

An Improved dynamic Load Balancing Algorithm applied to a Cafeteria System in a University Campus

Eman Daraghmi
Department of applied Computing
Palestine Technical University
Palestine
e.daraghmi@ptuk.edu.ps

Amna Eleyan
Department of Computer Science and
Information Systems
Manchester Metropolitan University
Manchester-UK
a.eleyan@mmu.ac.uk

ABSTRACT

Load-balancing algorithms play a key role in improving the performance of practical distributed systems that consist of heterogeneous nodes. The performance of any load-balancing algorithms and its convergence-rate is affected by the structural factors of the network that executes the algorithm. The performance deteriorated as the number of system nodes, the network-diameter, the communication-overhead increased. Moreover, additional technical-factors of the algorithm itself significantly affect the performance of rebalancing the load among nodes. Therefore, this paper proposes an approach that improves the performance of load-balancing algorithms by considering the load-balancing technical-factors and the structure of the network executes the algorithm. We applied the proposed method to a cafeteria system in a university campus and compared our approach with two significant methods presented in the literature. Results indicate that our approach considerably outperformed the original neighborhood approach and the nearest neighbor approach in terms of response time, throughput, communication overhead, and movements cost.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics • **Networks** → Network reliability

KEYWORDS

Small world, load balance, distributed systems, queue management systems

1 INTRODUCTION

Recently, dynamic load-balancing algorithms have become increasingly popular and powerful techniques in modern distributed computing systems [7]. They increase the performance of large-scale computing systems and applications since they are designed to redistribute the workloads over the components of the distributed system in a way that ensures expanding resource utilization, maximizing throughput, minimizing response time, and avoiding the overload situation [2]. To achieve the goal of maximum performance, it is prerequisite to smoothly spread the load among the nodes to

avoid, if possible, the situation where one node is heavily loaded with excess of workloads while another nodes are lightly loaded or idle [8,19].

Dynamic load-balancing algorithms are suitable to be applied in practical applications, such as the queue managements systems since the workloads are generally not completely known, and each node has different capacity and runs at different speed. The diffusion approach [13,20] is one of the dynamic load balancing techniques that have received much attention by researchers in the past decades to solve the load-balancing problem. In standard diffusion approach, a system which has different nodes exchanges workloads via the communication links between these nodes. The workloads are distributed among the nodes, and the load balancing process works in sequential rounds. In every round, each node is allowed to balance its load with all its neighbors by exchanging the workloads to balance the total system load globally, meaning to minimize the load difference between the nodes with minimum and maximum load. The nearest-neighbor approach [24] is another dynamic technique that allows the nodes to communicate and migrate the excess workloads with their immediate neighbors only. Each node balances the workload among its neighbors in the hope that after a number of iterations the entire system will approach the balanced state. However, dynamic load-balancing algorithms still present fundamental challenges when being executed at large-scale heterogeneous distributed systems.

Previous research [14–16] concluded that three structural factors may affect the performance of any load-balance algorithm: increasing the number of nodes within the system, increasing the network diameter, and increasing the communication overheads. In addition, previous studies concluded that [27] technical load-balancing factors, such as the algorithm policies should be considered when designing a load-balancing algorithm. Thus, this research proposes an improved dynamic load-balancing algorithm that decreases the effect of the structural factors by constructing a small world overlay network. Moreover, our proposed algorithm considers the technical load-balancing factors, such as the initialization rule, the information exchange rule, the load-measurement rule and the load-migration rule. To prove the efficiency of the proposed approach, we applied the algorithm to a cafeteria system in a university campus as a case study.

In summary, to apply our proposed approach to any queue management system, first, an overlay network based on the small world theory should be construct to decrease the effect of the

structural factors, and then the load balancing algorithm will be applied within the constructed network. We have evaluated the performance of our proposed approach and compared it against competing algorithms. Results are encouraging, indicating that our proposed algorithm dramatically outperforms them in terms of response time, throughput, communication overhead, and movements cost.

2 RELATED WORK

Previous studies have proposed numerous load-balancing algorithms targeting at static, small-scale, homogeneous and/or heterogeneous environments [1,13,18,21,22,26]. The diffusion approach [13,22] is a dynamic load-balancing technique where each node simultaneously sends the excessive workloads to its under loaded neighbors and receives workloads from its neighbors with higher workload [6], [9]. In 1990, Boillat et al. [6] presented a new approach to solve the load balancing problem for parallel programs. In 1989, Cybenko [9] studied the diffusion schemes for dynamic load balancing on a message passing multiprocessor networks. Robert Elsasser et al [10] generalized the standard diffusion schemes for homogenous networks to deal with the heterogeneous network. In [5], the first order diffusion load balancing, relaxed diffusion (RFOS) and generalized adaptive exchange (GAE) algorithms for totally dynamic networks were investigated. In [1], the authors proposed a modified version of diffusion algorithm for load balancing on dynamic networks. The authors in [3] considered a neighborhood load balancing algorithm in the context of selfish clients. They assumed that a network of n processors is given, with m tasks assigned to the processors. The processors may have different speeds and the tasks may have different weights. Every task is controlled by a selfish user. The objective of the user is to allocate his/her task to a processor with minimum load, where the load of a processor is defined as the weight of its tasks divided by its speed. Neighborhood load balancing algorithms [4] are diffusion algorithm that have the advantage that they are very simple and that the vertices do not need any global information to base their balancing decisions on.

3 SYSTEM OVERVIEW

As mentioned before, to apply our proposed approach to any queue management system, first, an overlay network based on the small world theory should be construct to decrease the effect of the structural factors, and then the load balancing algorithm will be applied within the constructed network. This section details how to construct the Functional Small World (FSW) overlay network within the cafeteria system. The notations used in this paper is summarized in Table 1.

Table 1: The symbols used in the paper

Symbo l	Description
FSW	Functional Small World
FS	The Functionality Set
G	The system that executes the load-balancing algorithm
N	The nodes in the system
E	The connection-links among nodes

AF	All Functions set
$WL(n_i)$	The set of assigned workloads for node n_i
c_i	The capacity of node n_i
ld_i	The load of node n_i
$Adj(n_i)$	The set of neighbor nodes for node n_i
<i>Info</i>	The set stored the information of neighbor nodes for node n_i
<i>mig</i>	The array that store the amount of migrated workloads
l_i	The effective-load of node n_i
l_{avg}	The average effective-load
N_{lower}	The set of assistant neighbors
<i>LD</i>	The load difference
δ_i	The excess workloads that node n_i must migrate
α_i	The amount of workloads that node n_i can accept

3.1 Overview

In our research, FSW has two important goals: 1) an overlay network that provides connectivity among the cafeteria nodes, and 2) a distributed solution that supports efficient dynamic load-balancing among cafeterias. In FSW, the nodes are organized in accordance with the Functionality Set (FS) defined by each node in the system. In a cafeteria system, the functionality set refers to the set of meals offered by each cafeteria node. Cafeteria nodes with similar functionality sets form one cluster. We based on the concept proposed by Tversky [25] to define the relation of “similar functionality” employed in our research.

Definition 1 (similar functionality). Generally, similar functionality is defined as the difference between the amount of functions in-common among nodes and the amount of functions unique to nodes. Formally, given any nodes $n_i, n_j \in N$ with a functionality set of each node FS_i, FS_j , the relation of similar functionality is defined by:

$$s(n_i, n_j) = |FS_{n_i} \cap FS_{n_j}| - (|FS_{n_i} - FS_{n_j}|) - (|FS_{n_j} - FS_{n_i}|) .$$

Therefore, nodes with $s(n_i, n_j) < 0$ are not similar, while nodes with $s(n_i, n_j) \geq 0$ are similar.

It is clear that functions in common increase similarity, whereas functions that are unique to one node decrease similarity.

In practice, the university cafeteria system is considered as a distributed system that can be modeled as an undirected graph $G = (N, E)$ where N represents the set of heterogeneous cafeteria nodes in the system and E describes the connection-links among them. Each cafeteria node $i \in N$ serve a set of meals; thus, each node has a set of functions that define the Functionality Set (FS). Two main properties distinguish the small world network: (1) low average hop count between any two random chosen nodes, and (2) high clustering coefficient; therefore, our approach, in order to construct the FSW, categorizes the cafeteria nodes in the system into two types: 1) an in-domain node, and 2) a master node. The in-domain node represents a cafeteria node in which located in one cafeteria cluster and only has connections via short-links with all in-domain nodes placed in the same cluster and the master

node of that cluster. The master node represents a node located in one cafeteria cluster and has a connection via short-links with all in-domain nodes placed in the same cluster and at the same time has connection via long-links with some master nodes located in other clusters. Fig. 1 shows an example of the FSW, where nodes n_1, n_4 and n_6 are in-domain nodes, while nodes n_2, n_3 and n_5 are master nodes. The long-links (i.e. blue lines in Fig. 1) creates connections among master nodes and is responsible for achieving the high clustering coefficient in the network (property 2 in small world networks). Short-links (i.e. black lines) creates connection among in-domain nodes, and among master nodes and in-domain nodes. Short-links and the long-links aim at achieving the properties (1) and (2).

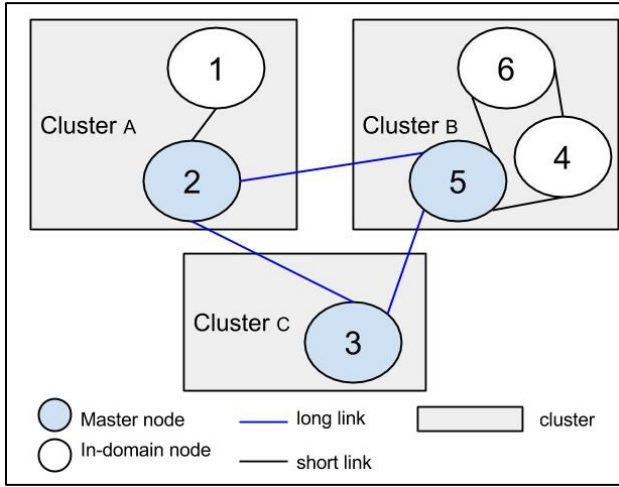


Figure 1: An Example of FSW overlay network, where white nodes present the in-domain cafeteria node, and the blue nodes represent the master nodes

In our design, we also define the cluster-size M to be the maximum number of nodes that are allowed to form one cafeteria cluster. Pre-defining the cluster size is important to keep small number of nodes in one cluster and to maintain good clustering effect. In this research, we adopt the guideline proposed by [17] to define M . Hui et al. suggested that the cluster size ranges from 1 to 64 maintains good clustering effect. Practically, designing a FSW overlay network plays an important role in decreasing the number of nodes that will exchange the workloads information, minimizing the network diameter, deteriorating the communication overhead, and decreasing the time delay results from the task re-migration process; therefore, this approach is efficient to be applied not only for the entire system but also clustering inside the cluster to increase the performance of the load-balancing algorithms.

In summary, the FSW of the cafeteria system can be formed as follows: Each cafeteria node maintains long-links to ensure the connectivity among master nodes (i.e. the connectivity among the clusters to provide shortcuts to allow a node reach other nodes that execute similar functionality and located in other clusters quickly) and/or short-links to ensure the connectivity among the in-domain nodes and the connectivity among the in-domain nodes and the master nodes so that a balancing message issued

from any node can reach any other node in the network. Via short-links and long-links, navigation and broadcasting in the network can be performed efficiently. In the following sections, we details the design of FSW.

3.2 Constructing Functional Small World (FSW) Overlay Network

Constructing a FSW overlay network for the cafeteria system depicted above involves three major tasks: 1) Functional-Clustering, 2) Cluster-Formation, and 3) Overlay Network Construction.

3.2.1 Functional-Clustering (FC). In general, the Functional-Clustering (FC) task aims at 1) defining the clusters (i.e. the number and the name of clusters) that should be created within the cafeteria overlay network, and 2) adding the nodes initially to the cluster(s) based on the in-common functions between the node and the defining cluster. In other words, if there is at least one function in-common between the node and the cluster, then the node will be added initially to that cluster. Note that: initially, in this step a node can be added to more than one cluster, but finally in the next tasks a node will only be added to one cluster.

This task is executed before or when a node joins the network. Each node n_i in the system defines its Functionality Set (FS), which indicates the functions that a node can perform and execute within the system, such as $FS_i = \{f_1, f_2, \dots, f_k\}$, where FS_i is the functionality set of node n_i , f_1 is a function that can be executed by node n_i , and k is the number of functions that node n_i can execute. In our manuscript, a cluster, namely, $Cluster_{i,j,\dots,k}$ has a functionality set $FS_{Cluster_{i,j,\dots,k}} = \{i, j, \dots, k\}$. Likewise, $Cluster_A$ has $FS = \{A\}$. Following are the steps performed by the functional-clustering task:

1. Let AF (All Functions) be the set of all functions executed in the system $AF = FS_1 \cup \dots \cup \{f_1, f_2, \dots, f_s\}$, where s is the total number of functions executed within the system, and FS_i is the functionality set of node n_i . In other words, AF is the union of all FSs defined in the system.
2. For each function $f \in AF$, create a cluster, namely, $cluster_f$.
3. Since each node n_i has its functionality set $FS_i = \{f_1, \dots, f_k\}$, in this step initially node n_i will be simultaneously added to $cluster_{f_1}, cluster_{f_2}, \dots, cluster_{f_k}$. In other nodes, if a node n_i executes a function f_a , then there is an in-common function between a node n_i and $cluster_a$. Thus, the node n_i will be added to $cluster_a$.

Note that, the number of clusters that a node can be added to depends on the number of functions that a node executes within the system; a node that executes more than one function will be added initially to more than one cluster at the end of this task.

3.2.2 *Cluster-Formation*. The Cluster-Formation (CF) task is a key task to ensure that a node will be added to only one cluster regarding the functional similarity. According to definition 1, nodes are considered as similar nodes if the amount of in-common functions among nodes is more than the amount of functions unique to nodes. The pseudo code of the cluster formation task is shown in Table 2.

Table 2. Pseudo code of the cluster formation task

Cluster-Formation task
<pre> Initialization Let A[] = { < cluster₁, cluster₁ >, < cluster₂, cluster₂ >, ..., < cluster_f, cluster_f > } where cluster₁ , cluster₂ , ..., cluster_f is the size of cluster₁, cluster₂, ..., cluster_f begin 1.int m[] = A.minArray(); // Finding the clusters that have the least cluster size 2.For each cluster "cluster_a" in A[] 3. For each node n_i added initially to cluster_a { 3.1. if FS_i = 1, then add n_i to cluster_a. // this means the functional similarity between a node // and the cluster is ≥ 0 since a node can execute one function and added to one cluster 3.2. if FS_i > 1, then // the node is initially added to more than one cluster //thus, these steps ensure positive similarity between a node and a cluster 3.2.1. if cluster_a ∈ m[] and m[] = 1 then add n_i to cluster_a. // m[] = 1 means the number of clusters that has the smallest cluster size is 1 3.2.2. elseif cluster_a ∈ m[] and m[] ≠ 1 then // here more than one cluster has the smallest cluster size 3.2.2.1. if n_i added to (one cluster cluster_a ∈ m[] and the other clusters not in m[]) then add n_i to cluster_a // this step ensures similarity and add node to cluster with smallest size 3.2.2.2. if n_i added to (more than one cluster_a ∈ m[]) then add a "wait" tag of n_i // this mean a node has in-common functions with two clusters in the same size, since // each cluster has different functionality, the similarity between a node and the cluster // may be negative; thus, additional steps must be done to ensure positive similarity 3.2.3. elseif cluster_a ∉ m[] then 3.2.3.1. check the FS = {f₁, ..., f_i, f_{id}} of n_i if it is a cluster ∈ m has the name cluster_{f_{id}} then n_i leave cluster_a otherwise add a tag "wait" to n_i } 4.For each node n_i tagged as wait 4.1. find TFS = FS₁ ∪ ∪ ∪ re z is the nodes z has a tag "wait" 4.2. create new cluster, namely, cluster_{TFS} ∪ ∪ ∪ 4.3. add n_i to cluster_{TFS} ∪ ∪ ∪ End </pre>

This task aims at: 1) deciding the nodes that must finally be added to the cluster, and 2) checking the cluster size; thus, if the cluster size exceeds M , which is a preset defined maximum cluster size, the cluster will be split into two clusters in order to maintain good clustering effect. To determine the cluster size, we adopt the guideline proposed by [17]. Hui et al. suggested that the maximum cluster size is 64 in order to maintain good clustering effect. If the cluster size exceeds M , the steps of the functional-clustering task, and the cluster-formation task will be applied to split that cluster (i.e. Note, new clusters with new names, such as $cluster_{A1}$ instead if $cluster_A$, will be created upon re-performing the tasks to split cluster(s)).

3.2.3 *Overlay Network Construction*. This task constructs the FSW overlay network for the cafeteria system (Fig.2 shows a view of dining area in the university campus) across the created clusters (i.e. after performing the previous two tasks) to form a functional small world network by:

1. Defining the in-domain nodes and the master nodes.

The size of the FS of each node located in one cluster will be checked (i.e. the number of functions that a node can execute); therefore, a node that has the largest FS size in cluster_i will be defined as a master node for cluster_i, and the other nodes located in cluster_i will be defined as the in-domain nodes for that cluster. Note, when two or more nodes have the largest FS size, then only one node from these nodes will be selected randomly as a master node for a cluster since that each cluster has only one master node.

2. Adding long-links and short-links among the nodes. Long-links connect a master node located in one cluster with other master nodes located in other clusters based on the functional similarity between these master nodes (i.e. see definition 1). Short-links connect the in-domain nodes located in one cluster with the other in-domain nodes located in the same cluster, and it also connects the in-domain nodes located in a cluster with the master node of the same cluster. In-domain nodes, master nodes, long-links and short-links play a key role in reducing the effect of the structural factors and transforming the network into a small world.



Figure 2: A view of dining areas in the university campus

4 Dynamic Load Balancing In Action

This section explains the proposed load-balancing algorithm that will be executed in the constructed FSW overlay network.

4.1 Problem Formulation

Generally, the entire cafeteria network is modeled as an undirected graph $G=(N,E)$ where N represents the set of heterogeneous cafeteria nodes and E describes the connections among them. Each cafeteria in the network (i.e. whether an in-domain node or a master node) will be assigned some orders or workloads w during the execution of the system, where each order assigned to a node consumes effort and time; thus, each workload has different weight w . The weight of the total

workloads assigned to a node is referred to as the load of a node $ld_i > 0$. Each assigned workload also is associated with a function that can process the assigned workload. Each node also has a capacity $c_i > 0$ which specifies its processing capacities (i.e. the largest amount of workload that can be assigned to a node n_i), where $c_i, ld_i \in \mathbb{Z}$. Since the capacity of each node in heterogeneous systems is not equal, our proposed algorithm considers the processing capacity of each node when deciding whether a node is overloaded or not.

Definition 2 (the effective-load). Given a cafeteria node $n_i \in N$ that has a capacity and assigned some orders, the effective-load l_i of that node n_i is defined as the total weight of the assigned orders divided by the cafeteria capacity. Formally, the effective-load of node n_i is the load of n_i divided by the capacity of n_i .

$$l_i = \frac{ld_i}{c_i} = \frac{\sum_{wl_j \in WL(n_i)} w(wl_j)}{c_i} \quad (1)$$

where $WL(n_i) = \{ \langle wl_1, w_1, ctr_{id}, F_{id} \rangle, \dots, \langle wl_z, w_z, ctr_{id}, F_{id} \rangle \}$ is the set of orders assigned to node n_i .

4.2 Our Proposed Algorithm

Our proposed algorithm is shown in Table 3: NeighborhoodLB. Each cafeteria node n_i in the system G executes the same algorithm in parallel. As mentioned before, based on the role of each node n_i within the system, n_i defines its functionality set (FS). Thus, the structure of the system is simplified by constructing the FSW to decrease the graph diameter, the number of nodes that exchange the load information and communication overhead. The steps of constructing FSW overlay network is illustrated in Section 3. The nodes will be spread into clusters, and each node will have in addition to the node id n_{id} , a cluster id ctr_{id} to show the cluster in which a node is located and FS_{id} to check if the received mi task can be processed by a node n_{id} . Following paragraphs demonstrate the proposed load-balancing algorithm that will be executed within the constructed overlay network in details.

4.2.1 The Initialization Stage. Let $WL(n_i)$ be the set of assigned orders, $WL(n_i) = \{ \langle wl_1, w_1, ctr_{id}, F_{id} \rangle, \dots, \langle wl_z, w_z, ctr_{id}, F_{id} \rangle \}$. Each assigned workload or order wl consumes time and efforts until being completed; thus, each assigned workload has weight w . Each workload wl assigned initially to ctr_{id} and associated with a function F (i.e. F is the function that can process the workload). Each node n_i also has, after constructing FSW, a pre-defined set of *neighbor-nodes* $Adj(n_i)$ to store the nodes that have connection-links either *long-links* or *short-links* with node n_i .

Each node n_i initializes its state (initialization stage) in steps 1 through step 3.

Step 1 (Line 1 in NeighborhoodLB Algorithm): Each node n_i defines a set $Info = \{ \langle ctr_{id}, n_{id}, ld_{id}, c_{id}, FS_{id} \rangle \}$ to store the information of the nodes in the neighbor-nodes set, where ctr_{id} is the id of the cluster in which a node the has n_{id} is located, n_{id} is the id of a node, $ld_{id} = \sum_{wl_j \in WL(n_{id})} w(wl_j)$ the load of node n_{id} (i.e. the total weight of all workloads assigned to the node n_{id}), c_{id} is the processing capacity of n_{id} , and FS_{id} is the functional set of n_{id} .

Step 2 (Line 2 in NeighborhoodLB Algorithm): Each node n_i also defines an array $mig(n_i)$ to store the amount of the migrated workload that node n_i will transfer to the under loaded nodes of the set neighbor-nodes. Initially, the workloads that will be transferred to other nodes is 0 for all nodes in the set of neighbor-nodes.

Step 3 (Line 3 in NeighborhoodLB Algorithm): Each node n_i computes its initial effective-load l_i via the equation defined in definition 2 (i.e. the total weight of the workloads assigned to node n_i divided by the capacity of node n_i). Each node in the system executes the same proposed algorithm in parallel. In the initialization stage, each node: (1) defines $Info$ set to store the information about its neighbor-nodes, (2) defines mig array to store the amount of excess workload to be transferred, and (3) computes its effective-load.

4.2.2 The information Broadcasting Stage. Step 4 (Line 4 in NeighborhoodLB Algorithm): Each node n_i broadcasts its initial state (i.e. initial information after executing the initialization stage) to only its *neighbor-nodes* (the nodes stored in the set adj). Since a *master node* has connections with some *master nodes* located in other clusters that have similar functionality via *long-links*, and it has also connections with the *in-domain nodes* located in the same cluster via *short-links*, the capacity of a *master node* that will be sent to other nodes is divided among the clusters $\frac{c_i}{|long-links|+1}$ in the broadcasting stage. In fact, each node maintains a FIFO message queue which holds the incoming messages. Each message has the format $\langle ctr_{id}, n_f, ld_f, c_f, FS_f, "T", [g, "F"] \rangle$, where ctr_{id} is the cluster id where the node that sends the message is located in, n_f is the id of the sender node, ld_f the loads of the sender node, c_f is the capacity of the sender node, FS_f is the functionality set of the sender node, T is the type of the message, g is the migration information (i.e. information about the migrated task and the function F that can process the migrated task). There are two types of messages:

Table 3: NeighborhoodLB

<p><i>Algorithm1.NeighborhoodLB</i></p> <p>n_{id} : The node where the algorithm is executed. ctr_{id} : The id of a cluster in which n_i is located c_i : The processing capacity of node n_i $Adj(n_i) = \{< ctr_{id}, n_{id} >\}$ The set of neighbor-nodes $WL(n_i) = \{< wl_{id}, w, ctr_{id}, F_{id} >\}$: The set of assigned workloads for n_i FS_{n_i} : the functionality set of n_i</p>
<p>Begin</p> <p>1. Let $Info_i = \{< ctr_{id}, n_{id}, ld_{id}, c_{id}, FS_i >\}$ 2. Let $mig(n_i) = 0$ for all $n_j \in Adj(n_i)$ 3. Compute the effective-load: $l_i = \frac{ld_i}{c_i} = \frac{\sum_{w_j \in WL(n_i)} w(w_j)}{c_i}$ 4. For each node $n_j \in Adj(n_i)$ do a. if n_i is master node then send message $\langle ctr_{id}, n_i, ld_i, \frac{c_i}{ long_links +1}, FS_i, "B", [0, ""] \rangle$ b. else send message $\langle ctr_{id}, n_i, ld_i, c_i, FS_i, "B", [0, ""] \rangle$ 5. Read messages from the messages queue a. if T="B" then $Info = Info \cup \langle n_f, ld_f, c_f, FS_f \rangle$ b. if T="G" then 1) $Info = Info \cup \langle n_i, ld_i + g, c_i, FS_i \rangle, \langle ctr_{id}, n_f, ld_f - g, c_f, FS_j \rangle$ 2) $l_i = \frac{ld_i + g}{c_i}$ 3) For each node $n_j \in Adj(n_i)$ do a. if n_j is master node then send message $\langle ctr_{id}, n_i, ld_i + g, \frac{c_i}{ long_links +1}, FS_i, "B", [0, ""] \rangle$ b. else send message $\langle ctr_{id}, n_i, ld_i + g, c_i, FS_i, "B", [0, ""] \rangle$ 6. Compute the average effective-load $l_{avg} = \frac{ld_i + \sum_{j \in Info} ld_j}{c_i + \sum_{j \in Info} c_j}$ 7. For each node $n_j \in Adj(n_i)$ do //Define the Assistant Neighbors a. if $\frac{ld_j}{c_j} < l_{avg}$ and $\frac{ld_j}{c_j} \leq l_i$ then $N_{lower} = N_{lower} \cup$ 8. Let load-difference $LD_i = (l_i - l_{avg})$ 9. If $LD \leq 0$ then exit; else $LB(WL(n_i), N_{lower}, LD_i)$ EndBegin</p>

1. Workload Migration message ("G"): n_i sends a "G"-message to n_j to tell it that n_i wants to migrate g units of workload to n_j .
2. Broadcast message ("B"): broadcast the status (i.e. cluster id, node id, load and capacity to all neighbor-nodes).

Step 5 (Line 5 in NeighborhoodLB Algorithm): The main part of the algorithm starts when a node takes the first message from the queue and processes the message according to its type. If the message type is "B", then the node only updates its information stored in the *Info* set. If the message type is "G", then it updates the information stored in the *Info* set, computes its effective load, and broadcasts its new status to its *neighbor-nodes*. Initially, the first message received by each node is "B" type messages.

4.2.3 Computing the average effective-load. Step 6 (Line 6 in NeighborhoodLB Algorithm): After updating the information stored in the *Info* set (i.e. after the broadcasting stage), each node computes the average effective-load l_{avg} of a node and its neighbor-nodes to facilitate 1) making a decision (i.e. whether a node overloaded or not) later by a node, and 2) defining the set of assistant neighbors in the next stage. The average effective-load is computed by the following equation:

$$l_{avg} = \frac{ld_i + \sum_{j \in info} ld_j}{c_i + \sum_{j \in info} c_j} \quad (2)$$

Note that, in the above formula the capacity of all nodes is considered since in heterogeneous systems the capacity is varied from one node to another.

4.2.4 Finding the set of assistant-neighbors Stage. Step 7 (Line 7 in NeighborhoodLB Algorithm): According to the average effective-load computed in step 6 by each node, each node defines in this stage its assistant-neighbors N_{lower} . The set of assistant-neighbors N_{lower} of node n_i are the set of nodes that have effective-load lower than the average effective-load computed by node n_i .

4.2.5 Workload transfer strategy. Step 8 (Line 8 in NeighborhoodLB Algorithm): The decision of calling a procedure LB to migrate the excess workloads or not depends on the load difference between the current *effective-load* of node n_i and the *average effective-load* computed by n_i . Therefore, the excess workload will be migrated if the load difference is positive.

4.2.6 Load-balancing mechanism (Procedure LB). The pseudo-code of the procedure LB is given in Table 4. In the procedure LB, the load difference LD_i , the set of *assistant-neighbors* N_{lower} , and the set of the assigned workloads $WL(n_i)$ are formed the procedure input parameters. The procedure will be called if the LD_i is positive, and it works until the load difference of the heavily loaded caller node n_i becomes less than zero $LD_i = l_i - l_{avg} < 0$. In other words, the procedure works until the heavily loaded node becomes under-loaded, which means the *effective-load* of a node is less than the *average effective-load* computed by a node. The procedure first computes the excess workload δ_i of the heavily-loaded node n_i that needs to be transferred.

Table 4: Procedure LB

Procedure LB($WL(n_i), LD_i, N_{lower}$)
Begin
While($LD_i > 0$)
1. Compute the excess workload of n_i : $\delta_i = LD_i \times c_i$
2. sort the submitted workloads in ascending order
3. sort the assistant neighbours in descending order
4. Let $j=0$
5. For a node n_j in N_{lower}
a. compute the excess workload n_j can receive $\alpha = (l_{avg} - l_j) \times c_j$
b. If $w(wl_k) \leq \alpha$ and F is in FS_{n_j} then
1) $k = k + 1$
2) send message to node $n_j < n_i, l_i, c_i, FS_i, "G", [\alpha, F] >$
else
1) go to step 5
End For
End While
End Begin

Then, it sorts: 1) the set of assistant-neighbors N_{lower} in descending order based on their effective-loads, and 2) the set of submitted workloads $WL(n_i)$ in ascending order in accordance with the weight of each submitted workload. The procedure also checks each node in the set N_{lower} and computes how much a node can receive α (i.e. the workload that a node can receive is equal to the difference between the effective-load of a node and the average effective-load). The procedure migrates only the workload that has weight less than or equal to α . This step plays a key role in redistributing the excess workloads to the assistant-neighbors in a way that ensures that the node who receives the workload maintains the under-loaded status. The LB procedure is terminated when the load difference of the caller heavily-loaded node becomes negative. In other words, the procedure is terminated when the node becomes under-loaded.

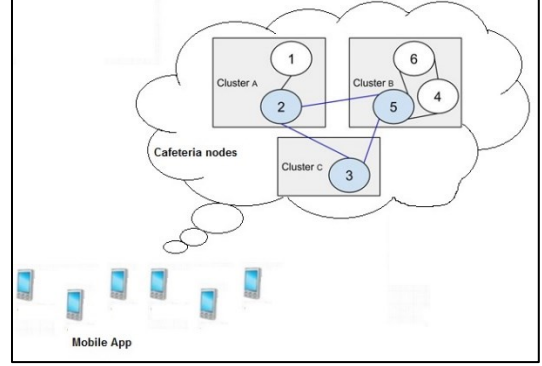
5 Simulations

5.1 Experimental Setting

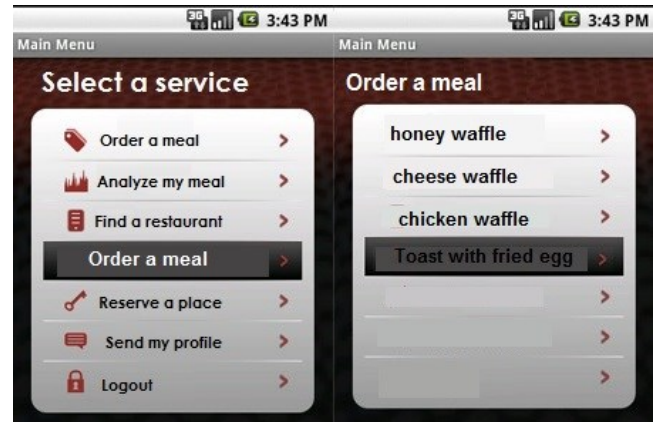
In order to perform our test, we build a system based on the server-client architecture for a cafeteria system in a university campus (Fig. 3). The client side is a mobile application that allows students to order meals from any cafeteria inside the campus (Fig. 4), while the server side represents the cafeteria nodes that run the load-balance algorithm in parallel. When a student order a meal, initially the order will be assigned to a cafeteria node that has that meal in its functionality set as well as has the lowest assigned workloads. When the order assigned to a cafeteria node, the name of the cafeteria will appeared to the student on the mobile application to pick his/her order without the need to wait in a queue.

To simulate the students orders, we have implemented a discrete-event simulator using the SimJava [12]. The simulation

is used to compare the performance of our proposed approach with two of the most popular dynamic diffusion approaches, the nearest neighbor algorithm [24] and the original neighborhood algorithm [22]. The three approaches were run on a set of default values: number of assigned workloads, number of nodes, maximum cluster size, and the average number of the functions executed per node. The simulation parameters, and their values are given in Table 5.

**Figure 3: The system architecture**

For fairness of comparison, we have tested the three approaches on random graphs (random scenario) generated via random generator [23]. In the random scenario, the generator will randomly distribute nodes with a functional set associated with each node in the graph. As shown in table 5, maximum number of functions that each node can execute is 20. Since, in this research, we propose a two-stage approach (creating a functional small world overall network and then run the NeighborhoodLB on the created FSW) to improve the performance of load-balance algorithm, the random graph, generated previously, will be converted to FSW before executing our proposed NeighborhoodLB algorithm. On the other hand, the other two algorithms, the nearest neighbor algorithm and the original neighborhood algorithm were executed directly on the generated random graph since they do not employ the first stage of creating FSW.

**Figure 4: The client side****Table 5. Parameters used in the simulations**

	Description	Values
1.	The assigned Workloads	1,000-10,000
2.	The number of nodes in the system	100-1,000
3.	The cluster size	1-64
4.	The number of functions in the FS per node	1-20

The comparison tests were based on two parameters: the assigned-workloads and the number of nodes, and the measurement of the performance of the algorithm was based on four metrics: the throughput, the response time or the completion time, the communication overhead, and the movement cost. The response time measures the total time that the system takes to serve a submitted request (task). In this experiment, to simulate real world distributes systems, we randomly submitted tasks to nodes. Initially, the request state will be “submitted to node” and will be changed to “complete” upon serving that request. To measure the response time, we count the time needs to change the node response time from “submitted to node” to “complete”. The throughput is the rate at which a node in the system sends or receives data (i.e. throughput = 1/ response time). In other words, the throughput is defined as the number of nodes that change its status to “complete” in a time unit. As we can see from the proposed algorithm, the load balancing algorithm needs to migrate request from one node to another one in order to achieve a balanced state. We use a simple linear cost model [11], where moving one request from any node to any other node costs one unit. Such a model reasonably captures both the network communication cost of transferring data, as well as the cost of modifying local data structure at the node.

Only one parameter was changed each time so that any changes in the performance would be based solely on this parameter. In fact, results achieved from these tests were used to study: (1) the behavior of the different load-balancing algorithms under the same condition; (2) the behavior of the algorithms for random systems with different number of nodes; (3) the behavior of the algorithms for different workloads distribution.

To study the effects of changing the assigned workloads on the average response time, the throughput, the communication overhead, and the movements cost, the assigned workloads were varied from 1000-10,000 workloads unit, and the workloads distribution among the nodes were carried in the following manner.

- The initial workload distributions varying 25% from the average effective-load to represent a situation where all nodes have similar workloads at the beginning and those workloads are close to the average effective-load; in other words, the initial situation is quite balanced.
- The initial workload distributions varying 50% from the average effective-load to constitute the intermediate situations.
- The initial workload distributions varying 75% from the average effective-load to constitute the advanced intermediate situations.
- The initial workload distributions varying 100% from the average effective-load to form the situation where the difference of workloads between nodes at the beginning is considerable.

To study the effects of changing the number of nodes on the average response time, the throughput, the communication overhead, and the movements cost, the number of nodes were varied from 100 – 1000 nodes and the distribution of the overloaded nodes were carried in the following manner.

- 25% of nodes are idle, 75% of nodes are overloaded.
- 50% of nodes are idle, 50% of nodes are overloaded.
- 75% of nodes are idle, 25% of nodes are overloaded.

5.2 Comparative Study

5.2.1 Average Response Time. The total time taken for the three algorithms, our proposed algorithm, the original neighborhood algorithm, and the nearest neighbor algorithm, to complete the assigned workloads increased as the assigned workloads was increased as shown in Fig. 5.

This situation is expected as the more workloads to be assigned, the longer it takes to complete all the assigned workloads. However, it was observed that our proposed method (i.e. the green line) performed better than both the nearest neighbor scheme and the original neighborhood algorithm in all cases. We can see that when comparing the results of our proposed method and the original neighborhood algorithm (i.e. the red line) and the nearest neighbor algorithm (i.e. the blue line), it is observed that the gap between these three curves was widening as the assigned workloads was increased. This shows that the method actually reduced the response time or the total completion time by a considerable amount (greater speedup) in comparison to the original neighborhood algorithm and the nearest neighbor algorithm as amount of workloads increased.

Fig. 6 shows that the response time of the proposed method (i.e. green line) slightly increased when the number of nodes was increased. In contrast, the response time of the original neighborhood method (i.e. red line) and the nearest neighbor method (i.e. blue line) sharply increased when the number of nodes was increased.

The reasons behind achieving better results (i.e. achieving better response time when increasing the assigned workloads or when increasing the number of nodes): 1) our proposed approach constructs a FSW overlay network and then executes the proposed neighborhood load-balancing within the constructed network. Specifically, constructing the overlay network reduces the number of nodes that exchange the workload information, decreases the network diameter, and the communication overhead. As a result, all the stages of the proposed algorithm, such as updating the information of the neighbor nodes, calculating the average effective-load, choosing the assistant neighbors, and migrating tasks to the assistant neighbor that can process the task, will be performed in less time. Our approach also plays a significant role in reducing the time delay results from the task re-migration process as nodes with similar functionality can communicate with each other. As illustrated before, re-migrating tasks occur because of out of the node service scope situation; 2) our proposed approach utilizes the on-state information exchange strategy to broadcast its information to only its neighbor-nodes, which has the advantages of achieving more accurate calculation to the effective-load and the average effective-load without increasing the communication overhead (i.e. each node collects

the information from less nodes, only neighbor nodes, as compared with the original neighborhood approach and the nearest neighbor approach); 3) our approach utilizes the concepts of assistant-neighbors and thus heavily loaded nodes will send only (i.e. without accepting any workloads from other nodes since the node is currently overloaded) the excess workloads to the lightly loaded nodes “assistant-neighbors”, whereas the lightly loaded nodes will only receive the migrated workloads without sending any workloads.

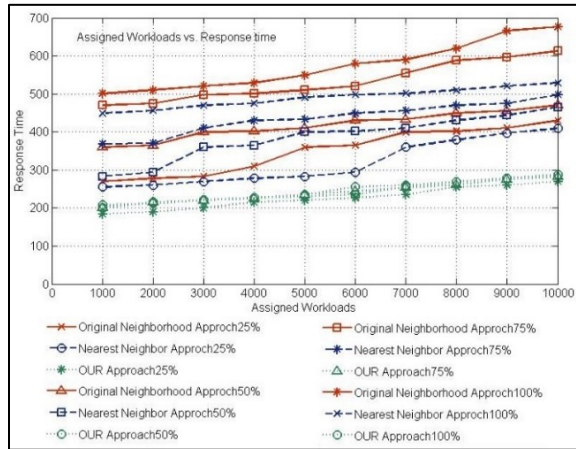


Figure 5: The response time of original neighborhood approach, nearest neighbor approach, and our approach for various assigned workloads

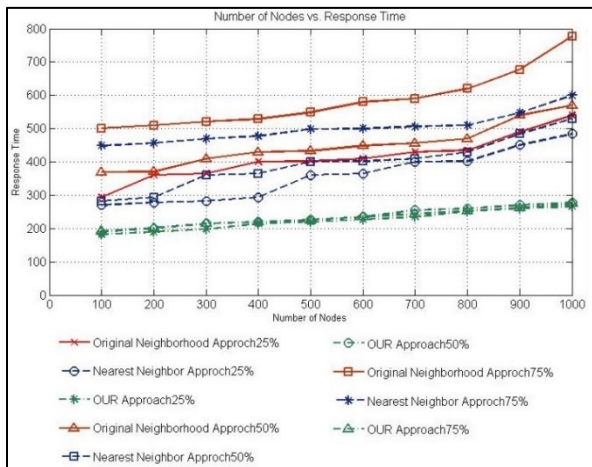


Figure 6: The response time of original neighborhood approach, nearest neighbor approach, and our approach for various number of nodes

In contrast, in the original neighborhood approach and the nearest neighbor approach, all nodes will send and receive workloads at the same time which in turn increase the communication overhead and thus increasing the response time; 4) it is intuitive that a system with longer diameter will take longer time to converge as the number of iterations to propagate the workloads to lightly loaded nodes is proportional to the network diameter; thus, reducing the network diameter via constructing FSW plays

a key role in reducing the response time of our proposed approach. In contrast, in the original neighborhood approach and the nearest neighbor approach, each node has to collect the workloads information from larger number of nodes which in turn leads to increase response time.

5.2.2 Throughput. As shown in Fig. 7, our method outperformed the original neighborhood algorithm and the nearest neighbor method in terms of the system throughput in all assigned workloads distribution cases.

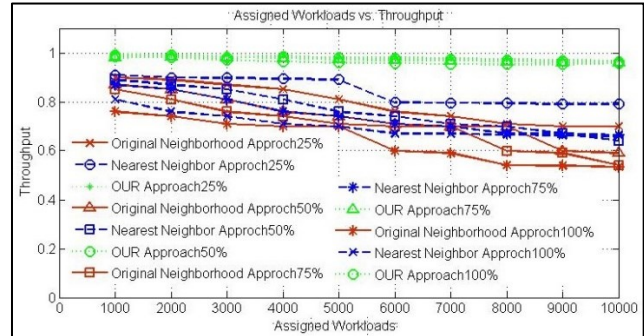


Figure 7: The throughput of original neighborhood approach, nearest neighbor approach, and our approach for various assigned workloads

We can notice that the throughput of the system that executes our proposed approach steadily increased even the assigned workloads increased, whereas the throughput of the system that execute the original neighborhood approach or the nearest neighbor approach drops quickly when the assigned workloads increased.

Fig. 8 shows that the throughput achieved by the original neighborhood algorithm as well as the nearest neighbor approach decreased sharply as the number of nodes in the system increased, while the throughput achieved by our proposed method remains stable even when increasing the number of nodes.

This is because our proposed approach reduces the task completion time which in turn increases the number of tasks completed in a time unit. The reasons behind this are: 1) constructing the FSW that allows nodes with similar functionality to communicate with each other, reduces the possibility of re-migrating tasks (re-migrating tasks consumes time); 2) checking the function that can process the task with the FS before migrating the task, eliminate the possibility of re-migrating tasks. Note that, the first point reduces the time of performing the second point; thus, better results are achieved; 3) reducing the number of nodes that exchange the workload information, decreasing the network diameter, and decreasing the communication overhead reduces the time of performing the proposed algorithm, such as updating the information of the neighbor nodes, calculating the average effective-load, choosing the assistant neighbors, and migrating tasks to the assistant neighbor. As a results, the number of tasks completed in a time unit will be increased; 3) utilizing the concepts of *assistant-neighbors* allowing only heavily loaded nodes to send only (i.e. without accepting any workloads from other nodes since the node is currently overloaded) the excess workloads to the lightly loaded nodes “*assistant-neighbors*”. Also, the lightly loaded

nodes will only receive the migrated workloads without sending any workloads. In contrast, in the original neighborhood approach and the nearest neighbor approach, all nodes will send and receive workloads at the same time which in turn increase the communication overhead and thus decreasing the task completion time. Moreover, the importance of the *average effective-load* also appears when deciding the amount of workloads to be migrated; if the migrated workloads to one node is too small, then the workload distribution will take longer (i.e. which in turn decreasing the system throughput). In contrast, if the migrated workloads to one node is too large, then the overloaded node may transfer too much workloads to its *neighbor-nodes* and thus this overloaded node will not have sufficient workload to transfer to the remaining lightly loaded nodes. Therefore, by using the *average effective-load*, each node obtains an amount of workload proportional to its capacity and thus no node is privileged which results in increasing the system throughput (i.e. the number of workloads completed in unit time).

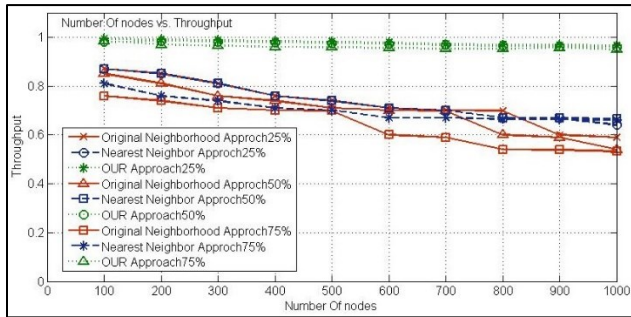


Figure 8: The throughput of original neighborhood approach, nearest neighbor approach, and our approach for various number of nodes

5.2.3 Communication overhead. Fig. 9 shows that the average number of messages sent per node increased when the assigned workloads increased. This is because when the assigned workloads increased, the number of messages sent per a node to broadcast its new status increased. We can see that our proposed approach produces less communication overhead than both the original neighborhood approach and the nearest neighbor approach even when increasing the assigned workloads.

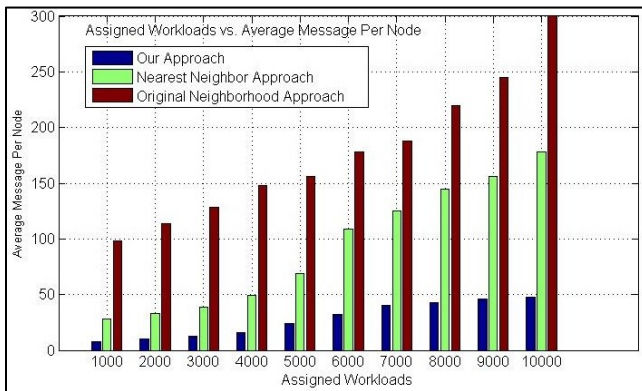


Figure 9: The average number of messages sent per node of original neighborhood approach, nearest neighbor approach, and our approach for various assigned workloads

Fig. 10 shows that the average number of messages sent per node increased when the number of nodes increased. This is because when the number of nodes increased, each node will send more messages to broadcast its information to the other nodes. We can see that our proposed approach produces less communication overhead than the both the original neighborhood approach and the nearest neighbor approach because: 1) constructing a FSW decreases the number of nodes that exchange the workloads information which in turn decreases the communication overhead; 2) constructing a FSW also decreases the network diameter which directly has the impact of decreasing the communication overhead; 3) each node that executes the proposed NeighborhoodLB algorithm sends/receives messages to/from only its neighbor nodes which plays a key role in reducing the communication overhead; 4) our approach utilizes the on-state information exchange strategy which reduces the communication overhead; 5) our approach (constructing the FSW, and the proposed load-balancing algorithm) eliminates the possibility of re-migrating tasks which in turn decreases the communication overhead.

5.2.4 Movement cost. Fig. 11 shows the movement cost of original neighborhood approach, the nearest neighbor approach, and our proposed approach vs. the assigned workloads, where the movements cost is defined as the total migrated workloads divided by the total assigned workloads in the system. Clearly, the movements cost of our proposed approach is only 0.32 times the cost of the original neighborhood approach, while the movements cost of our proposed approach is only 0.34 times the cost of the nearest neighbor approach.

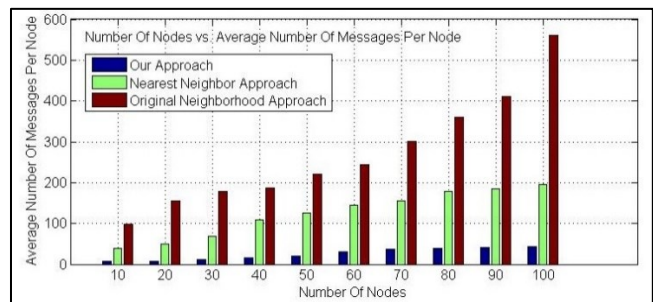


Figure 10: The average number of messages sent per node of original neighborhood approach, nearest neighbor approach, and our approach for various number of nodes

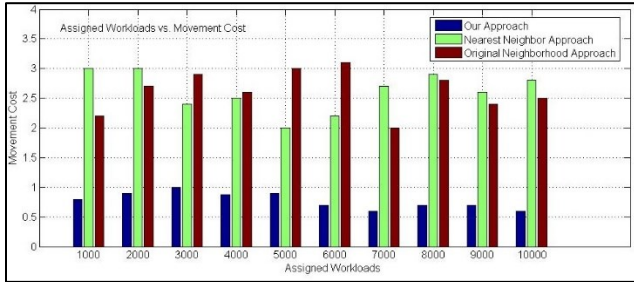


Figure 11: The movements cost of original neighborhood approach, nearest neighbor approach, and our approach for various assigned workloads

Fig. 12 shows the movement cost of original neighborhood approach, the nearest neighbor approach, and our proposed approach. We can see that the movements cost of our proposed approach is only 0.33 times the cost of the original neighborhood approach, while the movements cost of our proposed approach is only 0.30 times the cost of the nearest neighbor approach. This is because each node in our proposed algorithm calculates the average effective-load to decide whether a node itself is overloaded or not. Specifically, the importance of the average effective-load appears when deciding the amount of workloads to be migrated; if the migrated workloads to one node is too small, then the number of workloads that will be migrated will be high (i.e. which in turn increasing the movement costs). In contrast, if the migrated workloads to one node is too large, then the overloaded node may transfer too much workloads to one neighbor node and thus increasing the movements cost. Therefore, by using the average effective-load, each node obtains an amount of workload proportional to its capacity and no node is privileged which leads in decreasing the movements cost. Moreover, our approach utilizes the concepts of assistant-neighbors and thus heavily loaded nodes will send only (i.e. without accepting any workloads from other nodes since the node is currently overloaded) the excess workloads to the lightly loaded nodes "assistant-neighbors", whereas the lightly loaded nodes will only receive the migrated workloads without sending any workloads. In contrast, in the original neighborhood approach and the nearest neighbor approach, all nodes will send and receive workloads at the same time which in turn increase the number of workloads that will be migrated and thus increasing the movements cost. Finally, our approach (constructing the FSW, and the proposed load-balancing algorithm) eliminates the possibility of re-migrating tasks which in turn decreases the movements cost.

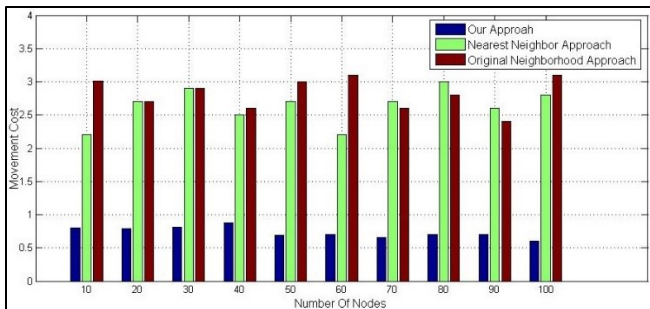


Figure 12: The movements cost of original neighborhood approach, nearest neighbor approach, and our approach for various number of nodes

A novel load-balancing approach to deal with load rebalancing problem in large scale, dynamic and heterogeneous systems has been presented in this paper. Previous research concluded that the technical, and the structural load-balancing factors: (1) increasing the number of nodes in the system (i.e. the number of the nodes exchange the workload information); (2) increasing the network diameter which is defined as the longest shortest path between any two nodes of the network; (3) increasing the communication overheads or the communication delays among the nodes decrease the performance of any load-balancing algorithm as well as affect the algorithm convergence rate. Moreover, additional delay may occur because of the task re-migration process. Therefore, we propose a two-stage approach that first constructs the FSW based on the properties of the small world network and the functionality of each node. Constructing the FSW reduces the number of nodes that exchange the workloads information in the system, decreases the diameter of the network and reduces the communication overhead, and decreases the delay resulted from re-migrating tasks. We also proposes a load-balancing algorithm that considers the capacity of each node in order to execute the algorithm within the constructed FSW overlay network. Our proposed approach strives to balance the loads of nodes, increase the system throughput, decrease the response time, reduce the communication overhead, deteriorate the demanded movements cost as much as possible, while taking the advantages of the nodes functionality and the nodes heterogeneity. In the absence of representative real workloads, we have investigated the performance of our proposed approach and compared it against competing algorithms, i.e. the original neighborhood approach, and the nearest neighbor approach. The simulation results are encouraging, indicating that our proposed algorithm performs very well. Our proposed approach dramatically outperforms the original neighborhood approach, and the nearest neighbor approach in terms of response time, throughput, communication overhead, and movements cost. Finally, we have proved that the proposed approach converges to the state of fairness where the effective-load in all nodes is the same since each node receives an amount of workload proportional to its processing capacity. Therefore, we conclude that this approach has the advantage of being fair, simple and no node is privileged.

6 Conclusion

An improved load-balancing approach applied to a cafeteria system in a university campus has been presented in this paper. Our proposed approach considers, first, the structure of the network that will execute the algorithm by constructing a Functionality Small World (FSW) overlay network to reduce the number of nodes that exchange its workloads information in the system, the diameter of the network and the communication overhead. It also considers the dynamic load-balancing algorithm parameters that will be executed within the constructed overlay

cafeteria network to achieve better performance. We have evaluated the performance of our proposed approach and compared it against competing algorithms, i.e. the original neighborhood approach, and the nearest neighbor approach. Results are encouraging, indicating that our proposed algorithm dramatically outperforms the original neighborhood approach, and the nearest neighbor approach in terms of response time, throughput, communication overhead, and movements cost.

REFERENCES

1. Aakanksha and Punam Bedi. 2007. Load Balancing on Dynamic Network Using Mobile Process Groups. In *15th International Conference on Advanced Computing and Communications*, 553–558.
2. Abdelzahir Abdelmaboud, Dayang N.A. Jawawi, Imran Ghani, Abubakar Elsafi, and Barbara Kitchenham. 2014. Quality of Service Approaches in Cloud Computing: A Systematic Mapping Study. *Journal of Systems and Software* 101: 159–179.
3. Clemens P J Adolphs, Petra Berenbrink, and V A Canada. 2012. Distributed Selfish Load Balancing with Weights and Speeds Categories and Subject Descriptors. In *ACM symposium on principles of distributed computing*, 135–144.
4. Hoda Akbari, Petra Berenbrink, and Thomas Sauerwald. 2012. A Simple Approach for Adapting Continuous Load Balancing Processes to Discrete Settings. In *PODC*, 271–279.
5. Jacques M Bahi, Flavien Vernier, and Belfort Cedex. 2007. Synchronous Distributed Load Balancing on Totally Dynamic Networks. In *IEEE International Parallel and Distributed Processing Symposium*, 1–8.
6. J E Boillat. 1990. Load balancing and Poisson equation in a graph. *Concurrency Practice and Experience* 2, 4: 289–313.
7. Hsien-Tsung Chang, Yi-Min Chang, and Sheng-Yuan Hsiao. 2014. Scalable network file systems with load balancing and fault tolerance for web services. *Journal of Systems and Software* 93: 102–109. Retrieved January 5, 2015 from <http://www.sciencedirect.com/science/article/pii/S0164121214000685>
8. Hoon Sung Chwa, Hyoungbu Back, Jinkyu Lee, Kieu-My Phan, and Insik Shin. 2015. Capturing urgency and parallelism using quasi-deadlines for real-time multiprocessor scheduling. *Journal of Systems and Software* 101: 15–29.
9. George Cybenko. 1989. Dynamic Load Balancing for Distributed Memory Multiprocessors. *Journal of parallel and Distributed Computing* 7, 2: 279–301.
10. Robert Elsässer, Burkhard Monien, Stefan Schamberger, and Gunther Rote. 2002. Toward optimal diffusion matrices \mathcal{L} . In *International Parallel and Distributed Processing Symposium*, 1530–2075.
11. Prasanna Ganeasan, Mayank Bawa, and Hector Garcia-Molina. 2004. Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems. In *30th Annual International Conference on Very Large Data Bases*, 444–455.
12. Fred Howell and Ross Mcnab. 1998. SimJava: A Discrete Event Simulation Library For Java. In *International conference on Web-Based Modeling and Simulation*, 51–56.
13. Y.F. Hu and R.J. Blake. 1999. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing* 25, 4: 417–444.
14. Chi-chung Hui and Samuel T Chanson. 1996. A Hydro-Dynamic Approach To Heterogeneous Dynamic Load Balancing. In *International Conference on Parallel Processing*, 140–147.
15. Chi-chung Hui and Samuel T Chanson. 1999. Hydrodynamic Load Balancing. *IEEE Transactions on Parallel and Distributed Systems* 10, 11: 1118–1137.
16. Chi-Chung Hui and Samuel T. Chanson. 1997. Theoretical Analysis of the Heterogeneous Dynamic Load-Balancing Problem Using a Hydrodynamic Approach. *Journal of Parallel and Distributed Computing* 43, 2: 139–146.
17. Ken Y.K. Hui, John C.S. Lui, and David K.Y. Yau. 2006. Small-world overlay P2P networks: Construction, management and handling of dynamic flash crowds. *Computer Networks* 50, 15: 2727–2746.
18. David R Karger and Matthias Ruhl. 2004. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *16th annual ACM symposium on Parallelism in algorithms and architectures*, 36–43.
19. Yifeng Luo, Shuigeng Zhou, and Jihong Guan. 2014. LAYER: A Cost-Efficient Mechanism to Support Multi-Tenant Database as a Service in Cloud. *Journal of Systems and Software* 101: 86–96.
20. E Luque, A Ripol, A Cortes, and T Margalef. 1995. A Distributed Diffusion Method for Dynamic Load Balancing on Parallel Computers. In *the Euromicro Workshop on Parallel and Distributed Processing*, 43–50.
21. Henning Meyerhenke. 2009. Dynamic Load Balancing for Parallel Numerical Simulations Based on Repartitioning with Disturbed Diffusion. In *15th International Conference on Parallel and Distributed Systems*, 150–157.
22. P Neelakantan. 2012. Decentralized Load Balancing In Heterogeneous Systems Using Diffusion Approach. *International journal of Distributed and Parallel systems* 3, 1: 229–239.
23. Tiago P. Peixoto. 2014. The graph-tool python library. *figshare*. Retrieved from <https://graph-tool.skewed.de/>
24. Harumasa Tada. 2011. Nearest Neighbor Task Allocation for Large-Scale Distributed Systems. In *10th International Symposium on Autonomous Decentralized Systems*, 227–232.
25. Amos Tversky. 1977. Features of similarity. *Psychological Review* 84, 4: 327–352.
26. Belabbas Yagoubi and Meriem Meddeber. 2010. Distributed Load Balancing Model for Grid Computing. *ARIMA Journal* 12: 43–60.
27. Albert Y Zomaya, Senior Member, and Yee-hwei Teh. 2001. Observations on Using Genetic Algorithms for Dynamic Load-Balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 9: 899–911.