An algorithm for geo-distributed and redundant storage in Garage

Mendes Oulamara
mendes@deuxfleurs.fr

Abstract

Garage

1 Introduction

Garage[1] is an open-source distributed object storage service tailored for self-hosting. It was designed by the Deuxfleurs association[2] to enable small structures (associations, collectives, small companies) to share storage resources to reliably self-host their data, possibly with old and non-reliable machines.

To achieve these reliability and availability goals, the data is broken into *partitions* and every partition is replicated over 3 different machines (that we call *nodes*). When the data is queried, a consensus algorithm allows to fetch it from one of the nodes. A *replication factor* of 3 ensures the best guarantees in the consensus algorithm [**?**], but this parameter can be different.

Moreover, if the nodes are spread over different *zones* (different houses, offices, cities...), we can ask the data to be replicated over nodes belonging to different zones, to improve the storage robustness against zone failure (such as power outage). To do so, we set a *redundancy parameter*, that is no more than the replication factor, and we ask that any partition is replicated over this number of zones at least.

In this work, we propose a repartition algorithm that, given the nodes specifications and the replication and redundancy parameters, computes an optimal assignation of partitions to nodes. We say that the assignation is optimal in the sense that it maximizes the size of the partitions, and hence the effective storage capacity of the system.

Moreover, when a former assignation exists, which is not optimal anymore due to nodes or zones updates, our algorithm computes a new optimal assignation that minimizes the amount of data to be transferred during the assignation update (the *transfer load*).

[1] `https://garagehq.deuxfleurs.fr/`
[2] `https://deuxfleurs.fr/`

We call the set of nodes cooperating to store the data a *cluster*, and a description of the nodes, zones and the assignation of partitions to nodes a *cluster layout*

1.1 Notations

Let k be some fixed parameter value, typically 8, that we call the "partition bits". Every object to be stored in the system is split into data blocks of fixed size. We compute a hash $h(\mathbf{b})$ of every such block $\mathbf{b}$, and we define the k last bits of this hash to be the partition number $p(\mathbf{b})$ of the block. This label can take $P = 2^k$ different values, and hence there are P different partitions. We denote $\mathbf{P}$ the set of partition labels (i.e. $\mathbf{P} = [\![1, P]\!]$).

We are given a set $\mathbf{N}$ of N nodes and a set $\mathbf{Z}$ of Z zones. Every node n has a non-negative storage capacity $c_n \geq 0$ and belongs to a zone $z_n \in \mathbf{Z}$. We are also given a replication parameter $\rho_{\mathbf{N}}$ and a redundancy parameter $\rho_{\mathbf{Z}}$ such that $1 \leq \rho_{\mathbf{Z}} \leq \rho_{\mathbf{N}}$ (typical values would be $\rho_N = 3$ and $\rho_Z = 2$).

Our goal is to compute an assignment $\alpha = (\alpha_p^1, \ldots, \alpha_p^{\rho_{\mathbf{N}}})_{p \in \mathbf{P}}$ such that every partition p is associated to $\rho_{\mathbf{N}}$ distinct nodes $\alpha_p^1, \ldots, \alpha_p^{\rho_{\mathbf{N}}} \in \mathbf{N}$ and these nodes belong to at least $\rho_{\mathbf{Z}}$ distinct zones. Among the possible assignations, we choose one that *maximizes* the effective storage capacity of the cluster. If the layout contained a previous assignment α', we *minimize* the amount of data to transfer during the layout update by making α as close as possible to α'. These maximization and minimization are described more formally in the following section.

1.2 Optimization parameters

To link the effective storage capacity of the cluster to partition assignment, we make the following assumption:

All partitions have the same size s. (H1)

This assumption is justified by the dispersion of the hashing function, when the number of partitions is small relative to the number of stored blocks.

Every node n wille store some number p_n of partitions (it is the number of partitions p such that n appears in the α_p). Hence the partitions stored by n (and hence all partitions by our assumption) have there size bounded by c_n/p_n. This remark leads us to define the optimal size that we will want to maximize:

$$s^* = \min_{n \in N} \frac{c_n}{p_n}. \tag{OPT}$$

When the capacities of the nodes are updated (this includes adding or removing a node), we want to update the assignment as well. However, transferring the data between nodes has a cost and we would like to limit the number of changes in the assignment. We make the following assumption:

Nodes updates happen rarely relatively to block operations. (H2)

This assumption justifies that when we compute the new assignment α, it is worth to optimize the partition size (OPT) first, and then, among the possible optimal solution, to try to minimize the number of partition transfers. More formally, we minimize the distance between two assignments defined by

$$d(\alpha, \alpha') := \#\{(n, p) \in \mathbf{N} \times \mathbf{P} \mid n \in \alpha_p \triangle \alpha'_p\} \tag{1}$$

where the symmetric difference $\alpha_p \triangle \alpha'_p$ denotes the nodes appearing in one of the assignations but not in both.

2 Computation of an optimal assignment

The algorithm that we propose takes as inputs the cluster layout parameters $\mathbf{N}$, $\mathbf{Z}$, $\mathbf{P}$, $(c_n)_{n\in\mathbf{N}}$, $\rho_{\mathbf{N}}$, $\rho_{\mathbf{Z}}$, that we defined in the introduction, together with the former assignation α' (if any). The computation of the new optimal assignation α^* is done in three successive steps that will be detailed in the following sections. The first step computes the largest partition size s^* that an assignation can achieve. The second step computes an optimal candidate assignment α that achieves s^* and a heuristic is used in the computation to make it hopefully close to α'. The third steps modifies α iteratively to reduces $d(\alpha, \alpha')$ and yields an assignation α^* achieving s^*, and minimizing $d(\cdot, \alpha')$ among such assignations.

We will explain in the next section how to represent an assignment α by a flow f on a weighted graph G to enable the use of flow and graph algorithms. The main function of the algorithm can be written as follows.

Algorithm

function COMPUTE LAYOUT($\mathbf{N}$, $\mathbf{Z}$, $\mathbf{P}$, $(c_n)_{n\in\mathbf{N}}$, $\rho_{\mathbf{N}}$, $\rho_{\mathbf{Z}}$, α')
    $s^* \leftarrow$ COMPUTE PARTITION SIZE($\mathbf{N}$, $\mathbf{Z}$, $\mathbf{P}$, $(c_n)_{n\in\mathbf{N}}$, $\rho_{\mathbf{N}}$, $\rho_{\mathbf{Z}}$)
    $G \leftarrow G(s^*)$
    $f \leftarrow$ COMPUTE CANDIDATE ASSIGNMENT(G, α')
    $f^* \leftarrow$ MINIMIZE TRANSFER LOAD(G, f, α')
    Build α^* from f^*
    return α^*
end function

Complexity

As we will see in the next sections, the worst case complexity of this algorithm is $O(P^2N^2)$. The minimization of transfer load is the most expensive step, and it can run with a timeout since it is only an optimization step. Without this step (or with a smart timeout), the worst cas complexity can be $O((PN)^{3/2} \log C)$ where C is the total storage capacity of the cluster.

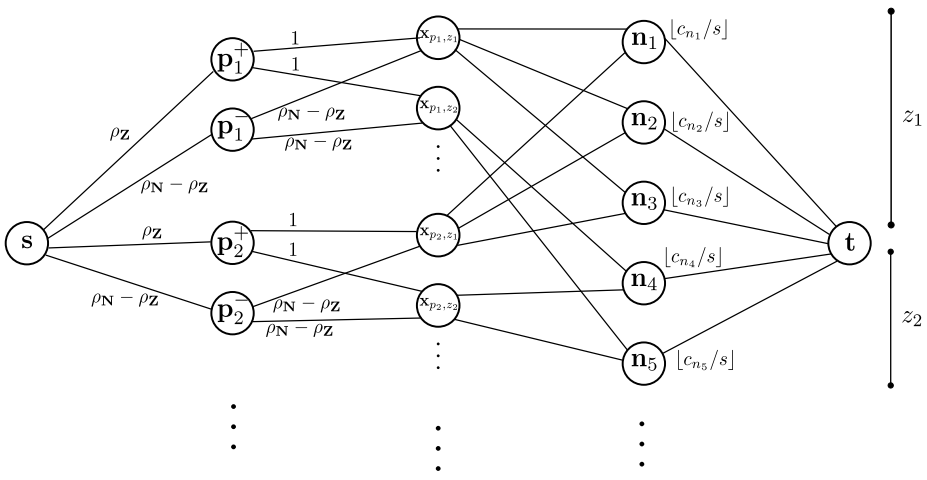

Figure 1: An example of graph $G(s)$. Arcs are oriented from left to right, and unlabeled arcs have capacity 1. In this example, nodes n_1, n_2, n_3 belong to zone z_1, and nodes n_4, n_5 belong to zone z_2.

2.1 Determination of the partition size s^*

We will represent an assignment α as a flow in a specific graph G. We will not compute the optimal partition size s^* a priori, but we will determine it by dichotomy, as the largest size s such that the maximal flow achievable on $G = G(s)$ has value $\rho_\mathbf{N} P$. We will assume that the capacities are given in a small enough unit (say, Megabytes), and we will determine s^* at the precision of the given unit.

Given some candidate size value s, we describe the oriented weighted graph $G = (V, E)$ with vertex set V arc set E (see Figure 1).

The set of vertices V contains the source $\mathbf{s}$, the sink $\mathbf{t}$, vertices $\mathbf{p}^+, \mathbf{p}^-$ for every partition p, vertices $\mathbf{x}_{p,z}$ for every partition p and zone z, and vertices $\mathbf{n}$ for every node n.

The set of arcs E contains:

- ($\mathbf{s}$,$\mathbf{p}^+$, $\rho_\mathbf{Z}$) for every partition p;
- ($\mathbf{s}$,$\mathbf{p}^-$, $\rho_\mathbf{N} - \rho_\mathbf{Z}$) for every partition p;
- ($\mathbf{p}^+$,$\mathbf{x}_{p,z}$, 1) for every partition p and zone z;
- ($\mathbf{p}^-$,$\mathbf{x}_{p,z}$, $\rho_\mathbf{N} - \rho_\mathbf{Z}$) for every partition p and zone z;
- ($\mathbf{x}_{p,z}$,$\mathbf{n}$, 1) for every partition p, zone z and node $n \in z$;
- ($\mathbf{n}$, $\mathbf{t}$, $\lfloor c_n/s \rfloor$) for every node n.

In the following complexity calculations, we will use the number of vertices and edges of G. Remark from now that $\#V = O(PZ)$ and $\#E = O(PN)$.

Proposition 1. *An assignment α is realizable with partition size s and the redundancy constraints $(\rho_{\mathbf{N}}, \rho_{\mathbf{Z}})$ if and only if there exists a maximal flow function f in G with total flow $\rho_{\mathbf{N}}P$, such that the arcs ($\mathbf{x}_{p,z}$,$\mathbf{n}$, 1) used are exactly those for which p is associated to n in α.*

Proof. Given such flow f, we can reconstruct a candidate α. In f, the flow passing through $\mathbf{p}^+$ and $\mathbf{p}^-$ is $\rho_{\mathbf{N}}$, and since the outgoing capacity of every $\mathbf{x}_{p,z}$ is 1, every partition is associated to $\rho_{\mathbf{N}}$ distinct nodes. The fraction $\rho_{\mathbf{Z}}$ of the flow passing through every $\mathbf{p}^+$ must be spread over as many distinct zones as every arc outgoing from $\mathbf{p}^+$ has capacity 1. So the reconstructed α verifies the redundancy constraints. For every node n, the flow between $\mathbf{n}$ and $\mathbf{t}$ corresponds to the number of partitions associated to n. By construction of f, this does not exceed $\lfloor c_n/s \rfloor$. We assumed that the partition size is s, hence this association does not exceed the storage capacity of the nodes.

In the other direction, given an assignment α, one can similarly check that the facts that α respects the redundancy constraints, and the storage capacities of the nodes, are necessary condition to construct a maximal flow function f. □

Implementation remark: In the flow algorithm, while exploring the graph, we explore the neighbours of every vertex in a random order to heuristically spread the associations between nodes and partitions.

Algorithm

With this result mind, we can describe the first step of our algorithm. All divisions are supposed to be integer divisions.

function COMPUTE PARTITION SIZE($\mathbf{N}$, $\mathbf{Z}$, $\mathbf{P}$, $(c_n)_{n\in\mathbf{N}}$, $\rho_{\mathbf{N}}$, $\rho_{\mathbf{Z}}$)
    Build the graph $G = G(s = 1)$
    $f \leftarrow$ MAXIMAL FLOW(G)
    if f.totalflow $< \rho_{\mathbf{N}}P$ **then**
        return Error: capacities too small or constraints too strong.
    end if
    $s^- \leftarrow 1$
    $s^+ \leftarrow 1 + \frac{1}{\rho_{\mathbf{N}}} \sum_{n\in\mathbf{N}} c_n$
    while $s^- + 1 < s^+$ **do**
        Build the graph $G = G(s = (s^- + s^+)/2)$
        $f \leftarrow$ MAXIMAL FLOW(G)
        if f.totalflow $< \rho_{\mathbf{N}}P$ **then**
            $s^+ \leftarrow (s^- + s^+)/2$
        else
            $s^- \leftarrow (s^- + s^+)/2$
        end if
    end while
    return s^-

19: **end function**

Complexity

To compute the maximal flow, we use Dinic's algorithm. Its complexity on general graphs is $O(\#V^2\#E)$, but on graphs with edge capacity bounded by a constant, it turns out to be $O(\#E^{3/2})$. The graph G does not fall in this case since the capacities of the arcs incoming to $\mathbf{t}$ are far from bounded. However, the proof of this complexity function works readily for graphs where we only ask the edges *not* incoming to the sink $\mathbf{t}$ to have their capacities bounded by a constant. One can find the proof of this claim in [1, Section 2]. The dichotomy adds a logarithmic factor $\log(C)$ where $C = \sum_{n\in\mathbf{N}} c_n$ is the total capacity of the cluster. The total complexity of this first function is hence $O(\#E^{3/2}\log C) = O\big((PN)^{3/2}\log C\big)$.

Metrics

We can display the discrepancy between the computed s^* and the best size we could have hoped for the given total capacity, that is $C/\rho_{\mathbf{N}}$.

2.2 Computation of a candidate assignment

Now that we have the optimal partition size s^*, to compute a candidate assignment it would be enough to compute a maximal flow function f on $G(s^*)$. This is what we do if there is no former assignation α'.

If there is some α', we add a step that will heuristically help to obtain a candidate α closer to α'. We fist compute a flow function $\tilde{f}$ that uses only the partition-to-node associations appearing in α'. Most likely, $\tilde{f}$ will not be a maximal flow of $G(s^*)$. In Dinic's algorithm, we can start from a non maximal flow function and then discover improving paths. This is what we do by starting from $\tilde{f}$. The hope[3] is that the final flow function f will tend to keep the associations appearing in $\tilde{f}$.

More formally, we construct the graph $G_{|\alpha'}$ from G by removing all the arcs $(\mathbf{x}_{p,z}, \mathbf{n}, 1)$ where p is not associated to n in α'. We compute a maximal flow function $\tilde{f}$ in $G_{|\alpha'}$. The flow $\tilde{f}$ is also a valid (most likely non maximal) flow function on G. We compute a maximal flow function f on G by starting Dinic's algorithm on $\tilde{f}$.

Algorithm

1: **function** Compute Candidate Assignment(G, α')
2: Build the graph $G_{|\alpha'}$
3: $\tilde{f} \leftarrow$ Maximal flow($G_{|\alpha'}$)
4: $f \leftarrow$ Maximal flow from flow(G, $\tilde{f}$)

[3]This is only a hope, because one can find examples where the construction of f from $\tilde{f}$ produces an assignment α that is not as close as possible to α'.

5: **return** f
6: **end function**

Remark: The function "Maximal flow" can be just seen as the function "Maximal flow from flow" called with the zero flow function as starting flow.

Complexity

With the considerations of the last section, we have the complexity of the Dinic's algorithm $O(\#E^{3/2}) = O((PN)^{3/2})$.

Metrics

We can display the flow value of $\tilde{f}$, which is an upper bound of the distance between α and α'. It might be more a Debug level display than Info.

2.3 Minimization of the transfer load

Now that we have a candidate flow function f, we want to modify it to make its corresponding assignation α as close as possible to α'. Denote by f' the maximal flow corresponding to α', and let $d(f, \alpha') = d(f, f') := d(\alpha, \alpha')$[4]. We want to build a sequence $f = f_0, f_1, f_2 \dots$ of maximal flows such that $d(f_i, \alpha')$ decreases as i increases. The distance being a non-negative integer, this sequence of flow functions must be finite. We now explain how to find some improving f_{i+1} from f_i.

For any maximal flow f in G, we define the oriented weighted graph $G_f = (V, E_f)$ as follows. The vertices of G_f are the same as the vertices of G. E_f contains the arc (v_1, v_2, w) between vertices $v_1, v_2 \in V$ with weight w if and only if the arc (v_1, v_2) is not saturated in f (i.e. $c(v_1, v_2) - f(v_1, v_2) \geq 1$, we also consider reversed arcs). The weight w is:

- -1 if (v_1, v_2) is of type $(\mathbf{x}_{p,z}, \mathbf{n})$ or $(\mathbf{x}_{p,z}, \mathbf{n})$ and is saturated in only one of the two flows f, f';
- $+1$ if (v_1, v_2) is of type $(\mathbf{x}_{p,z}, \mathbf{n})$ or $(\mathbf{x}_{p,z}, \mathbf{n})$ and is saturated in either both or none of the two flows f, f';
- 0 otherwise.

If γ is a simple cycle of arcs in G_f, we define its weight $w(\gamma)$ as the sum of the weights of its arcs. We can add $+1$ to the value of f on the arcs of γ, and by construction of G_f and the fact that γ is a cycle, the function that we get is still a valid flow function on G, it is maximal as it has the same flow value as f. We denote this new function $f + \gamma$.

Proposition 2. *Given a maximal flow f and a simple cycle γ in G_f, we have $d(f + \gamma, f') - d(f, f') = w(\gamma)$.*

[4]It is the number of arcs of type $(\mathbf{x}_{p,z}, \mathbf{n})$ saturated in one flow and not in the other.

Proof. Let X be the set of arcs of type $(\mathbf{x}_{p,z}, \mathbf{n})$. Then we can express $d(f, f')$ as

$$\begin{aligned} d(f, f') &= \#\{e \in X \mid f(e) \neq f'(e)\} = \sum_{e \in X} 1_{f(e) \neq f'(e)} \\ &= \frac{1}{2}\big(\#X + \sum_{e \in X} 1_{f(e) \neq f'(e)} - 1_{f(e) = f'(e)}\big). \end{aligned}$$

We can express the cycle weight as

$$w(\gamma) = \sum_{e \in X, e \in \gamma} -1_{f(e) \neq f'(e)} + 1_{f(e) = f'(e)}.$$

Remark that since we passed on unit of flow in γ to construct $f + \gamma$, we have for any $e \in X$, $f(e) = f'(e)$ if and only if $(f + \gamma)(e) \neq f'(e)$. Hence

$$\begin{aligned} w(\gamma) &= \frac{1}{2}(w(\gamma) + w(\gamma)) \\ &= \frac{1}{2}\Big(\sum_{e \in X, e \in \gamma} -1_{f(e) \neq f'(e)} + 1_{f(e) = f'(e)} \\ &\qquad + \sum_{e \in X, e \in \gamma} 1_{(f+\gamma)(e) \neq f'(e)} + 1_{(f+\gamma)(e) = f'(e)} \Big). \end{aligned}$$

Plugging this in the previous equation, we find that

$$d(f, f') + w(\gamma) = d(f + \gamma, f').$$

□

This result suggests that given some flow f_i, we just need to find a negative cycle γ in G_{f_i} to construct f_{i+1} as $f_i + \gamma$. The following proposition ensures that this greedy strategy reaches an optimal flow.

Proposition 3. *For any maximal flow f, G_f contains a negative cycle if and only if there exists a maximal flow f^* in G such that $d(f^*, f') < d(f, f')$.*

Proof. Suppose that there is such flow f^*. Define the oriented multigraph $M_{f,f^*} = (V, E_M)$ with the same vertex set V as in G, and for every $v_1, v_2 \in V$, E_M contains $(f^*(v_1, v_2) - f(v_1, v_2))_+$ copies of the arc (v_1, v_2). For every vertex v, its total degree (meaning its outer degree minus its inner degree) is equal to

$$\begin{aligned} \deg v &= \sum_{u \in V} (f^*(v, u) - f(v, u))_+ - \sum_{u \in V} (f^*(u, v) - f(u, v))_+ \\ &= \sum_{u \in V} f^*(v, u) - f(v, u) = \sum_{u \in V} f^*(v, u) - \sum_{u \in V} f(v, u). \end{aligned}$$

The last two sums are zero for any inner vertex since f, f^* are flows, and they are equal on the source and sink since the two flows are both maximal and have hence the same value. Thus, $\deg v = 0$ for every vertex v.

This implies that the multigraph M_{f,f^*} is the union of disjoint simple cycles. f can be transformed into f^* by pushing a mass 1 along all these cycles in any order. Since $d(f^*, f') < d(f, f')$, there must exists one of these simple cycles γ with $d(f + \gamma, f') < d(f, f')$. Finally, since we can push a mass in f along γ, it must appear in G_f. Hence γ is a cycle of G_f with negative weight. □

In the next section we describe the corresponding algorithm. Instead of discovering only one cycle, we are allowed to discover a set Γ of disjoint negative cycles.

Algorithm

```
1: function MINIMIZE TRANSFER LOAD(G, f, α')
2:     Build the graph G_f
3:     Γ ← DETECT NEGATIVE CYCLES(G_f)
4:     while Γ ≠ ∅ do
5:         for all γ ∈ Γ do
6:             f ← f + γ
7:         end for
8:         Update G_f
9:         Γ ← DETECT NEGATIVE CYCLES(G_f)
10:    end while
11:    return f
12: end function
```

Complexity

The distance $d(f, f')$ is bounded by the maximal number of differences in the associated assignment. If these assignment are totally disjoint, this distance is $2\rho_N P$. At every iteration of the While loop, the distance decreases, so there is at most $O(\rho_N P) = O(P)$ iterations.

The detection of negative cycle is done with the Bellman-Ford algorithm, whose complexity should normally be $O(\#E\#V)$. In our case, it amounts to $O(P^2ZN)$. Multiplied by the complexity of the outer loop, it amounts to $O(P^3ZN)$ which is a lot when the number of partitions and nodes starts to be large. To avoid that, we adapt the Bellman-Ford algorithm.

The Bellman-Ford algorithm runs $\#V$ iterations of an outer loop, and an inner loop over E. The idea is to compute the shortest paths from a source vertex v to all other vertices. After k iterations of the outer loop, the algorithm has computed all shortest path of length at most k. All simple paths have length at most $\#V - 1$, so if there is an update in the last iteration of the loop, it means that there is a negative cycle in the graph. The observation that will enable us to improve the complexity is the following:

Proposition 4. *In the graph G_f (and G), all simple paths have a length at most $4N$.*

Proof. Since f is a maximal flow, there is no outgoing edge from $\mathbf{s}$ in G_f. One can thus check than any simple path of length 4 must contain at least two node of type $\mathbf{n}$. Hence on a path, at most 4 arcs separate two successive nodes of type $\mathbf{n}$. □

Thus, in the absence of negative cycles, shortest paths in G_f have length at most $4N$. So we can do only $4N+1$ iterations of the outer loop in the Bellman-Ford algorithm. This makes the complexity of the detection of one set of cycle to be $O(N\#E) = O(N^2P)$.

With this improvement, the complexity of the whole algorithm is, in the worst case, $O(N^2P^2)$. However, since we detect several cycles at once and we start with a flow that might be close to the previous one, the number of iterations of the outer loop might be smaller in practice.

Metrics

We can display the node and zone utilization ratio, by dividing the flow passing through them divided by their outgoing capacity. In particular, we can pinpoint saturated nodes and zones (i.e. used at their full potential).

We can display the distance to the previous assignment, and the number of partition transfers.

References

[1] S. Even and R. E. Tarjan, "Network flow and testing graph connectivity," *SIAM journal on computing*, vol. 4, no. 4, pp. 507–518, 1975.