Datacenter Network Multi-path Load Balancing using weighted paths and greedy algorithms

Datacenter Network Multi-path Load Balancing

Monkey In My Mind

Datacenter Network Multi-path Load Balancing using weighted paths and greedy algorithms

Abstract

Networking is a key part of distributed systems. Traditional network transmission protocols, such as TCP/IP, only designate a dedicated route for transmitting data between two nodes and cannot fully utilize the fat tree networks. Turns out, such a task can be quite effectively addressed by a greedy algorithm in combination with the use of weighted paths in the graph and some additional optimizations. By using a weight function for the vertices and Dijkstra's algorithm, it is possible to evaluate which paths in the graph are good. However, over time, some vertices become more congested, which, of course, cannot be predicted in advance. Therefore, it is necessary to use dynamic solution models that evaluate the current congestion of network nodes during transmission. This involves two main methods - using a greedy algorithm when choosing the next vertex of the path, and also randomization, which distributes the load across the network. Thus, the solution model can be divided into two parts. The first - static - involves calculating the weights of paths in the graph, which occurs before the start of the request transmission process. The second - dynamic - takes into account the current network congestion and makes decisions on message transmission based on it. It is also worth noting that this is a fairly simple and resource-intensive algorithm, which can be easily optimized further.

Keywords

Multi-path Load Balancing, fat tree, greedy algorithms, weighted paths, random, Dijkstra's algorithm

Introduction

The article will be dedicated to the problem of successful and fast message delivery in a network that is a k-ary tree. Initially, some characteristics of the graph and the processes taking place will be evaluated. Then, a basic model solving the problem will be described, as well as some optimizations that significantly improve the metrics required for the task. The model uses simple algorithms, such as finding weighted paths in the graph, greedy algorithms, and randomization. It is worth noting that the model practically does not use data transmission between vertices. For example, when solving, the cluster controller is not used at all, and all other transmitted data use significantly less memory than is permissible. The described model solves the problem very well: the proportion of successfully delivered requests is almost equal to 1, the average latency of delivering one request does not exceed 9.5, and the biggest latency request is less than 20.

Assumptions and Symbols

current_node - since the model operates with a single algorithm for all vertices, when explaining certain algorithms, we will assume that we are in an arbitrary vertex *current_node*, unless stated otherwise.

to_node - the vertex to which a message is sent during the given time interval from *current_node* (in particular, *to_node* and *current_node* are connected by an edge).

All numerical constants k_{const} are selected based on the test results.

Analysis of the problem

The task is to optimally deliver messages on the fat tree structure. When optimizing message delivery on the fat tree graph, it is necessary to take into account the graph structure, the shortest path for the message from server to server, as well as the characteristics of the vertices (bw_in, bw_out, bw_size) and the available space in their buffers ($buffer\_size$). The maximum access level is the maximum level to which one needs to ascend to deliver the message. From the graph topology, it follows that all shortest paths are determined by a vertex at the maximum access level. The remaining paths will be formed by adding edges to these. It should be noted that the maximum size of the shortest path is 6 (achieved on the path from server to server, corresponding to ascending from level 0 to level 3 and descending from level 3 to level 0).

When the maximum access level of messages is 2, the task reduces to the selection of a single vertex, and if the access level is 3, it is necessary to choose a core vertex.

The second and first levels of vertices locally form a complete bipartite graph, which eliminates a large part of the defects of greedy algorithms (an algorithm in which at each step the locally optimal solution is chosen, without considering possible consequences for subsequent steps).

One can define such an abstract characteristic as $delay = \frac{bw\_in}{bw\_out}$, which will show how many time slices are needed to send all the messages that we received within one time slice. Under sufficient network load, we need the delay to be approximately equal to 1, as otherwise there will be many unsuccessful transmissions. Assuming that the delays at vertices of the same level do not differ significantly, the main parameter for selection will be the $buffer\_size$. To avoid excessive delay and buffer overflow, it is necessary to artificially limit bw_size.

Model building

In the simplest solution model, it is sufficient to maintain all the information coming from the simulator and $buffer\_size_{current\_node}$, and also to send from the vertex no more than its $bw\_out_{current\_node}$ messages to arbitrary vertices connected to it by an edge.

To select the node to which the message will be sent, we introduce a certain function $f(node_1, node_2)$ of two nodes: $node_1$ and $node_2$. The key observation is that when sending a message, the inequality $f(current\_node, target) > f(to\_node, target)$ should be satisfied, where the target is the server to which the message should be delivered.

In the first approximation, let's take the shortest distance function f as the distance in the graph from $node_1$ to $node_2$. In Figure 1, the distance between nodes 18 and 9 is 5 (if counted in edges) and is achieved by two paths: $18-26-33-28-20-9$ and $18-27-34-29-20-9$. This model works much better, however, it does not fully utilize such node characteristics as bw_out, $buffer\_size$, and does not use bw_in and information about the level of nodes at all.

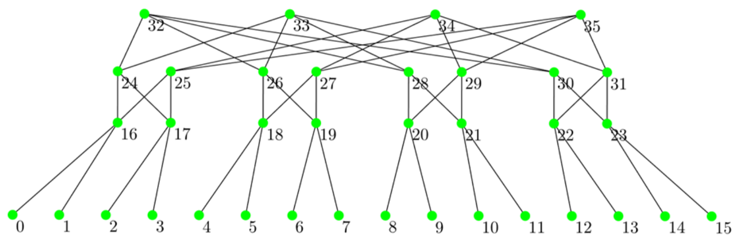

Figure 1: 4-ary fat tree

The following optimization uses the ability to transfer data between vertices and also utilizes information about $buffer\_size$. At the end of each time interval, we will transmit the current $buffer\_size$ of the vertex (notice that it is not difficult to maintain the current $buffer\_size$ of the vertex at any given moment). (Now, when each vertex knows the information about the $buffer\_size$ of its neighbors, we will choose the to_node according to the following rule: among all the neighbors of the $current\_node$ vertex, for which $distance(neighbor, target) < distance(current_node, target)$, we will choose the vertex with the largest buffer. This simple greedy algorithm already works very well (such a model scores around 90 points).

Notice that some vertices have a very small bandwidth, as a result of which messages that enter them cannot be sent out of them over a large number of time slices. Specifically, let's consider the following situation: suppose there is an aggregate switch with a $buffer\_size$ of 1000 and bw_out of 20. Even with the minimum bw_in of this vertex, it can fill up in a short amount of time. However, due to the very small bw_out, some messages can be delayed in it for about $buffer_size/bw_out = 50$ time intervals, which is a lot. In connection with this, the following optimization arises: artificially limit the $buffer\_size$ of each vertex that is not an access switch through its bw_out and bw_in, namely, we will consider that

$$buffer\_size_i = min(k_{buffer} * buffer\_size_i, k_{bw\_out} * bw\_out_i, k_{bw\_in} * bw\_in_i)$$

where $k_{buffer}, k_{bw\_out}, k_{bw\_in}$ are some numerical constants. Notice that with this optimization, $buffer\_size_i$ will practically always be less than the actual. Therefore, there may be a situation where more messages arrive at a vertex in some time interval than the limited $buffer\_size_i$ set by us, i.e. $buffer\_size_i$ can sometimes be negative. To prevent such vertices from overflowing, we will only send messages to those vertices whose $buffer\_size$ is strictly positive.

Although this model is already working quite well, it still does not fully utilize the information about $buffer\_size$, bw_out, and bw_in. Specifically, the vertices only have information about their neighbors, and when choosing to_node for a specific message, the model does not use information about vertices that are not neighbors of the $current\_node$. To fix this, we will rewrite the distance function. Now we will consider not just the distance in the graph between vertices, but weighted distances. To do this, we will introduce a certain function $weight(node_i)$ for each vertex in the graph that is not a server, which uses information about the characteristics of the vertex $node_i$. As such a function, we will take

$$weighti = \frac{bw\_in_i}{k_{level_i} * bw\_out_i + level_i} + 0.5$$

Here K_{flevel_i} is a numerical constant depending on the level of the vertex $node_i$. Now each path in the graph is the sum of the weights of the vertices on it, and the function f is the $weight\_distance(node_1, node_2)$ function, which shows the minimum possible weight of the path between vertices $node_1$ and $node_2$.

This is almost the final version of the model. To improve the model's performance, let's introduce a few more optimizations.

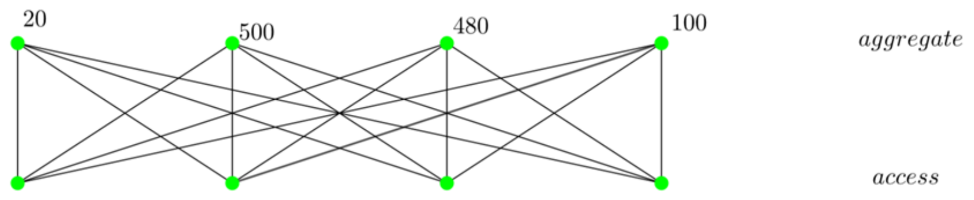

Figure 2: one of the k complete bipartite graphs

Before this, we sent messages from the vertex in random order. However, we notice that it would be much more optimal to send the messages that have been in the vertex the longest first. To do this, before sending the messages, we will sort them by $request\_begin\_time$. In the case of equal $request\_begin\_time$, we will sort them in descending order of distance, which also provides a slight improvement.

Another optimization is related to the fact that the algorithm is not flexible enough when choosing the to_node (among the neighbors of current_node, the algorithm simply selects the node with the largest $buffer\_size$ if the $weight\_distance$ from it is smaller). As mentioned earlier, the structure of a fat tree is such that the nodes of the first and second levels are divided into k parts, each of which forms a complete bipartite graph. Figure 2 shows one of such parts. The numbers indicate the sizes of the buffers.

Therefore, if when choosing the to_node for a certain message, we consider nodes with buffers 480 and 500, we will always choose the node with a buffer of 500. However, the number 480 does not differ much from 500, which leads to the following optimization. Let's maintain two largest buffers, max_buffer_1 and max_buffer_2, among the neighbors of $current\_node$, for which $weight\_distance(neighbor, target) < weight\_distance(current\_node, target)$. And if the inequality $max\_buffer1 < k_{diff} * max\_buffer2$ is satisfied, then with probability p we will send the message to the node with a buffer equal to max_buffer1, otherwise we will send it to the node with a buffer equal to max_buffer_2, where k_{diff} and p are some constants.

This optimization has another interesting variation. Among the neighbors of $current\_node$, we will simply select the node with the largest characteristic size, where

$$size = \frac{buffer\_size_i}{(weight\_distance(i, target) + 1)^{distance(i,target)*k_{deg}}}$$

where k_{deg} is a numerical constant. This formula is directly proportional to $buffer\_size$ and inversely proportional to $weight\_distance * distance^{k_{deg}}$, meaning that the larger the distance from i to target, the more influence $weight\_distance$ has on the formula. Conversely, if the distance is small (for example, equal to 2), then the $buffer\_size$ has the greatest impact on the formula.

Finally, another important optimization is related to the fact that some $access\_switches$ may be too full, causing some requests not to fit in the nodes. To fix this, let's consider messages whose distance is equal to 2 (meaning they are in the $aggregate\_switch$ and their path is uniquely determined by the $access\_switch$). We will try to always maintain the buffer size of the first-level nodes not less than a certain constant k_{access_buff}. To do this, we will introduce a counter $neighbor\_in_i$ inside each node, which indicates how many messages have been passed to node i during a given time interval. Knowing the $buffer\_size$ of the first-level node j as well as the number of deg nodes in the second level connected to each first-level node, let's introduce $in\_accept = \frac{buffer\_size_j - k_{access\_buff}}{deg}$, which shows how many messages can be passed from each second-level node. Thus, at a given moment, a maximum of $in\_accept * deg = buffer\_size_j - k_{access\_buff}$ messages will be passed, which ensures that there is always k_{access_buff} free space in the first-level nodes (of course, provided that $buffer\_size_j k_{access\_buff}$ before the start of the time interval). It is clear that such optimization slightly slows down the message delivery process. Therefore, it is necessary to carefully choose the constant k_{access_buff}.

For example, in Figure 3, $deg = 4$. With k_{access_buff}, no messages will be sent to the node with a buffer of 100, a maximum of 50 messages from each aggregate switch will be sent to the node with a buffer of 350, a maximum of 87 messages from each aggregate switch will be sent to the node with a buffer of 500, and finally, a maximum of 12 messages from each aggregate switch will be sent to the node with a buffer of 200 within one time interval. Figure 3: buffer sizes are also indicated by numbers This concludes the model construction. Let's explain how some algorithms are implemented and explain the correctness of their operation.

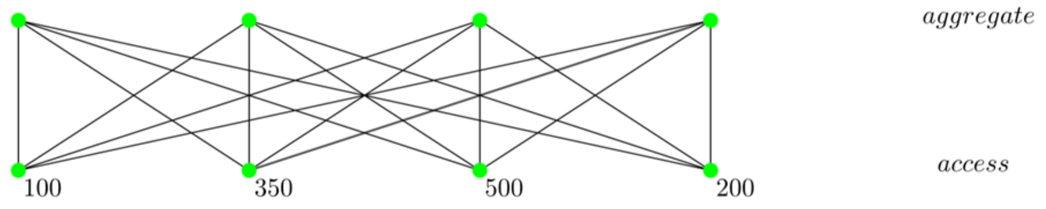

Figure 3: Numbers also represent the $buffer\_size$

Model solving

In this part of the article, we will explain how the algorithms used in solving the problem were implemented.

To calculate the distance, we can use the breadth-first search algorithm, which calculates the shortest distances in edges in $O(n)$ time for each vertex in the graph, where n is the number of vertices. Considering that we need to calculate this for each vertex in the graph, the overall complexity becomes $O(n^2)$. This value also needs to be multiplied by n, as we calculate this value for each algorithm instance, resulting in a complexity of $O(n^3)$.

However, later on, we needed to calculate weighted distances where breadth-first search is no longer applicable. We assigned a specific weight to each vertex and defined the weighted distance between vertices as the sum of the weights of the vertices on the path. We needed to find the path with the minimum weight for each pair of vertices. To solve this problem, we can use the $Floyd-Warshall$ algorithm, which calculates these values for all pairs of vertices in the graph and works with a complexity of $O(n^3)$ for each vertex, resulting in a total complexity of $O(n^4)$, which is quite time-consuming and accounts for almost the entire program runtime.

Alternatively, we can use Dijkstra's algorithm, which calculates the minimum weighted distances from one vertex to all others in $O(nlogn)$ time, considering that the number of edges in the graph is of the order of $O(n)$. By running it for each vertex, we achieve a complexity of $O(n^2 \log n)$, which is faster than the $Floyd - Warshall$ algorithm. In the end, this approach works in $O(n^3 \log n)$ time. Thanks to this, we can recalculate the weights around 80 times by making some changes to the formula. However, note that Dijkstra's algorithm calculates distances if the edges of the graph have weights. To use it as we need, we assign a weight to each edge equal to the average value of the weights of the vertices it connects. We calculate the minimum distances. But note that the path weight is missing half of the weights of the initial and final vertices. We iterate through all pairs of vertices and add this value to the minimum weight value.

To transmit information between vertices, SwitchStatsInfo can be used in the $next\_round$ function. We will pass the vertex number and its current $buffer\_size$ to neighboring vertices so that the receiving vertices can determine whose $buffer\_size$ to update. It should be noted that at the time of selecting the vertex to which messages will be sent, the algorithm instance has up-to-date information about the $buffer\_sizes$ of its neighbors (excluding the fact that messages from other vertices have already been sent to this vertex in the current time slice).

Now let's explain the choice of the weight function. It is clear that we need to somehow define the capacity of the vertex. Since we will be using Dijkstra's algorithm to find the shortest distances, we need to define the formula in such a way that the best vertex has the smallest weight. It is also important to note that for Dijkstra's algorithm to work, the weights must be positive.

$$weight_i = \frac{bw\_in_i}{coeff_{level_i} * bw\_out_i + level_i} + 0.5$$

The first approximation for the vertex throughput is $\frac{bw\_in}{bw\_out}$. Indeed, the larger the bw_out, the better. It is also not very good if a vertex has a high bw_in compared to bw_out because in such a case, it will receive many vertices but send out few, and it will quickly become congested. Additionally, in order to minimize communication between level 1 and level 2 vertices, it is desirable for level 1 vertices to have a larger $buffer\_size$ to increase $success\_rate$, coefficients dependent on the vertex level were introduced in the denominator. The higher the level, the larger the coefficient. Moreover, the denominator also includes the $level$ itself to further account for this. However, the resulting formula does not consider the length of the path in edges, which affects latency. Therefore, we added 0.5 to this expression to account for it.

We also tried other functions that depended on $buffer\_size$ and recalculated weights during program execution, but it was not successful. This formula performed the best in practice.

Model Summarizing

Our model operates based on a greedy algorithm that selects vertices whose path weight to the $target\_node$ is lower than the vertex where the message is currently located, and sends the message to the vertex with the largest buffer_size. The weight takes into account the individual characteristics of each vertex, such as bw_in, bw_out, $level$, and primarily vertices go through the most optimal vertices. We also send messages in a specific order to improve latency and $success\_rate$.

Test Result Description

It is important to understand why such an algorithm works well. Let's state from the outset that we are operating under the assumption that vertices at the same level have roughly similar characteristics. Let's consider two scenarios. If the maximum access level is two, a single choice needs to be made. Therefore, choosing the first move optimally is equivalent to selecting the best algorithm. In the case of a maximum access level of three, the situation is no longer as straightforward. Assuming the existence of an "ideal algorithm"their initial moves will align. Our algorithm's second move will evenly distribute the load across third-level vertices, effectively distributing the load within the recipient node server. It is also worth noting that our algorithm ensures a good indication of successful "acceptance"from the sending server, as we maintain a sufficient amount of free space in the buffers of first-level vertices. As a result, we achieve the following metrics. We were able to achieve a 100% $successrate$ in three out of four tests, and 98.7 in the last one. Moreover, we guarantee that all messages have been sent and will arrive promptly. Specifically, the $average\ latency$ of our algorithm is 7. This is an excellent result, considering that almost all paths have a length of 5 (strictly speaking, 6, but we do not count the first edge).