Bachelor's Thesis

Design, Simulation, and Evaluation of a Load Balancing Algorithm for Peer-to-Peer-Networks based on Push-Pull Sum and Deal-Agreement-Based Algorithms

Emre Bayazıtoğlu

Examiner: Prof. Dr. Christian Schindelhauer
Advisers: Saptadi Nugroho

University of Freiburg
Faculty of Engineering
Department of Computer Science
Chair of Computer Networks and Telematics

January 25th, 2025

Writing Period

18. 12. 2024 – 01. 03. 2025

Examiner

Prof. Dr. Christian Schindelhauer

Advisers

Saptadi Nugroho

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

Abstract

The Push-Pull Sum protocol, as introduced in [1], combines the Push-Sum [2] and Pull-Sum protocols. The Push-Sum protocol, proposed by Kempe et al., is a load balancing algorithm where each node randomly selects a neighbor and transfers half of its current sum and weight to that neighbor. Like the Push-Sum, the Push-Pull Sum protocol is also a randomized load balancing algorithm. These protocols are used to balance loads in peer-to-peer networks, modeled as undirected graphs. In these networks, nodes exchange loads with their neighbors in order to reach a balanced network state. The Single-Proposal Deal-Agreement-Based load balancing algorithm as proposed in [3], incorporates a Deal-Agreement into the load transfer to achieve only fair load transfers between two nodes.

In this thesis, we introduce and implement a variation of the Push-Pull Sum protocol using principles of the Deal-Agreement-Based protocol and adaptive thresholding, designed to add or modify certain properties of the original Push-Pull Sum protocol. For the newly introduced load balancing approach, we provide the pseudocode, implement the algorithm in a peer-to-peer simulation tool, and analyze the simulation outcomes for different topologies. The aim is to find a compromis solution including overall good performance in different topologies.

The performance of this variations is evaluated using the mean squared error (MSE) reduction over time as a metric for convergence to the ground truth. The results are presented through log-log and log-linear graphs, which allow a comparison of the convergence rates and stability in different scenarios. The slope of the MSE

curves reflects how efficiently the protocols distribute the load across the network. In addition to evaluating the efficiency, this thesis also analyzes potential drawbacks of each variation, such as increased communication overhead or slower convergence in certain scenarios. The data is fitted into different models to get an insight into the rate of convergence.

The findings suggest that while the modification provide improvements in specific aspects—such as faster convergence in some scenarios—others highlight trade-offs between performance and reliability. **(TODO: Some more information about the outcomes.)**

Contents

List of Figures

List of Tables

List of Algorithms

Listings

1 Introduction

Peer-to-peer (P2P) networks are decentralized networks of computers, which are also referred to as nodes or participants. The computers directly interact with each other. Each of the computers is assigned a non-negative load. Loads represent various computational taks such as, CPU usage, memory utilization, or internet traffic. The main objective when applying load balancing algorithms is to balance the state of the network, meaning that each node has the average load of the network. This objective is achieved by combating over- and unterloading through load transfers and effective scheduling. Heavily overloaded nodes may fail to complete the assigned tasks due to overheating. This risk can be avoided by transfering loads to underloaded nodes. In times when clouds are becoming increasingly popular for data storage, synchronization and computing tasks etc., efficient load bolancing in networks is crucial. In that sense, load balancing enhances coordination in distributed systems.

1.1 Preliminaries

A graph G is composed of two sets. The set $V := \{v_0, v_1, ..., v_{n-1}\}$ called vertices and the set $E := \{e_1, e_2, ..., e_{n-1}\}$ which contains the edges. A graph $G := (V, E)$ may contain only one vertex and zero edges. The number of vertices $|V|$ is often called the order of G. A node and a edge are incident, if the node is an endpoint of the edge. The degree of a node is the number of incident edges. If a edge connects two vertices, these vertices are *adjacent* to each other, or also refered to as *neighboring*

vertices. A loop is an edge with equal endpoints. Multiple edges are edges with same pair of endpoints. A simple graph has no loops, nor multiple edges. A clique is a set of pairwise adjacent vertices. A path is a simple graph whose vertices can be ordered such that two vertices are adjacent, if and only if they are consecutive in the list. A graph is connected if each pair of vertices in the graph G belongs to a path. A graph is classified as bipartite graph, if the set of graph edges is the union of two disjoint independent sets. If G has a path from node u to node v, then the distance from u to v, $d_G(u, v)$ is the shortest length from a $u, v-$path. The diameter of a graph diam$G := max_{u,v \in V} d_G(u, v)$ [4].

(TODO: Make sure that all these information are taken by the slides of Graph theory by schindelhauer.)

1.2 Motivation

The motivation for this thesis stems from the observed performance discrepancies between the Single-Proposal Deal-Agreement-Based protocol proposed by Yefim Dinitz, Shlomi Dolev and Manish Kumar [3] and the Push-Pull Sum Protocol described in the paper by Saptadi Nugroho, Alexander Weinmann and Christian Schindelhauer [1] across different network topologies. The obervations were gathered in a the student project with the title "Comparative Analysis of Load Balancing Algorithms in General Graphs" [5]. Judging from the simulations conducted in the student project, the Push-Pull Sum Protocol seems to perform better in reducing the mean squared error for the complete graph and the star graph compared to the Deal-Agreement-Based algorithm performing better in reducing the mean squared error per round for the torus grid graph and the ring graph. The simulations were conducted for different network sizes. The network size also played an role in faster convergence.

To address this, the proposed research introduces a novel algorithm, the Adaptive

Threshold Push-Pull Sum algorithm that leverages the strengths of both protocols, creating a Trade-off to mitigate their individual weaknesses. The newly introduced algorithm uses the mechanic of adaptive thresholding, in order to prevent load transfers with low effect in error reduction, since load transfers are pricy operations. By adapting to the structural characteristics of the investigated networks, this new solution aims to achieve robust performance across a wide range of scenarios, bridging the gap between the Deal-Agreement-Based algorithm and the Push-Pull Sum algorithm.

1.3 Related Work

Nugroho et al.[1] proposed the Push-Pull Sum algorithm, which essentially is a composition of two algorithms: the push algorithm proposed by David Kempe, Alin Dobra and Johannes Gehrke [2] and the Pull-Sum algorithm. The Push and the Pull mechanics are directly adopted from the Push-Pull Sum algorithm. Nugroho et al. used the mean squared error as a metric to evaluate the performance of their algorithm. We will follow an similar approach. Nugroho et al. conducted their experiments in static general graphs, which is also the case for this research. Dinitz et al. [3] suggested two versions of the Single-Proposal Deal-Agreement-Based Protocol two versions of a Multi-Neihbor Load Balancing Algorithm, a round robin approach and a Self-Stabilizing load balancing algorithm. However the only comparable algorithm with ours are the Single-Proposal Deal-Agreement-Based protocols, from which there exist two variations. One for the contiunous setting and one for the discrete setting. The main difference between these two is that in the contiunous setting any load may be transfered over the edges and in the discrete setting all load transfers must contain integers. For Multi-Neighbor load balancing the nodes may transfer loads to several neighbors in one round. This is only the case for the the pull actions where one node responds to every calling node by sending loads back.

The Self-Stabilizing and the round robin approach are asynchronous algorithms and so are not comparable to our synchronous algorithm.

1.4 Hypothesis

The novel hybrid load-balancing algorithm, which integrates key features of the Deal-Agreement-Based Protocol and the Push-Pull Sum Protocol, will demonstrate performance that is intermediate between the two in terms of mean squared error (MSE) reduction across six distinct network topologies. Specifically, it is expected to perform better than the Deal-Agreement-Based Protocol in high-degree networks and perform better than the Push-Pull Sum Protocol in low-degree networks, achieving a balance that improves overall adaptability and efficiency compared to the both.

This hypothesis will be tested through comparative analysis of the MSE for six distinct topologies, demonstrating the algorithm's robustness and scalability across network environments. Model fitting is applied as a analysis technique to see the trend of the data and be able to make statements regarding the convergence rate.

1.5 Contribution

This study introduces a novel load balancing algorithm that combines the strengths of two established approaches randomized load balancing and deal agreement-based balancing while integrating an adaptive threshold mechanism to enhance performance by adapting to the current conditions. The load balancing algorithm uses the push and pull mechanics for convergence to a balanced state.

2 Problem Overview

We consider an undirected general graph $G_r = (V, E_r \subseteq V \times V)$, where V denotes the set of vertices (nodes) and E_t denotes the set of edges at each round r. A load transfer between two nodes u and v may happen, if the two nodes are connected by an edge.

2.1 Setting

The Peer-to-peer network is modeled as a static general graph, meaning that the set of edges may not change during the application of the load balancing algorithms. The main objective is to balance the state of the network. This objective will be achieved with local algorithms, so that each node only collects information of nodes in their direct neighborhood. While the Deal-Agreement-Based approach is solving the load balancing problem by determinism, the Push-Pull Sum protocol and the Adaptive Threshold Push-Pull Sum protocol use elements of randomness, to spread loads more evenly. The nodes for the Push-Pull Sum protocol are provided with the initial sum $s_{i,0}$ and weight $w_{i,0}$ values and list of neighbors as initial information. The Deal-Agreement-Based protocols nodes are provided with initial node values and a list of neighbors. The initial weight values for each node i $w_{i,0} = 1$. The sum of all weights $\sum_i w_i, r$ at any round r is equal to the network size n and the sum $s_{i,0}$ is equal to the data input [1]. The setting is a contiunous setting, where data transfer over the edges may contain any amount not just integers. We assume an

synchronous message delivery, where the time of message delivery is constant e.g. $O(1)$ time. Eventhough it is possible to make the asynchronous model synchronous by simply enlarging the time unit, we stick with the synchronous model, where each round takes $O(1)$ time.

2.2 Approach

This research contains of three steps, the design of a load balancing algorithm composed of elements of two established load balancing algorithms, the simulation step to test the ability to balance the state of the network for distinct topologies, and a comparative analysis step, where the simulation outcomes are evaluated, using methods from statistics. In the previous research "Comparative Analysis of Load Balancing Algorithms in General Graphs" [5], the strengths and weaknesses of two distinct load balancing algorithms were determined, by simulating them in different topologies for different network sizes to test the scalability and adaptability of the algorithms to different situations. The design of a novel adaptive threshold load balancing algorithm is based on this knowledge. Simulation outcomes for 30 distinct experiments for further statistical significance, since we operate with randomized algorithms, are gathered and finally evaluated using model fitting to determine the trend of the algorithms regarding mean squared error reduction, and slopes are calculated per round to see the consistency in mean squared error reduction. The results are presented in comprehensive plots, with ellaboration on why the plots look like they do.

3 Algorithms

In the following the three load balancing algorithms under test are presented. Firstly, the classic Push-Pull Sum algorithm and the Contiunous Single-Proposal Deal-Agreement-Based Algorithm are introduces. The novel algorithm that is proposed in this paper, the Adaptive Threshold Push-Pull Sum algorithm is then introduced, and it is depicted how the algorithm is composed of the other two algorithms, and why the protocol is composed like that. For all algorithms the pseudo-code and a description of how the algorithms work is presented. Furthermore, example are provided to enhance the understanding of how the algorithms balance load. For the Adaptive Threshold Push-Pull Sum Protocol additionaly the aspired outcome is presented.

3.1 Classic Push-Pull Sum Protocol

The Push-Pull Sum algorithm as proposed in [1] requires each node to have sum $s_{i,r}$ and weight $w_{i,r}$ values as information. The initial weight $w_{i,0}$ for each node is equal to 1. The sum of all weights is equal to the network size N. The sum of $s_{i,0}$ is equal to whatever the required input $x_i \in \mathbb{R}_0^+$ is, in the paper the values for the sums are uniformly distributed values between 0 and 100 [1]. The pseudo code is to be found in figure 3. The Push-Pull Sum algorithm is composed of three different procedures, namely the *RequestData* procedure, the *ResponseData* procedure and the *Aggregate* procedure. In every round r the *Aggregate* procedure is called, except for the first

round, since there is no information to be aggregated. In this procedure, every node gatheres all incoming messages $Mnode, r$ sent by other nodes $\{(s_m, w_m)\}$ in the previous round $r-1$, requesting data. Also, nodes update their sum values which essentially is $\sum_{m \in M_{node,r}} s_m$ and weight values $\sum_{m \in M_{node,r}} w_m$. And finally these values are form the respective load values by dividing sum by weight. Following that, each node calls the *RequestData* procedure. In this procedure, each node chooses a random neighbor node and sends half of its sum $\frac{s_{node,r}}{2}$ and half of its weight $\frac{w_{node,r}}{2}$ to the chosen node and itself. This is the push mechanism. The pull mechanism is described in the *ResponseData* procedure. Here, each node gathers the incoming requests per round r in a set $R_{node,r}$. Then, each node replies to each requesting node (including itself) with the half of its sum value divided by the number of incoming requests, so $\frac{\frac{s_{node,r}}{2}}{|R_{node,r}|}$.

Algorithm 1 Push-Pull Sum protocol

procedure RequestData
    Chose a random neighbor node v
    Send $(\frac{s_{u,t}}{2}, \frac{w_{u,t}}{2})$ to the chosen node v and the node u itself
end procedure
procedure ResponseData
    $R_{u,t} \leftarrow$ Set of the nodes calling u at a round t
    for all $i \in R_{u,t}$ **do**
        Reply to i with $\left(\frac{\frac{s_{u,t}}{2}}{|R_{u,t}|}, \frac{\frac{w_{u,t}}{2}}{|R_{u,t}|}\right)$
    end for
end procedure
procedure Aggregate
    $M_{u,t} \leftarrow \{(s_m, w_m)\}$ messages sent to u at a round $t-1$
    $s_{u,t} \leftarrow \sum_{m \in M_{u,t}} s_m, w_{u,t} \leftarrow \sum_{m \in M_{u,t}} w_m$
    $f_{avg} \leftarrow \frac{s_{u,t}}{w_{u,t}}$
end procedure

The Push-Pull Sum algorithm has the mass-conservation property [1]. This property ensures that the values will converge to the same value but not the correct aggregate of the ground truth. They converge to the true mean. The setting in that paper is similar to ours. While they only inspect a complete graph with 10^4 nodes, we have network sizes of 2^{10} nodes and more topologies under test. 50 experiments each

conducted for 30 rounds were performed. The paper showed that the Push-Pull Sum protocol decreases the expected potential Φ_r exponentially. The potential function is defined as $\Phi_t = \sum_{i,j} \left(v_{i,j,r} - \frac{w_{i,r}}{n}\right)^2$. The $v_{i,j,r}$ component stores the fractional value of node j's contribution at round r. The conditional expectation of $\Phi_r + 1$ for the Push-Pull Sum protocol is $\mathbb{E}[\Phi_r + 1 | \Phi_r = \phi] = (\frac{2e-1}{4e} - \frac{1}{4n})\phi$ [1].

The Push-Pull Sum protocol seemed to perform very well for the complete graph, the Ring of Cliques with large clique size and Lollipop graphs with large clique size, and the star graph. For the star graph the internal node acts as a distributor of the load. While every leaf chooses with 100% possibility the internal node as a "random" neighbor the internal node is involved as a endpoint of $n-1$ external push operations, and redistributes the load via pull operations to the leaves, where the sum is $\frac{\frac{s_{i,r}}{2}}{n-1}$ and accordingly the weight is $\frac{\frac{w_{i,r}}{2}}{n-1}$. A different explanation applies for the Complete graph, the Ring of Cliques and the Lollipop graph, where the density of each node plays an crucial role. Nodes choose a random neighbor, since the dense graphs have many edges. Randomly choosing a neighbor allows the algorithm to spread loads effectively without relying on a deterministic path, reducing the likelihood of bottlenecks or overloading a single node. That is the reason, why the Push-Pull Sum algorithm did not perform so well for the Torus Grid graph and the Ring graph, where the number of edges is limited to 4 and 2 respectively. Here bottlenecks may occur, since two nodes may push back and fourth, with the sum being $\frac{s_{i,r}}{2}$ and the weight being $\frac{w_{i,r}}{2}$, eventhough the load of the nodes may already been averaged and better options for load transfers are ignored due to the lack of determinism.

3.1.1 Example

The example in figure **??** depicts a complete graph K_4 (4 nodes) labeled from A to D. Each node has a initial sum and weight value assigned. The undirected graphs depict the connections indicating which nodes are neighboring nodes. The solid directed edges depict the push operations and the dashed directed edges depict the

pull operations. Each node has a set $R_{i,r}$ where i is the node id and r is the round where we inspect the set. The load of the node is calculated by $\frac{s_{i,r}}{w_{i,r}}$. The example depicts the first round of a execution of the Push-Pull Sum algorithm. The push and pull operations are distinct. The left-hand side depicts the behaviour of the nodes while executing the push operations, while the right-hand side depicts the case of pull operations. Each node chooses a random neighbor to push load to. A selected node C and pushes half of its sum and weight to the chosen node C and to itself (the loops are not included into the graphics, due to the readability of the figures). Node B chose A as a trading partner, node C and D push to node B and themselves. Given the push operations for each node the set $R_{i,r}$ is computed. Due to node B pushing load to node A and node A pushing load to itself, the set $R_{A,1}$ evaluates to $\{A, B\}$. The updated sum and weight values can be insptected on the right hand side of figure **??**. Following the push actions, each node proceeds with the *ResponseData*-procedure. For all the nodes in $R_{i,1}$ the nodes reply with the pull values. Since node A has two nodes in its set $R_{A,1}$, node A replies with $\left(\frac{\frac{3}{2}}{2}, \frac{\frac{1}{2}}{2}\right)$ to node B and itself, indicated by a dashed directed edge. Accordingly, the remaining nodes *B, C and D* execute the pull operations. Following the push and pull operations the nodes sum and weight values are updated. The setting after round one is depicted in figure **??**. The mean squared error in the beginning of round 1 is 542.50, following one round of applying the Push-Pull Sum algorithm the mean squared error after round 1 is 63.49. The average values (loads) can be taken from table 1.

3.2 Contiunous Single-Proposal Deal-Agreement-Based Protocol

The Continuous Single-Proposal Algorithm Deal-Agreement-Based algorithm proposed by [3], is unlike the Push-Pull Sum protocol not an diffusion-based load

balancing algorithm. The goal of load balancing is achieved based on deterministic deal-agreements, where one node proposes node to one neighboring node and the neighboring node either accepts transfer proposal in either full or partialy. Dinitz et al. algorithm is a anytime algorithm, as they never the worsen the state of the network during execution. They can be stopped at *any time* of the execution. They studied the algorithm in a dynamic setting, which means that the graph may experience changes between rounds. Furthermore, the algorithm is a localized one, where the nodes only work with the information regarding the load state of themselves and their neighbors load state. Each node has a set of neighboring nodes and their including the node's initial load itself as initial information. The algorithm is divided into three phases. There is the *proposal*, *deal* and the *summary*-phase. In the *proposal*-phase each node u contacts the minimal loaded neighbor v and sends a proposal to that neighbor, if the neighbor is less loaded. The proposal is of value $(\frac{load_r(u)-load_r(v)}{2})$, which is labeled as a *fair* proposal. Since the load transfer is fair, the resulting load of u is not lower than that of v. Following that, the nodes enter the *deal*-phase. In this phase nodes evaluate the deals proposed to them. A node accepts the deal of the node that proposes the maximal load transfer. The actual transfer happens and the load values are being updated. Finally, in the *summary*-phase each node informs their neighbors regarding their updated load values. [3] The analysis in this paper is based on a potential function. Dinitz et al. defines potential for a node u as $p(u) = (load(u) - L_{avg})^2$ where L_{avg} is the current load average in the network. The potential for the Graph $p(G)$ is defined as $p(G) = \sum_{u \in V} p(u)$, which essentially is the sum of all potential of each node in the graph G. Any fair load transfer of load l decreases the potential of the graph by at least $2 * l^2$. And as a result of any round r of the Contiunous Single-Proposal Deal-Agreement-Based Protocol, the graph potential decreases by at least $\frac{K_r^2}{2D_r}$ where K is the initial discrepancy and D is a bound for the graph diameter. [3]

The Single-Proposal Deal-Agreement-Based load balancing algorithm seems to have difficulties in reducing the mean squared error as rapidly as the Push-Pull Sum

Algorithm 2 Continuous Single-Proposal Deal-Agreement-Based protocol

Input: An undirected graph $G = (V, E, load)$
Output: A load state with discrepancy at most ϵ on G

```
1: for r = 1 and on do
2:     for every node u do
3:         Find a neighbor, v, with the minimal load
4:         if load_r(u) − load_r(v) > 0 then
5:             u sends to v a transfer proposal of value (load_r(u) − load_r(v))/2
6:         end if
7:     end for
8:     for every node u do
9:         if there is at least one transfer proposal to u then
10:            Find a neighbor, w, proposing to u the maximal transfer
11:            Node u makes a deal: informs node w on accepting its proposal
12:            The actual transfer from w to u is executed
13:        end if
14:    end for
15:    for every node u do
16:        Node u sends the updated value of its load to its neighbors
17:    end for
18: end for
```

algorithm for dense graphs like the a complete graph, the Lollipop graph with large clique size, the ring of cliques with large clique sizes. This seems to be the case, since each node is looking for a minimal load partner. For a complete graph the minimal load partner are the same nodes with same loads equal to the minimal load of the network. This causes each node to propose to the same node with L_{min} which then evaluates the proposals, and accepts exactly one transfer proposal, namely the maximal one. An analouge scenario happens for the Ring of Cliques and Lollipop graph, with the difference that for the Ring of Cliques not each node is proposing to the same minimal neighbor, but for the minimal laoded neighbor within their respective cliques. For the Lollipop graph, the nodes in the path graph balance their loads more quickly than the nodes in the clique which is a bottleneck in this scenario. Looking at the Torus Grid graph and the Ring graph, this Deal-Agreement-Based protocol seems to perform very well in reducing the error, since the proposals are more distributed for the nodes due to the density of the graph, and thus more nodes

are involved in load transfers. The star graph makes an exception to this case, since we have a similar bottleneck as in the complete graph, where the internal node is the common neighbor for each leaf and thus each leaf proposes load to the internal node. The internal node, then chooses only the maximal proposing transfer. The simulation results in the student project [5] indicate the performances of each load balancing algorithm for each topology very clearly.

3.2.1 Example

Figure **??** depicts two settings. The setting is the same as in the example above in the Push-Pull Sum section, with the difference that each node has a load value assigned, instead sum and weight values. The dashed directed edge in this example represent a proposal from one node to another. The solid directed edge represents the actual load transfered from one node to another. We again consider the nodes *A, B, C and D*. Each node looks for its minimal loaded neighbor. For nodes *B, C and D* this is node *A*. For node *A* this is node *B*, since node *B* has more load than node *A*, node *A* does not send an transfer proposal to node *B*. Nodes *B, C and D* each send a transfer proposal of value $\frac{(load_r(i)-load_r(A))}{2}$ to node *A*. Node *A* evaluates each transfer proposal and accepts node *C*'s transfer proposal, since node *C* proposes the maximal amount of load, namely 29.5. The actual transfer happens and 29.5 of loads are transfered from node *C* to node *A*. The right-hand side of figure **??** depicts the setting after round 1. Node *A* and *C* each have a load of 32.5, the loads of nodes *B* and *D* are unchanged. The mean squared in the beginning of round 1 is at 542.5, in the end of round 1 the mean squared error decreases to a value of 107.375, which is close to a fifth of the initial mean squared error.

3.3 Adaptive Threshold Push-Pull Sum Protocol

The Adaptive Threhsold Push-Pull Sum algorithm is composed of different ideas and elements of the two former algorithms, extended by the idea of adaptive thresholding. The Adaptive Threshold Push-Pull Sum consits of different procedures. In the *Check-ThresholdsRequestData* each node i chooses a subset RN of $\log_2(|neighborhood|)$ random neighbors. This is to enhance the chance of a good load transfer to happen. If we only choose one neighbor the chance is lower to find an optimal or good neighbor to execute a load transfer. Then the load difference between the node i and each node in RN is computed and checked if the load difference is bigger than some threshold θ. The threshold is computed in the *CalculateThresholds*-procedure. The threshold θ is computed as $k * \sqrt{MSE_r - 1}$ where k is some factor to adjust the sensitivity of the threshold. A larger k means the threshold is more sensitive meaning that less nodes are eligiable for load transfer and vice versa. This condition prevents load transfers with low effect, and guarantees a selection of nodes with effect above the given threshold. The first eligiable node receives the sum and the weight in form of a push action ($\frac{s_i,r}{2}$) for the sum and ($\frac{w_i,r}{2}$) for the weight. The *ResponseData*, and the *Aggregate*-procedure are analoge to the classic Push-Pull Sum algorithm. The push and the pull mechanism are directly taken from the Push-Pull Sum algorithm, thus the name Adaptive Threshold Push-Pull Sum algorithm. Two ideas are derived from the Single-Proposal Deal-Agreement-Based algorithm. The idea of the conditional load transfer is analog to the Single-Proposal Deal-Agreement-Based's one, with the only difference that the difference of load between the two negotiating nodes shouldn't be larger than 0, but bigger than a threshold θ. And instead of initiaing a load transfer with the maximal loaded node, the Adaptive Threshold Push-Pull Sum algorithm orders the nodes to initiate a load transfer with the first node proposing load. The reason for that is that we want to avoid that each node looks through the whole set R_n and needs to evaluate each proposal, instead the first node proposing a load transfer is accepted as a transfer partner.

Algorithm 3 Adaptive Threshold Push-Pull Sum protocol

procedure CalculateThresholds
    $\theta \leftarrow k * \sqrt{MSE_{r-1}}$
end procedure
procedure CheckTresholdRequestData
    $RN \leftarrow$ choose $\lceil \log_2 (|neighborhood(u)|) \rceil$ random neighbor
    for every node $v_i \in RN$ **do**
        $\Delta_{u,v_i} \leftarrow |(load(u) - load(v_i)|$
        if $\Delta_{u,v} > \theta$ **then**
            Send $(\frac{s_{u,t}}{2}, \frac{w_{u,t}}{2})$ to first node v fulfilling condition and the node u itself
        end if
    end for
end procedure
procedure ResponseData
    $R_{u,t} \leftarrow$ Set of the nodes calling u at a round t
    for all $i \in R_{u,t}$ **do**
        Reply to i with $\left(\frac{\frac{s_{u,t}}{2}}{|R_{u,t}|}, \frac{\frac{w_{u,t}}{2}}{|R_{u,t}|} \right)$
    end for
end procedure
procedure Aggregate
    $M_{u,t} \leftarrow \{(s_m, w_m)\}$ messages sent to u at a round $t-1$
    $s_{u,t} \leftarrow \sum_{m \in M_{u,t}} s_m, w_{u,t} \leftarrow \sum_{m \in M_{u,t}} w_m$
    $load(u) \leftarrow \frac{s_{u,t}}{w_{u,t}}$
end procedure

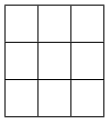

Table 1: Overview over example outcomes

3.3.1 Example

Figure **??** depicts a similar setting as described in the section 3.1.1. Now, according to the Adaptive Threshold Push-Pull Sum algorithm, each node chooses $\log_2(|neighborhood|)$ neighbors. For this setting, this means that each node chooses 2 neighbors. These neighbors are added to the set $RN_{i,1}$ for each node i. For instance, node A computed $RN_{A,1}$ as $\{B, C\}$. The load differences between node A and nodes B and C are computed respectively. In this example, k is 0.01, thus the threshold Θ in this case is $0.01 * \sqrt{542.5} \approx 0.23$. Since both load differences between nodes A and B, and nodes A and C pass this threshold, both nodes are eligiable to propose to transfer load with node A. In this scenario node A chooses node B as a transfer partner and pushes half of its sum 1.5 and weight 0.5 to node B and itself. Accordingly, for each node respectively. Similar to the example in section 3.1.1 the nodes execute the pull operation. The result is depicted in figure **??**. The mean squared error dropped from 542.5 initially, to 251.86.

3.3.2 Aspired Outcome

Thinking of the simulation results presented in [5] the discrepancy between the MSE reduction per round for the different topologies show that the algorithms perform either very well or medicore to bad. The main idea and perspective designing this algorithm was to find a compromise solution to that, to provide a better adaptability for different network topologies.

(TODO: Table with the results of the algorithms during and after round 1 summed up.)

4 Topologies

The simulations were conducted for six distinct network topologies. Each network has 2^{10} (1024) nodes. The behaviour of the algorithms in these different topologies is observed, since each topology has different characteristics, different performances exploiting the specials of the topologies are expected. The topologies contain the *complete graph*, *Torus Grid Graph*, *Ring Graph*, *Star Graph*, *Lollipop Graph* and *Ring of Cliques*. In the following the topologies including there characteristics are presented.

(TODO: introduce and check the abbr. for each graph: complete graph: K n and so on and verify new edge size and so on)

4.1 Complete Graph

The complete graph K_n as illustrated in figure **??** is a graph where each pair of distinct nodes is connected by an edge. The complete graph or also refered to as fully-connected graph has n nodes and $\frac{n\times(n-1)}{2}$ edges. The complete graph is a regular graph, where each node has a degree of $n-1$. As it contains the maximum possible number of edges for a given set of nodes (the maximum number of edges per node for undirected graphs is n, when self-loops are allowed), the complete graph is considered a dense graph [4]. The diameter of a complete graph is 1, as each node is directly connected to every other node. Since node has same degree and connectivity, algorithms can treat all nodes uniformly.

(TODO: images auskommentieren und mehr zu den besonderen Charakteristiken der einzelnen graphentypen schreiben)

4.2 Torus Grid Graph

The two-dimensional torus grid graph or also refered to as $m \times n$-torus graph is a two-dimensional mesh with wrap around edges as depicted in figure **??** [6]. m denotes the height and n denotes the width of the grid. The torus grid graph has $2 \times m \times n$ edges and $m \times n$ nodes and is a regular graph, where each node has a degree of 4. The diameter of a torus grid graph is $\frac{min(m,n)}{2}$. Tori are scalable, as the number of connections per node is constant, regardless of the graph size. In the previous research "Comparative Analysis of Load Balancing Algorithms in General Graphs" [5], we saw this as the simulation results did not vary over different network sizes.

4.3 Ring Graph

The ring graph is a regular graph, where each node has a degree of two, forming a cycle. The ring graph can be constructed by building a path graph which is closed, meaning that the first and the last nodes of the path graph are connected by an edge as depicted in figure **??**. The graph consists of n nodes and n edges. The diameter of a ring is $\lfloor \frac{n}{2} \rfloor$. The uniform structure ensures no single node has an advantage, simplifying algorithm design and guaranteeing fairness in load balancing.

4.4 Star Graph

A star graph as depicted in figure **??** is a bipartite graph [**?**], structured like a tree graph with one internal node and k-leaves (but no internal nodes and $k + 1$ leaves

when $k \leq 1$). The internal node belongs to one of these disjoint sets, while the leaves are assigned to the other set. A star graph with n nodes has $n-1$ edges. In this structure, every leave node has a degree of 1, meaning it is connected only to the internal node. The central node has a degree of $n-1$. The diameter of a star graph is 2 (path from one leave through internal node to another leave). Regarding load balancing the internal node acts as a point of redistribution. This topology is particularly suitable for master-slave or client-server models where the central node delegates tasks and collects results. However, the central node may become overloaded in high-load scenarios, requiring careful design to prevent bottlenecks.

4.5 Lollipop Graph

A (k, m)-lollipop graph ($L_{k,m}$) is a graph that consists of a complete graph and a path graph as depicted in figure **??**. The complete graph and the path graph are connected by a bridge node, thus a single edge. The (k, m)-Lollipop graph is composed of a complete graph with k nodes and a path size of m nodes. A lollipop graph has $k + m = n$ nodes and $\binom{k}{m} + k$ or $(\binom{k}{2} + (n - k - 1))$??? edges [**?**].

4.6 Ring of Cliques

The $(k \times m)$-ring of cliques, consists of k cliques, each containing k nodes. The cliques are connected to form a ring structure. To create the ring, one edge from each clique is removed, and the endpoints of these removed edges are connected to form a regular graph [6]. A $k \times m$ ring of cliques has $\left(k \times \left(\frac{m \times (m-1)}{2} - 1\right)\right) + k$ edges. The connectivity of the graph increases with larger clique sizes and decreases with smaller clique sizes.

4.7 Expected Outcome

(TODO: Remove these notes/reformulate them) Many load balancing algorithms perform well on 2D torus grids due to the focus on local redistribution, thanks to the wrap around edges eliminiating boundary effects, ensuring balanced redistribution across entire grid.

Ring: Unlike complete or torus graphs, the sequential nature of data transfer makes it slower for loads to propagate across the entire ring. Load movement in the ring graph is sequential, making it slower and less flexible than the complete graph.

Star: For the Push-Pull mechanism the star graph is an advantage, since each push action contains a pull action. All leaves request data from the internal node, so the internal node acts here as an point of redistribution. However for the Deal-Agreement-Based protocol some mechanisms like "if condition" act as a contraproductive way of load balancing.

Lollipop: Algorithms must account for the highly connected complete graph (dense communication) and the linear path graph (sequential communication), which introduces a mixed topology challenge. The node linking the complete graph and the path graph often becomes a bottleneck since it bridges two vastly different connectivity regions. In the complete graph, load balancing is efficient due to the high connectivity, enabling rapid redistribution among nodes. Load movement in the path graph is sequential, making it slower and less flexible than the complete graph. Interesting because dense head and sparse tail. For example, diffusion-based approaches work well in the head but may require additional iterations in the tail. We saw that in "Comparative Analysis of Load Balancing algorithms in General Graphs" [5], once we changed the path size in relation to same complete graph, the DAB was faster in reducing error for the first 50 rounds. **(TODO: Do the simulations over again with different path and complete graph size and compare.)**

Ring of cliques: Within each clique, balancing is fast due to the dense connectivity. Load redistributes efficiently among the clique's nodes. Balancing between cliques depends on the single shared node linking each pair of cliques. This structure ensures sequential redistribution across the ring. The shared nodes act as gateways, controlling the flow of load between cliques. Overloading these nodes can cause bottlenecks.

5 Implementation and Technology Stack

Different technologies were used to achieve the underlying results. Simulations are conducted using *PeerSim* a simulation framework for *Java*. As an artefact of a simulation a output file containing different simulation parameters, settings and metrics is generated. The data analysis part of this project is mostly implemented using *Python* as a programming language.

5.1 Programming Languages

Python *3.12.6* is used for several tasks, mostly for preparing and analyzing data necessary to concuct simulations and post simulation analysis. For plotting purposes *matplolib v.3.9.1* is used and for handling datas and data analysis *scipy v.1.14.1* is used.

For conducting the simulations itself *Java Oracle OpenJDK 21.0.1* and *PeerSim v.1.0.5*.

5.2 Simulation Framwork

PeerSim is a simulation tool developed for the Java programming language, which is composed of two engines. The cycle-driven engine and the event-driven engine. We

chose PeerSim for our simulations, since PeerSim is suited for large scale Peer-to-Peer simulations. In previous projects, we were able to conduct simulations up to 2^{14} nodes and could probably push the boundaries by scaling to even higher dimentsions. PeerSim is designed to use pluggable components, implemented in interfaces, which are intuitive to use generally. A simulation in PeerSim can be realized following four simple steps. First, in order to probably set up a simulation a **configuration file** is needed. Here, simulation parameters may be defined, like the network size, the load balancing protocols that are beeing simulated etc.

```
1  # network size declaration and initialization
2  SIZE = 1024
3  network.size SIZE
4
5  # Synchrounous CD Protocol
6  CYCLES = 100
7  simulation.cycles CYCLES
8
9  # classic PPS protocol definition
10 protocol.loadBalancingProtocols loadBalancingProtocols.
11 PushPullSumProtocol
12 protocol.loadBalancingProtocols.linkable loadBalancingProtocols
13
14 # Control classic PPS
15 control.avgo loadBalancingProtocols.PushPullSumObserver
16
17 # The protocol to operate on
18 control.avgo.protocol loadBalancingProtocols
19 control.avgo.numberOfCycles CYCLES
```

Listing 5.1: Example Configuration

Listing 5.1 shows parts of an configuration file used in our project. This configuration sets up a simulation of the Push-Pull Sum algorithm for a network with 1024 nodes (**lines 2 and 3** in Listing 5.1) for 100 rounds (**lines 6 and 7** in Listing 5.1). From **lines 10 to 12** the Push-Pull Sum protocol is loaded. The way we implemented the algorithms, one algorithm contains of at least two Java classes, the *<ProtocolName>Protocol.java* and the *<ProtocolName>Observer* and a shared *loadBalancingParameters*-class containing parameters like the cycle or topology specific parameters as depicted in figure **??**. Finally at **line 15** the controll class is declared and used at lines **18 and 19**, with the parameters of type **protocol** and **numberOfCycles**. And so step 2 and three of creating a simulation is also handled by **selecting the protocols to simulate** and **select control objects**. Following that, the last step is to **invoke the simulator class**, which is *peersim.Simulator.class*. For that, the IDE may be configured to call the simulator class on execution of the program code. Alternatively, a command line in the terminal can also invoke the simulator class. More on this in the PeerSim documentation [7].

PeerSim provides a variety of classes and interfaces for simulation. For the cycle-driven approach, the framework offers the CDProtocol interface, which defines the nextCycle method. The nextCycle method is executed at the beginning of every simulation round.

In PeerSim, nodes are implemented as containers that hold various protocols. Each node is uniquely identified by an ID and interacts with the Linkable interface, which provides access to neighboring protocols. A class implementing the Linkable interface can override several methods, such as:

- **getNeighbor**: Retrieves a neighbor with a specified ID.
- **degree**: Returns the number of connections (or neighbors) a node has.
- **addNeighbor**: Adds a neighbor to the node's set of neighbors.

To monitor or modify simulations, control objects are required. A class implementing the Control interface must define the execute method, which can be used to observe or alter the simulation at each round [7].

5.3 Implementation Details

Figure **??** depicts a process model, modelling the methodic chosen to get from the creation of experiments to the plots and data analysis part. First I wrote a Python script that generates configuration files where each node has uniformly distributed random load/sum values. 50 distinct experiments were created to improve statistical significance. These configuration files are read with a Java script and each node is assigned a initial load value. Then the simulations are conducted. Each simulation outputs a file containing the simulation results, mainly the mean squared error per round, the loads per round and the configuration of the network (e.g. which topology chosen, network size...). The simulation results are averaged per round and then analyzed. Out of the simulation results plots are generated, showing the mean squared error reduction per round in log-log or log-linear graphs. Also the technique of model fitting is applied.

6 Simulation Outcomes

In the following the load balancing algorithms are abbreviated. The Single-Proposal Deal-Agreement-Based algorithm is abbreviated as DAB, the Push-Pull Sum algorithm is abbreviated as PPS and the Adaptive Threshold Push-PullS-Sum algorithm is abbreviated as ATPPS. The simulation outcomes are presented in plots, mostly log-log or log-linear graphs (logs are base 10), since the mean squared error values vary a lot over the 100 rounds of simulation. Figures 2, <slopes images referencing> depict the slopes for three distinct regions (of the x-axis) and the general slope over all the 100 rounds.

6.1 Complete Graph

Figures 1 show the mean squared error reduction over the rounds for the three load balancing algorithms simulated on the complete graph on a log-log graph. **Deterministic Agreement-Based Algorithm (DAB)**: One can observe that the black line shows constant mean squared error reduction, indicating that this algorithm does not improve the load balance by much over time. The DAB performs poorly due to the rigid determinism; DAB uses a fixed, deterministic load redistribution rule, where one node looks for the minimal loaded neighbor and proposes to that node. Since for the complete graph each node are interconnected, the neighbors with the minimal loads are the same for every node. Thus, the deterministic strategy proposing to the minimal neighbor does not distribute load in suboptimal ways. The

number of load transfers in this scenario per round is very limited, resulting in poor mean squared error reduction. This is also observavle in the slopes depicted in figure 2. The main observation is that deterministic methods can struggle in high-connectivity environments because they fail to exploit randomness or adaptivity.

Push-Pull Sum algorithm (PPS): The PPS-curve decreases gradually in mean squared error, showing significant error reduction over time. Since all nodes are equally connected, no structural constraints slow down the process of pushing and pulling load from neighbors. Over time, repeated randomized exchanges smooth out load imbalances, leading to a exponential decay in mean squared error. Classic PPS does not differentiate between "highly imbalanced" and "near-balanced" nodes. Load transfers happen uniformly, even when the system is close to equilibrium, which slows convergence in later rounds.

Adaptive Push-Pull Algorithm (ATPPS): The ATPPS-curve decreases faster than the red curve, reaching lower MSE values earlier. Adaptive Thresholding: Nodes adaptively decide when and how much load to exchange based on thresholds. For example: Nodes with significantly higher load push more load to neighbors. Nodes closer to balance reduce their interactions. Adaptive strategies reduce redundant exchanges when the system is near equilibrium, focusing resources on nodes with the largest imbalance. In a complete graph, adaptive decisions propagate quickly because every node can directly communicate with others. The adaptive mechanism leverages the symmetry and high connectivity of the complete graph to reduce MSE more quickly and efficiently.

Figure **??** shows three figures. On the left-hand side the PPS curve is fitted to the exponential model. **(TODO: Elaborate on Slopes)**

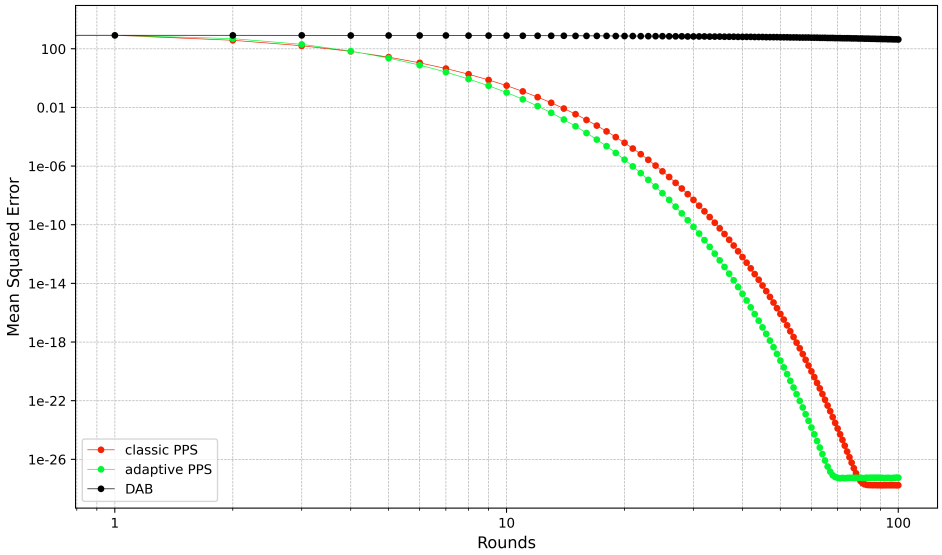

Figure 1: Complete Graph: mean squared error per rounds (log-log)

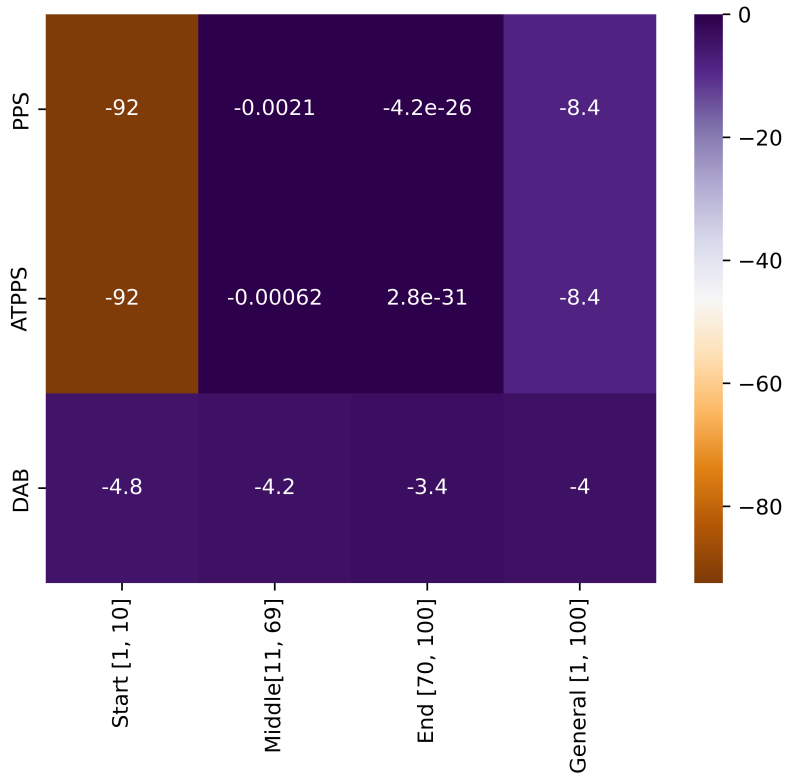

Figure 2: Complete Graph: heat map of slopes per region

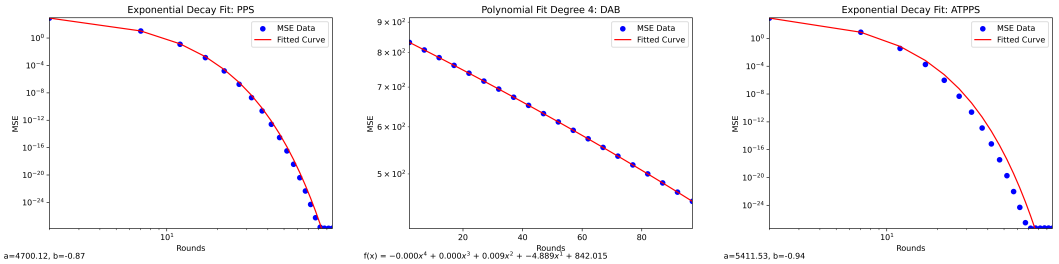

Figure 3: Exponential Decay Fit: PPS and ATPPS; Polyomial Fit DAB

6.2 Star Graph

6.3 Ring Graph

6.4 Torus Grid Graph

6.5 Ring of Cliques

6.6 Lollipop Graph

7 Conclusion

8 Acknowledgments

First and foremost, I would like to thank...

- advisers
- examiner
- person1 for the dataset
- person2 for the great suggestion
- proofreaders

9 Appendix

ToDo Counters

To Dos: 8; 1, 2, 3, 4, 5, 6, 7, 8

Parts to extend: 0;

Draft parts: 0;

Bibliography

[1] S. Nugroho, A. Weinmann, and C. Schindelhauer, *Adding Pull to Push Sum for Approximate Data Aggregation*. Springer, 2023.

[2] D. Kempe, A. Dobra, and J. Gehrke, "Gossip-based computation of aggregate information," in *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, pp. 482–491, 2003.

[3] Y. Dinitz, S. Dolev, and M. Kumar, "Local deal-agreement algorithms for load balancing in dynamic general graphs," *Theory of Computing Systems*, vol. 67, pp. 348–382, Apr 2023.

[4] C. Schindelhauer, "Lecture notes in graph theory," 2021.

[5] E. Bayazitoglu, "Comparative analysis of load balancing algorithms in general graphs." University of Freiburg, 2024.

[6] P. Mahlmann, *Peer-to-peer networks based on random graphs*. PhD thesis, University of Paderborn, 2010. Paderborn, Univ., Diss., 2010.

[7] Drakos, N. and Moore, R., *PeerSim: HOWTO: Build a new protocol for the PeerSim 1.0 simulator*, December 2005. Accessed: August 13, 2024.