Agent Systems

1 Introduction

Cloud computing is unequivocally the backbone of a large fraction of AI systems, where it provides computational functionality and data storage to the ever-growing number of Internet-of-Things (IoT), mobile, and embedded devices. In today's traditional cloud computing architecture, the compute and storage resources are typically housed in data centers that may be managed by different public and private organizations. For example, Amazon's AWS and Microsoft's Azure systems are two examples of popular cloud computing services that are provided by Amazon and Microsoft to the public for a fee.

As the number of IoT and similar devices continue to grow [6], so will the demand for cloud services. This increase in demand will eventually strain the bandwidth limitations to the data centers, thereby resulting in a drop in the quality of service of such services. To alleviate this limitation, researchers have proposed a new paradigm called
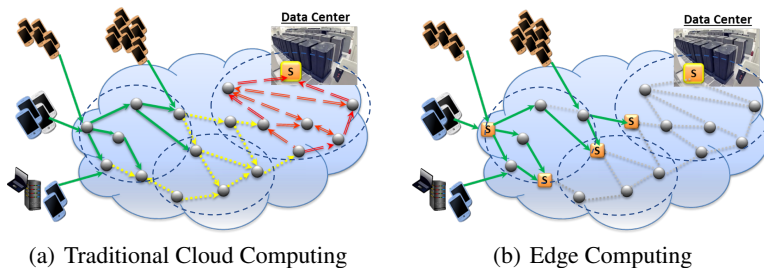
(a) Traditional Cloud Computing   (b) Edge Computing

FIG. 1: Illustration of Cloud & Edge Computing Paradigms. Figure adapted from [15]

edge computing, whereby the compute and storage resources are migrated from the data centers distant from users to compute resources that are closer to the user devices at the edges of the network. Figure 1(a) shows the traditional cloud computing paradigm, where the colors of the arrows denote the congestion in the network – green arrows represent uncongested links, yellow arrows represent marginally congested links, and red arrows represent very congested links. Figure 1(b) shows an edge computing paradigm, where services are hosted at resources at nodes labeled with 'S' and requests are routed to those nodes, resulting in decreased network congestion, and thus in increased performance of services. How to manage decisions about dispersal and placement of services in such a paradigm, however, is still an open problem.

In this paper, we model the distributed load balancing problem for edge computing as distributed constraint satisfaction and optimization problems, where agents that control nodes in the network need to coordinate to identify which node should host which services, subject to constraints on the capacity of the nodes and the requirement to satisfy all expected incoming requests. We also propose the *Distributed Constraint-based Diffusion Algorithm* (CDIFF) and *Distributed Routing-based Diffusion Algorithm* (RDIFF) to solve this problem and show results on random and scale-free graphs.

2   Background: DisCSP and DCOP

A *Distributed Constraint Satisfaction Problem* (DisCSP) [22] is a tuple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \alpha \rangle$, where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of *variables*; $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of finite *domains* (i.e., $x_i \in D_i$); $\mathcal{C} = \{c_1, \ldots, c_e\}$ is a set of *constraints* – each constraint c_i is defined over its *scope* $\mathbf{x}^{c_i} \subseteq \mathcal{X}$ and specifies the *satisfying* combination of value assignments in its scope; $\mathcal{A} = \{a_1, \ldots, a_p\}$ is a set of *agents*; and $\alpha : \mathcal{X} \to \mathcal{A}$ is a function that maps each variable to one agent.

A *Distributed Constraint Optimization Problem* (DCOP) [13, 16, 7] generalizes DisCSPs by encoding the constraints as functions $c_i : \prod_{x \in \mathbf{x}^{c_i}} D_x \to \mathbb{R}_0^+ \cup \{-\infty\}$ that return a finite non-negative utility $\mathbb{R}_0^+$ for satisfying combination of value assignments and a negative infinity utility $-\infty$ for infeasible combination of value assignments. To ease readability, we assume that each agent controls exactly one variable. Thus, we will use the terms "variable" and "agent" interchangeably and assume that $\alpha(x_i) = a_i$.

A *solution* is a value assignment σ for all the variables $\mathbf{x}_\sigma \subseteq \mathcal{X}$ that is consistent with their respective domains. The utility $\mathcal{C}(\mathbf{x}_\sigma) = \sum_{c \in \mathcal{C}, \mathbf{x}^c \subseteq \mathbf{x}_\sigma} c(\mathbf{x}_\sigma)$ is the sum of the

utilities across all the applicable constraints in $\mathbf{x}_\sigma$. A solution σ is a *complete solution* if $\mathbf{x}_\sigma = \mathcal{X}$. The goal of a DisCSP is to find a complete solution that satisfies all constraints while the goal of a DCOP is to maximize the sum of utilities across all constraints. Or, more formally, to find a complete solution $\mathbf{x}^* = \operatorname{argmax}_{\mathbf{x}} \mathcal{C}(\mathbf{x})$.

3   Load Balancing in Edge Computing

We now provide a simplistic description of the load balancing problem in edge computing architectures. For more detailed discussions, we refer readers to the following resources [17, 18, 21].

Assume that the network can be represented as a graph $G = \langle V, E \rangle$, where each vertex $v \in V$ corresponds to a compute node in the network that is able to host services and each edge $e \in E$ indicates that the two nodes connected by that edge can communicate directly with each other. Each node v has a capacity $cap(v)$ indicating the amount of available resources to host services. Some of the nodes in the graph are data centers, which are default nodes of these services. Some of the nodes $c \in C \subset V$ at the edge of the cloud are connected to pools of IoT devices, referred to as *client pools*. Further, we assume that an estimate of the load $load(v, s, c)$ of each service $s \in S$ induced by each client pool c at each node v is available.

The primary objective of the problem is to distribute the hosting of services across the compute network in such a way that all loads can be successfully served. The problem also has a secondary objective of minimizing the latency of the service requests where possible (i.e., services should be hosted as close to the client pools as possible).

4   Mapping to DisCSPs and DCOPs

We now show how one can model this problem as a DisCSP, which takes into account the primary objective only, as well as a DCOP, which additionally takes into account the secondary objective. To model this problem as a DisCSP:

- Each agent $a_v \in \mathcal{A}$ maps to a vertex $v \in V$ in the graph.
- Each variable $x_{v,s,c} \in \mathcal{X}$ maps to a pair $\langle v, s, c \rangle$ of vertex $v \in V$, service $s \in S$, and client pool $c \in C$.
- Each variable $x_{v,s,c}$ is controlled by agent a_v.
- The domain $D_{v,s,c}$ of each variable $x_{v,s,c}$ maps to the range of capacity $[0, cap(v)]$ of the vertex v.
- A constraint $\sum_{s \in S, c \in C} v_{v,s,c} \leq cap(v)$ for each vertex $v \in V$ is imposed to ensure that agent a_v does not over-allocate resources to host services, where $v_{v,s,c}$ is the value assignment for variable $x_{v,s,c}$ in the solution.
- Finally, a constraint $\sum_{v \in V, s \in S, c \in C} v_{v,s,c} \geq \sum_{v \in V, s \in S, c \in C} load(v, s, c)$ is imposed to ensure that the total load can be satisfied by the network.

To model this problem as a DCOP, one needs to include an additional global soft constraint that takes as inputs the value assignments of all variables and outputs the utility:

$$\sum_{v \in V, s \in S, c \in C} \frac{v_{v,s,c}}{dist(v, c)} \tag{1}$$

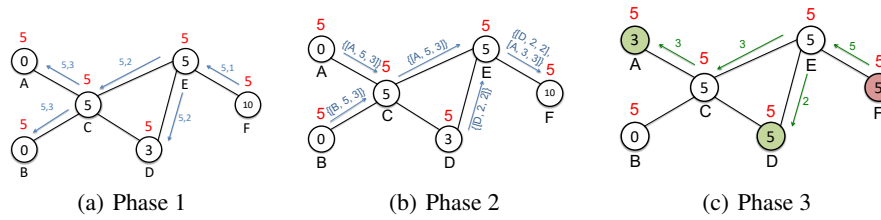(a) Phase 1   (b) Phase 2   (c) Phase 3

FIG. 2: Illustration of CDIFF Operations. Figure adapted from [15]

where $dist(v, c)$ is the number of hops between nodes v and c in the graph G. While it may be better to use the latency between two nodes as the distance metric, latency is dependent on network traffic, which depends on the allocation of services based on the DCOP solution as well as the background traffic. These dependencies also rely on the network protocols employed. Since accurate models of these dependencies are unavailable, we use the number of hops as a proxy in this paper.

As the DCOP model subsumes the DisCSP model and, consequently, the DCOP algorithms will likely find solutions that are better than those found by DisCSP algorithms, it may seem that there is little value in discussing DisCSPs in this paper. However, we would like to highlight that the number of hops between all pairs of nodes, which is required in the DCOP formulation, require every agent to have complete knowledge about the entire network topology. Such an assumption violates a common requirement of distributed constraint reasoning approaches as well as our distributed load balancing problem – that an agent should have access to local information only. Therefore, our DCOP model is actually not suitable for this application. Nonetheless, we propose the DCOP model so that we can evaluate our two DisCSP algorithms against an off-the-shelf optimal DCOP algorithm in terms of the quality of their solutions found with respect to the secondary objective of minimizing latency of service requests in the load balancing problem.

5   Proposed Algorithms

We now discuss our two DisCSP algorithms – *Distributed Constraint-based Diffusion Algorithm* (CDIFF) and *Distributed Routing-based Diffusion Algorithm* (RDIFF).

5.1   Distributed Constraint-based Diffusion Algorithm (CDIFF)

This algorithm is inspired by other diffusion-based algorithms in the literature [3, 14, 11]. At a high level, each overloaded agent (i.e., those agents that control nodes where $load(v) > cap(v)$) identifies to which subset of other agents that it should shed its excess load. Figure 2 illustrates its three phases, where numbers in circles are the current loads of the nodes and red numbers are the capacities. Node F is the overloaded agent, and nodes A, B, and D are possible nodes that can absorb the excess load from A.[1] We now describe the three phases at a high level:

---

[1] While the figure illustrates an example with only one overloaded region, our description below generalizes to the case where there are multiple overloaded regions.

Algorithm 1: CDIFF (**D**)

---

1 CDIFF-INITVARS()
2 $newMap \leftarrow hostMap \leftarrow excessMap \leftarrow \emptyset$
3 **foreach** (s, l) *in* **D do**
4 $\quad\lfloor\ newMap \leftarrow newMap \cup \{(s, l, hop)\}$
5 $(excessMap, hostMap) \leftarrow$ KEEPPOSSIBLE$(newMap)$
6 **if** $excessMap \neq \emptyset$ **then**
7 $\quad\lfloor\ phase \leftarrow 1$
8 send $(phase, excessMap)$ message to each neighbor $n \in$ **N**
9 **while** *true* **do**
10 $\quad$ **while** not received message from all neighbors **do**
11 $\quad\quad msgs \leftarrow \emptyset$
12 $\quad\quad$ **if** received message *m* from neighbor *n* **then**
13 $\quad\quad\quad\lfloor\ msgs \leftarrow msgs \cup \{(m, n)\}$
14 $\quad$ **if** $phase = 0$ **then** CDIFF-PROCESSPHASE0$(msgs)$;
15 $\quad$ **else if** $phase = 1$ **then** CDIFF-PROCESSPHASE1$(msgs)$;
16 $\quad$ **else if** $phase = 2$ **then** CDIFF-PROCESSPHASE2$(msgs)$;
17 $\quad$ send $(0, null)$ message to each neighbor that the agent did not send a message to in this cycle

---

Procedure CDIFF-InitVars()

---

18 $Reject \leftarrow \emptyset$
19 $parent \leftarrow msg \leftarrow null$
20 $hop \leftarrow phase \leftarrow 0$
21 $Children \leftarrow$ **N**

- **Phase 1:** Each overloaded agent sends a message to all neighboring agents with the amount of excess load it needs to shed as well as a hop count indicator that is initialized to 1.[2] When an agent receives such a message for the first time, it will propagate the received information to its neighbors after incrementing the hop counter by 1. The agent will then ignore subsequent Phase 1 messages by other neighbors and respond to them after Phase 3. This propagation of information continues until it reaches either nodes with sufficient capacity to accept the excess load or nodes that have received information from a closer overloaded agent. At the end of this phase, the agents have built a directed spanning forest, with roots at every overloaded agent.[3] In the example in Figure 2, node F is the sole root as it is the only overloaded agent and nodes A, B, and D are the leaves.
- **Phase 2:** Each leaf agent v of the spanning forest sends a message to its parent with its node ID, its available capacity $cap(v) - load(v)$, and the number of hops it is away from its root. When an agent receives this information from all its children, it aggregates the information received so that the sum of available capacities is at

---

[2] This indicator counts the number of hops a node is from the overloaded agent.

[3] If an agent receives this information from more than one neighbor at the same time, it breaks ties by identifiers.

Procedure CDIFF-ProcessPhase0($msgs$)

---

22   $(m, n) \leftarrow$ choose any $((type, map), sender)$ from $msgs$ where $type = 1$

23   $parent \leftarrow n$

24   $(excessMap, hostMap) \leftarrow$ KEEPPOSSIBLE($m.map$)

25   **foreach** $((type, map), n) \in msgs$ *where* $n \neq parent \wedge type = 1$ **do**

26      $Reject \leftarrow Reject \cup \{n\}$

27   send $(REJECT, null)$ message to n

28   $Children \leftarrow Children \setminus (Reject \cup \{parent\})$

29   **if** $parent \neq null$ **then**

30   **if** $excessMap \neq \emptyset \wedge Children \neq \emptyset$ **then**

31         $phase \leftarrow 1$

32   send $(phase, excessMap)$ message to each child $n \in Children$

33   **else**

34         $phase \leftarrow 2$

35         $newMap \leftarrow \emptyset$

36   **foreach** $(s, load, h)$ *in* $hostMap$ **do**

37            $newMap \leftarrow newMap \cup \{(s, load, h + 1)\}$

38   send $(phase, newMap)$ message to $parent$

---

most the amount of excess load needed to be shed, preferring nodes with smaller hop counts, and sends the aggregated information to its parent. This process continues until each root (i.e., the overloaded agent) receives the messages from all its children.

- **Phase 3:** Each root agent then sends a message to each of its children indicating the amount of excess load it intends to shed to them and their descendants in the spanning tree. This information gets propagated down to the leaves, which then terminates the algorithm. For example, in Figure 2, node F sheds 5 units of load – 2 units to node D and 3 units to node A.

These three phases continue until all overloaded regions successfully shed their loads.

Algorithm 1 shows the pseudocode of CDIFF that is executed by each agent $a_v \in \mathcal{A}$. It takes as inputs its estimated load $\mathbf{D}_v$, which is a mapping of (s, l) indicating a load of $l \in \mathbb{R}_0^+$ for service $s \in S$. Additionally, we assume that the agent always knows about the set of neighboring agents $\mathbf{N}_v$. In the pseudocode, we drop the subscripts v since they always refer to the "self" agent. Each agent maintains the following key data structures:

- $parent$ and $Children$ refer to the agent's parent and set of children, respectively, in the spanning forest built in Phase 1 of the algorithm.
- hop refers to the number of hops the agent is away from the root of its tree.
- $phase$ refers to the phase of the algorithm that the agent is currently in.
- $hostMap$ and $excessMap$ are sets of service s, load l, and hop h tuples (s, l, h); $hostMap$ contains the information on how much load from each service will it host and $excessMap$ contains the information on how much excess load for each service that it needs to shed.

Each agent first initializes its variables (lines 1-2). Then, it tries to host as much load as possible via the function KEEPPOSSIBLE (lines 3-5). The function takes as input the demand that it received aggregated with the hop value and updates how much

---

Procedure CDIFF-ProcessPhase1($msgs$)

---

39 **foreach** $((type, map), n) \in msgs$ **do**
40 | **if** $type = REJECT$ **then**
41 | | $Children \leftarrow Children \setminus \{n\}$
42 | **else if** $type = 1 \wedge n \neq parent$ **then**
43 | | $Children \leftarrow Children \setminus \{n\}$
44 | | send $(REJECT, null)$ message to n
45 | **else if** $type = 2$ **then**
46 | | store capacity availabilities from children

47 **if** $parent = null$ **then**
48 | **if** $Children = \emptyset$ **then**
49 | | send $(phase, excessMap)$ message to each neighbor $n \in \mathbf{N}$
50 | **else**
51 | | $phase \leftarrow 3$
52 | | $plan \leftarrow$ shed load in $excessMap$ to $Children$ prioritizing smaller hop counts
53 | | send $(phase, plan)$ message to each child $n \in Children$
54 **else if** $Children = \emptyset$ **then**
55 | $phase \leftarrow 2$
56 | $newMap \leftarrow \emptyset$
57 | **foreach** $(s, load, h)$ *in* $hostMap$ **do**
58 | | $newMap \leftarrow newMap \cup \{(s, load, h + 1)\}$
59 | send $(phase, newMap)$ message to $parent$
60 **else**
61 | $phase \leftarrow 2$
62 | $aggPlan \leftarrow$ aggregated capacity availabilities from children
63 | send $(phase, aggPlan)$ message to $parent$

---

of the demand it will host in $hostMap$ as well as how much excess it must shed in $excessMap$. If it can host all demand, then it remains in Phase 0. Otherwise, it is overloaded and goes into Phase 1 (line 7). It then sends a message to each neighbor and goes into an infinite loop (or until timeout) where it runs the following processes in each cycle: It waits for messages from all neighbors; processes those messages based on its phase; and sends a message to each neighbor at the end of that process (lines 8-17).

If an agent is in Phase 0, it runs the CDIFF-PROCESSPHASE0 procedure. Upon receiving a message from a neighbor that is in Phase 1 (i.e., it is overloaded and is asking for help), the agent sets the neighbor as parent and checks how much excess load it can host from the parent. If the agent is able to host all the load, it then replies to its parent with its capacity availabilities (lines 34-38). Otherwise, it propagates the request from the parent to other neighbors to ask for help (lines 30-32). If it receives such a message from more than one neighbor, then it breaks ties randomly, chooses to help only one of them and rejects the other requests (lines 22-27). Therefore, the request from the overloaded region (i.e., root of a tree) will be propagated throughout the network until it reaches agents with neighbors that are all not in Phase 0 (i.e., they are already part of the spanning forest). In such a case, the leaf agents will go into Phase 2 and respond to its parent with its available capacity (lines 34-38).

Procedure CDIFF-ProcessPhase2($msgs$)

---

64 **foreach** $((type, map), n) \in msgs$ **do**
65   **if** $type = 1$ **then**
66   send $(REJECT, null)$ message to n
67   **else if** $type = 3$ **then**
68         $phase \leftarrow 3$
69         $(excessMap, hostMap) \leftarrow$ FOLLOWPLAN(map)
70         $plan \leftarrow$ shed load in $excessMap$ to $Children$ prioritizing smaller hop counts

71 send $(phase, plan)$ message to each child $n \in Children$
72 CDIFF-INITVARS()

If an agent is in Phase 1, it runs the CDIFF-PROCESSPHASE1 procedure, where it goes into Phase 2 and send the aggregate available capacity received from all children to the parent (lines 54-63). This process continues until the information reaches the root, which will go into Phase 3, plans for how to shed its excess load, and sends the final plans back to its children (lines 51-53).

If an agent is in Phase 2, it runs the CDIFF-PROCESSPHASE2 procedure, where it goes into Phase 3, hosts as much load as possible based on the plan received from the parent, plans for how to shed its excess load based on the available capacities received from its children before, and sends those plans to its children (lines 68-71). The agent then reinitializes its variables, goes back into Phase 0 and is ready to help with new overloaded agents (line 72).

5.2 Distributed Routing-based Diffusion Algorithm (RDIFF)

One limitation of CDIFF is that it does not take into account information of where the client pools are located when deciding where the overloaded agents should shed its load. As such, it is not able to optimize the secondary objective of our problem. RDIFF addresses this limitation by shedding not only the excess load of overloaded agents, but as much load as possible to the agents that are of close proximity to the client pools. To do so, the agents operate in the following manner:

- **Phase 1:** Each data center propagates its entire load received from each client pool back towards that client pool by back-tracing the paths the requests took from the client pool to the data center. At the end of this phase, the agents have built a directed graph, where each branch of the graph corresponds to the path requests from a client pool took to get to a data center.[4]

- **Phase 2:** Each client pool will host as much of the load it received as possible, up to its capacity, and sheds its excess load to its parent (the next node along the branch from client pool towards the data center). This process repeats until all of the excess load is hosted. In the worst case where none of the agents along the branch has excess capacity, the data center will host the entire load.

Algorithm 2 shows the pseudocode of RDIFF that is executed by each agent $a_v \in \mathcal{A}$. It takes as inputs its estimated load $\mathbf{D}_v$, which is a set of mappings (s, l, c) indicating

---

[4] If there are multiple paths per client pool, we randomly choose one of them.

Algorithm 2: RDIFF (**D**)

---

73    $Parents \leftarrow Children \leftarrow \emptyset$
74    $hostMap \leftarrow pushMap \leftarrow excessMap \leftarrow 2DCMap \leftarrow \emptyset$
75   **foreach** (s, l, c) *in* **D do**
76   **if** *c is the "self" agent* **then**
77          $2DCMap \leftarrow 2DCMap \cup \{(s, l, c, c)\}$
78   **else**
79          $pushMap \leftarrow pushMap \cup \{(s, l, c, r)\}$

80    $(2DCMap, hostMap) \leftarrow$ KEEPPOSSIBLE$(2DCMap)$
81   **while** *true* **do**
82   **foreach** $(s, l, c, r) \in pushMap$ **do**
83   **if** $\nexists n : (s, n, c, r) \in Children$ **then**
84             $Children \leftarrow Children \cup \{(s, n, c, r)\}$ where n is the neighbor that sent
                   the request for service s from client pool c to server r
85   send $(1, (s, l, c, r))$ message to child n where $(s, n, c, r) \in Children$

86   **foreach** $(s, l, c, r) \in excessMap$ **do**
87   send $(2, (s, l, c, r))$ message to parent n where $(s, n, c, r) \in Parents$

88   **foreach** $(s, l, c, r) \in 2DCMap$ **do**
89   send $(DC, (s, l, c, r))$ message to DC(s)

90   send $(null, null)$ message to each neighbor that the agent did not send a message to
         in this cycle
91   **while** not received message from all neighbors **do**
92          $msgs \leftarrow \emptyset$
93   **if** received message *m* from neighbor *n* **then**
94             $msgs \leftarrow msgs \cup \{(m, n)\}$

95   RDIFF-PROCESSMESSAGES$(msgs)$;

---

a load of $l \in \mathbb{R}_0^+$ for service $s \in S$ from client pool $c \in \mathcal{A}$. Note that this estimated load is different than that in CDIFF, where it now includes the information per client pool. Like for CDIFF, we assume that the agent knows about its set of neighboring agents $\mathbf{N}_v$. Also, the agent knows the data center DC(s) for each service $s \in S$. In the pseudocode, we drop the subscripts v since they always refer to the "self" agent.

Each agent maintains the following key data structures:

- $Parents$ refers to the sets of service s, agent a, client pool c, and server r tuples (s, a, c, r), where the agent a is the agent's parent for service s, client pool c, and server r in the directed graph built in Phase 1 of the algorithm.

- $Children$ refers to a similar set as $Parents$, except that the agent a in the tuple is the agent's child for service s, client pool c, and server r in the directed graph built in Phase 1 of the algorithm.

- $hostMap, pushMap, excessMap$, and $2DCMap$ are sets of service s, load l, client pool c, and server r tuples (s, l, c, r); $hostMap$ contains the information on how much load l from each service s whose request originated from client pool c towards server r will be hosted; $pushMap$ contains the information for how much load will be pushed from the server towards the client pool; $excessMap$ contains the infor-

---

Procedure RDIFF-ProcessMessages($msgs$)

---

96   **foreach** $((type, map), n) \in msgs$ **do**

97   **if** $type = 1$ **then**

98   **foreach** (s, l, c, r) *in map* **do**

99   **if** *c is the "self" agent* **then**

100            $(excessMap, hostMap) \leftarrow$ KEEPPOSSIBLE($\{(s, l, c, r)\}$)

101   **else**

102            $pushMap \leftarrow pushMap \cup \{(s, l, c, r)\}$

103   **if** $\nexists p : (s, p, c, r) \in Parents$ **then**

104            $Parents \leftarrow Parents \cup \{(s, n, c, r)\}$

105   **else if** $type = 2$ *or* $type = DC$ **then**

106   **foreach** (s, l, c, r) *in map* **do**

107   **if** *r is the "self" agent* **then**

108            $(2DCMap, hostMap) \leftarrow$ KEEPPOSSIBLE($\{(s, l, c, r)\}$)

109   **else**

110            $(excessMap, hostMap) \leftarrow$ KEEPPOSSIBLE($\{(s, l, c, r)\}$)

---

mation for how much load will be pushed from the client pool towards the server; and $2DCMap$ contains the information for how much load will be pushed from the server directly to the data center.

Each agent first initializes these variables (lines 73-74). Then, it tries to host as much of its own load as possible if it is the client pool. Excess load is aggregated into $2DCMap$ in preparation to be pushed towards the data center (line 76-77). Loads from other client pools are aggregated into $pushMap$ in preparation to be pushed out towards those client pools (line 79). It then goes into an infinite loop (or until timeout) where it runs the following processes in each cycle: It sends a message to each neighbor; waits for messages from all neighbors; and processes those messages (lines 81-95).

At the start, the agent will iterate through $pushMap$ and send a Phase 1 message containing the amount of load to be pushed for each service, client pool, and server combination to the appropriate child (lines 82-85). Upon receiving this information, the child will aggregate that information into its own $pushMap$ in preparation to be sent to its child in the next cycle (line 102). This process continues until it reaches the client pool, at which point, it hosts as much load as possible and stores the excess load in $excessMap$ (line 100).

In the next cycle, the client pool will iterate through $excessMap$ and send a Phase 2 message containing the load to be pushed towards the server to the appropriate parent (lines 86-87). Upon receiving this information, the parent will host as much load as well and pushes the excess to its parent in the next cycle (line 110). This process continues until all the load is hosted or it reaches the server. If the server does not have enough capacity to host all the load, then it stores the excess load in $2DCMap$ (line 108).

In the next cycle, the server will iterate through $2DCMap$ and send a DC message containing the amount to be pushed to the appropriate data center (line 89). As we assume that the data centers have infinite capacity, all the load will then be hosted.

6 Theoretical Results

We now discuss some of the theoretical properties of the algorithms, where we make the standard assumptions that: (1) messages sent are never lost and are received in the order that they were sent; and (2) there exists a path from each node of the network to every other node of the network.

Lemma 1. *In CDIFF, an agent with available capacity will eventually be part of the tree of an overloaded agent as long as one such overloaded agent exists.*

Proof (Sketch) : Since overloaded agents send Phase 1 messages that are propagated throughout the network, the agent with available capacity will eventually receive one such message and insert itself into the spanning tree of the overloaded agent that initiated the series of messages that it received.

Lemma 2. *In CDIFF, an overloaded agent will shed some of its excess load if its tree includes agents with available capacity.*

Proof (Sketch) : Since Phase 1 messages are repeatedly propagated throughout the network until they either reach agents with enough capacity to host all the overloaded services of the root agent or reach agents without any free neighbors (i.e., neighbors in Phase 0), the phase is guaranteed to end after a finite number of cycles since the network is of finite size. Then, Phase 2 messages are propagated from the leafs to the root of the tree, upon which the root sheds some of its load based on the available capacities of the agents in its tree in Phase 3.

Theorem 1. *CDIFF is guaranteed to find a satisfying solution if one exists.*

Proof (Sketch) : Based on Lemmas 1 and 2, it is easy to see that all overloaded agents will eventually succeed at shedding their load assuming that there exists agents with available capacity.

Theorem 2. *RDIFF is guaranteed to find a satisfying solution if data centers have infinite capacity.*

Proof (Sketch) : It is trivial to see that each server will successfully push its load to the client pools in Phase 1, and the agents along the path will host as much as possible in Phase 2. Should the combined capacities of the agents along the path be insufficient to host all the load, then the agent will send the excess load to the data center, which will be able to host it since it has infinite capacity.

Theorem 3. *If CDIFF and RDIFF finds a satisfying solution, it will take at most $O(|\mathcal{A}| \cdot d)$ cycles to do so, where d is the diameter of the network graph.*

Proof (Sketch) : In the worst case, the network is a chain of length d and every agent a_1, $\ldots$, a_{d-1} along the chain is overloaded except for one agent a_d at the end of the chain that has sufficient capacity. In this scenario, agent a_{d-1} will first succeed in shedding its load to a_d. Then, agent a_{d-2} will shed its load and so on until agent a_1 sheds its load. Each time an agent sheds its load, it will take $O(d)$ number of cycles to do so since each phase of the algorithms take $O(d)$ cycles and there is only a constant number of phases. Since there are $O(|\mathcal{A}|)$ overloaded agents, the total runtime is $O(|\mathcal{A}| \cdot d)$ cycles.

7   Related Work

Since our work is on the use of DCOPs for cloud computing applications, we will first discuss work at this intersection before broadening the discussion to other DCOP-based approaches on similar load balancing applications and other multi-agent approaches for cloud computing applications. Within this intersection, the work by Rust *et al.* [19] is most relevant, where they used DCOPs to model the problem of resiliently distributing computation nodes in edge computing scenarios. Specifically, given a dynamic network, where nodes in the network may fail and disappear over time, the goal is to identify k nodes to host replicated services (aside from the one currently hosting the service) such that the service is resilient to the node failures. There are several key differences between their work and ours: (1) Their approach allows for node failures while our approach assumes that the network remains unchanged over time. (2) Their approach seeks to only identify k replicas to host services and migrates services to one of the replicated nodes when the node hosting the service fails. In contrast, our approach seeks to distribute the load across all the nodes that are hosting services to ensure that all load can be served while optimizing for quality-of-service metrics like response times.

Within the broader application of DCOPs, aside from the many applications listed in the introduction, the most relevant one to our problem is the one on dynamic load balancing problems in wireless LANs [2]. In this problem, a set of access points need to coordinate and identify who should serve each mobile station in a set of such mobile stations. DCOPs are used to model this optimization problem, where the objective optimizes the received signal strength of each mobile station as well as distribute the load among all access points as evenly as possible. The key difference between their work and ours is that the sources of load in their problem are the mobile stations that physically move within an environment. In contrast, the sources of load in our problem are service requests made by clients within a fixed topology.

Finally, other multi-agent based approaches such as negotiation and auctions have also been used for resource allocation and load balancing problems in cloud computing [20, 4] and grid computing [8, 12]. The key difference is that these negotiation and auction-based approaches often assume that the agents are self-interested and seek to optimize their individual objective functions. In contrast, agents in our DCOP-based approach are completely cooperative, where the goal is to optimize a global objective function.

8   Experimental Results

In this paper, we empirically evaluate CDIFF and RDIFF algorithms against DPOP [16], a state-of-the-art complete DCOP algorithm. However, it is important to note that it will be impractical to use DPOP (or any other DCOP algorithm) as the information that they need to optimize their utility function, which is the number of hops between all pairs of nodes in the network, is often unavailable in practice. We therefore include the results of DPOP mostly as a way to quantify the quality of solutions found by CDIFF and RDIFF with respect to the optimal DCOP solution.

We evaluate the algorithms on random networks [5] with a density p_1 of 0.5 and scale-free networks [1], where we randomly choose a node as the data center in random

$|\mathcal{A}|$	CDIFF				RDIFF				DPOP
	A	B	C	Total	A	B	C	Total	Total
5	0.32	0.25	0.34	0.91	0.33	0.29	0.38	1.00	1.59
10	0.29	0.29	0.32	0.90	0.23	0.30	0.35	0.88	-
15	0.28	0.33	0.27	0.88	0.39	0.38	0.34	1.11	-
20	0.23	0.33	0.34	0.90	0.35	0.34	0.34	1.03	-

TABLE 1: Quality of Solutions on Random Networks

$|\mathcal{A}|$	CDIFF				RDIFF				DPOP
	A	B	C	Total	A	B	C	Total	Total
5	0.28	0.28	0.31	0.87	0.29	0.27	0.36	0.92	1.56
10	0.27	0.23	0.30	0.80	0.26	0.25	0.33	0.84	-
15	0.14	0.15	0.18	0.47	0.19	0.18	0.23	0.60	-
20	0.21	0.21	0.23	0.65	0.20	0.19	0.24	0.63	-

TABLE 2: Quality of Solutions on Scale-Free Networks

networks and choose the node with the most number of neighbors as the data center in scale-free networks. The data center initially hosts three services A, B, and C, and will then redirect the service request to other nodes by following the solutions of the algorithms. We randomly place three client pools in both random and scale-free networks. Each client pool makes 10 batches of service requests, and each batch starts a minute after each other. Each batch has 20 requests per service, and each request induces a load of 0.1 units of resource. Each node has has a capacity of 20 units of resource, and the data center has a capacity of 8 units. For all three algorithms, we set the thresholded capacity of nodes to be 55% of their actual capacity. A node is a considered overloaded if its predicted load is greater than its thresholded capacity.

We vary the number of agents $|\mathcal{A}|$, which are nodes in the graph, from 5 to 20, and we report the quality of solution measured using the utility function defined by Equation (1) as well as the number of successful requests as a function of the number of hops between the client pool and the server that served the requests. All experiments were performed on an Intel Core i5, 2.0GHz machine with 8GB of RAM. Data points are averaged of over 20 instances.

Tables 1 and 2 tabulate the quality of solutions found by CDIFF, RDIFF, and DPOP on random and scale-free networks, respectively. As expected, DPOP finds the best solutions since it explicitly optimizes for the global utility function while CDIFF and RDIFF do not. Nonetheless, DPOP fails to solve the larger problems as it ran out of memory. In both networks, RDIFF is often able to find better solutions with larger utilities than CDIFF. The reason is that RDIFF sheds the excess load towards the edge, closer to the client pools. On the other hand, CDIFF sheds the excess load to nodes around the data center, which tends to be further away from the client pools. The difference in solution qualities between RDIFF and CDIFF is more pronounced in scale-free networks than in random networks. The reason is that the distance between client pools and data centers is often times larger in scale-free networks than in random graphs.

$|\mathcal{A}|$	CDIFF					RDIFF					DPOP				
	0	1	2	3	Total	0	1	2	3	Total	0	1	2	3	Total
5	1805	683	845	127	3460	10407	230	354	50	11041	1290	1084	307	0	2681
10	1307	1241	728	59	3335	11652	503	296	49	12500	-	-	-	-	-
15	522	1391	799	49	2761	10607	976	523	33	12139	-	-	-	-	-
20	107	1286	977	0	2370	1362	1032	851	0	3245	-	-	-	-	-

TABLE 3: Number of Successful Requests on Random Networks

$|\mathcal{A}|$	CDIFF				RDIFF				DPOP			
	0-3	4-6	7-10	Total	0-3	4-6	7-10	Total	0-3	4-6	7-10	Total
5	3681	0	0	3681	11687	0	0	11687	4576	0	0	4576
10	2929	187	0	3116	11171	92	0	11263	-	-	-	-
15	1834	452	30	2316	12065	181	0	12246	-	-	-	-
20	2315	383	74	2772	11359	172	24	11555	-	-	-	-

TABLE 4: Number of Successful Requests on Scale-Free Networks

Tables 3 and 4 tabulate the number of successful requests as a function of the number of hops between the client pool and the server that served the requests. We make the following observations:

- RDIFF has a larger fraction of successful requests served closer to the client pool compared to CDIFF. In fact, in most cases, more than 80% of the requests were served in the node that is also the client pool (when the number of hops is zero). This observation is to be expected since RDIFF pushes the load towards the client pools while CDIFF diffuses the load around the data center.
- All three algorithms did not succeed in successfully serving all requests. A request is considered to be a failed request if it is not served within a prescribed time window, which occurs when the request is directed to a node that is busy serving other requests and have a large number of pending requests in its queue. This occurs during the execution of the algorithms before they found a load-balanced solution. During the execution time of the algorithms, the agents execute the default strategy of serving all requests at the data center, which is overloaded. As a result, the longer the execution time of an algorithm, the larger the number of failed requests due to the default strategy. Another reason for failed request is a mismatch between the *actual load*, which is based on the actual number of requests, and the *estimated load*, which is based on the number of requests in the past. Since the load-balanced solutions are based on estimated loads, some requests may fail if the actual load is underestimated.
- DPOP has the smallest number of successful requests compared to CDIFF and RDIFF. The reason is because it has the longest runtime. As a result, it has the most number of requests being directed to an overloaded data center.
- RDIFF has a larger number of successful requests than CDIFF. The reason is because there are more nodes with loads that are closer to the thresholded capacity in CDIFF than in RDIFF. In CDIFF, loads from all three services are congregated around the data center. In contrast, in RDIFF, load from each service is dissipated towards the client pool for that service.

In summary, these empirical results show that RDIFF is better than CDIFF in terms of both the DCOP utility function of, despite both algorithms not optimizing that function explicitly, as well as the number of successful requests served.

9   Conclusions and Future Work

In this paper, we proposed a distributed constraint reasoning approach to model and solve a distributed load balancing problem in edge computing. Our two algorithms, Distributed Constraint-based Diffusion Algorithm (CDIFF) and Distributed Routing-based Diffusion Algorithm (RDIFF), are guaranteed to find satisfying solutions (i.e., all the estimated load will be served by nodes in the network) under certain conditions. Further, despite not optimizing for the global objective function explicitly, because it is impractical to do so as the information needed to do so (= number of hops between all pairs or nodes in the network) is often unavailable in practice, RDIFF still found solutions that are within 60% of optimal. Experimental results also show that both CDIFF and RDIFF can scale better than DPOP, a state-of-the-art DCOP algorithm, and that RDIFF is better than CDIFF in terms of the number of successful requests served.

Future work includes integrating CDIFF and RDIFF into a single algorithm, where, like RDIFF, data centers propagate their entire loads received to their respective client pools. However, unlike RDIFF, which allocates the loads to the agents along the paths from the client pools to the data centers only, the integrated algorithm allocates the loads *around* the client pools like in CDIFF. This
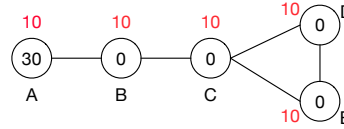

FIG. 3: Motivating Scenario for Integrating CDIFF and RDIFF

will likely result in more load being hosted closer to the client pools. Figure 3 illustrates such a motivating example, where, like the example in Figure 2, numbers in circles are the current loads of the nodes and red numbers are the capacities. In this example, node A is the data center serving 30 units of load from a client pool at node D. RDIFF would have allocated 10 units of load to nodes B, C, and D, and cannot consider allocations to E since it is not on the path from A to D. In contrast, a better solution would be to allocate 10 units of load to C, D, and E, and such a solution would be found by the integrated algorithm. Finally, we also plan to improve all of hese algorithms so that they are more resilient to dynamic changes [19] as well as proactively take into account anticipated future changes [9, 10].

10   Acknowledgment

References

1. Barabási, A.L.: Scale-free networks: a decade and beyond. Science **325**(5939), 412–413 (2009)
2. Cheng, S., Raja, A., Xie, J., Howitt, I.: DLB-SDPOP: A multiagent pseudo-tree repair algorithm for load balancing in WLANs. In: Proceedings of WIIAT. pp. 311–318 (2010)
3. Cybenko, G.: Dynamic load balancing for distributed memory multiprocessors. Journal of Parallel and Distributed Computing **7**(2), 279–301 (1989)
4. Du, L., Bigham, J., Cuthbert, L., Nahi, P., Parini, C.: Intelligent cellular network load balancing using a cooperative negotiation approach. In: Proceedings of WCNC. pp. 1675–1679 (2003)
5. Erdös, P., Rényi, A.: On random graphs, i. Publicationes Mathematicae (Debrecen) **6**, 290–297 (1959)
6. Evans, D.: The internet of things: How the next evolution of the internet is changing everything. CISCO White Paper **1**(2011), 1–11 (2011)
7. Fioretto, F., Pontelli, E., Yeoh, W.: Distributed constraint optimization problems and applications: A survey. Journal of Artificial Intelligence Research **61**, 623–698 (2018)
8. Grosu, D., Das, A.: Auction-based resource allocation protocols in grids. In: Proceedings of ICDCS. pp. 20–27 (2004)
9. Hoang, K.D., Fioretto, F., Hou, P., Yokoo, M., Yeoh, W., Zivan, R.: Proactive dynamic distributed constraint optimization. In: Proceedings of AAMAS. pp. 597–605 (2016)
10. Hoang, K.D., Hou, P., Fioretto, F., Yeoh, W., Zivan, R., Yokoo, M.: Infinite-horizon proactive dynamic DCOPs. In: Proceedings of AAMAS. pp. 212–220 (2017)
11. Hu, Y., Blake, R.: An optimal dynamic load balancing algorithm. Tech. rep., SCAN-9509056 (1995)
12. Izakian, H., Abraham, A., Ladani, B.T.: An auction method for resource allocation in computational grids. Future Generation Computer Systems **26**(2), 228–235 (2010)
13. Modi, P., Shen, W.M., Tambe, M., Yokoo, M.: ADOPT: Asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence **161**(1–2), 149–180 (2005)
14. Muthukrishnan, S., Ghosh, B., Schultz, M.H.: First-and second-order diffusive methods for rapid, coarse, distributed load balancing. Theory of Computing Systems **31**(4), 331–354 (1998)
15. Paulos, A., Dasgupta, S., Beal, J., Mo, Y., Hoang, K., Lyles, J.B., Pal, P., Schantz, R., Schewe, J., Sitaraman, R., Wald, A., Wayllace, C., , Yeoh, W.: A framework for self-adaptive dispersal of computing services. In: IEEE Self-Adaptive and Self-Organizing Systems Workshops (2019)
16. Petcu, A., Faltings, B.: A scalable method for multiagent constraint optimization. In: Proceedings of IJCAI. pp. 1413–1420 (2005)
17. Puthal, D., Obaidat, M.S., Nanda, P., Prasad, M., Mohanty, S.P., Zomaya, A.Y.: Secure and sustainable load balancing of edge data centers in fog computing. IEEE Communications Magazine **56**(5), 60–65 (2018)
18. Rabinovich, M., Xiao, Z., Aggarwal, A.: Computing on the edge: A platform for replicating internet applications. In: Web Content Caching and Distribution, pp. 57–77. Springer (2004)
19. Rust, P., Picard, G., Ramparany, F.: Self-organized and resilient distribution of decisions over dynamic multi-agent systems. In: International Workshop on Optimization in Multiagent Systems (2018)
20. Shen, W., Li, Y., Ghenniwa, H., Wang, C., et al.: Adaptive negotiation for agent-based grid computing. Journal of the American Statistical Association **97**(457), 210–214 (2002)
21. Shi, W., Dustdar, S.: The promise of edge computing. Computer **49**(5), 78–81 (2016)
22. Yokoo, M., Durfee, E., Ishida, T., Kuwabara, K.: Distributed constraint satisfaction for formalizing distributed problem solving. In: Proceedings of ICDCS. pp. 614–621 (1992)