

Functional Requirements

Erlang Router Team: Davis, George, Graham, Paul

Last Edit: April 16, 2015

Abstract

Abstract forthcoming

1 Introduction

The goal of this project is to develop a load-balancing system in an IOT (Internet of Things) environment. A multitude of devices with varying processing capabilities will be connected to a cluster of interconnected servers in a data center (a "cloud") via persistent TCP connections. These devices are known as "clients".

Clients are each assigned to a single organization and will communicate with other clients in that group. They will send messages to one another. It is assumed that these messages are more expensive the more they propagate between servers.

The following sections establish the problems we aim to address and the mathematical representations used to analyze them.

1.1 Problems/Tasks

Re-Allocation Need an efficient method to consistently re-allocate clients so that clients in the same group tend to move congregate on the same server(s).

Health-checks Need to be able to detect server health in terms of client load. This may be derived from a server capacity ratio, or it may be more specific to the server in question.

Free Server Once an unhealthy server is discovered, it will need to have its clients redistributed efficiently.

Crash Contingency How server crashes are handled.

control Architecture Currently we plan to implement the cluster managing through a master server, we may need contingencies if this crashes.

1.2 Servers, Groups, and their Clients

As mentioned previously, the system will operate upon a network (or cluster) of interconnected servers. Clients connected to the cluster of servers in real time, and each is tethered to a server. Clients that communicate with each other are subdivided into 'groups'.

We denote the set of servers $S = [s_0, s_1, \dots, s_n]$, with $|S|$ representing the total number of servers.

We denote the set of groups $G = [g_0, g_1, \dots, g_n]$, with $|G|$ representing the total number of groups.

We then denote client positions c in terms of servers and groups, such that $c(s_0, g_0)$ refers to clients on server s_0 that are also in group g_0 . Or the set $C_{g_0, s_0} = \{ client : client \in s_0 \wedge client \in g_0 \}$

A column representation SxG of the established elements parallels the form of the corresponding data structure in our code:

$$\begin{bmatrix} c_{1,1} \\ c_{1,2} \\ c_{1,3} \\ c_{1,4} \\ c_{1,5} \\ \dots \end{bmatrix} \begin{bmatrix} c_{2,1} \\ c_{2,2} \\ c_{2,3} \\ c_{2,4} \\ c_{2,5} \\ \dots \end{bmatrix} \begin{bmatrix} c_{3,1} \\ c_{3,2} \\ c_{3,3} \\ c_{3,4} \\ c_{3,5} \\ \dots \end{bmatrix} \begin{bmatrix} c_{4,1} \\ c_{4,2} \\ c_{4,3} \\ c_{4,4} \\ c_{4,5} \\ \dots \end{bmatrix} \begin{bmatrix} c_{5,1} \\ c_{5,2} \\ c_{5,3} \\ c_{5,4} \\ c_{5,5} \\ \dots \end{bmatrix} \begin{bmatrix} \dots \\ \dots \\ \dots \\ \dots \\ \dots \\ \dots \end{bmatrix}$$

SxG , as a matrix:

$$\begin{bmatrix} c_{1,1} & c_{2,1} & c_{3,1} & c_{4,1} & c_{5,1} & \dots \\ c_{1,2} & c_{2,2} & c_{3,2} & c_{4,2} & c_{5,2} & \dots \\ c_{1,3} & c_{2,3} & c_{3,3} & c_{4,3} & c_{5,3} & \dots \\ c_{1,4} & c_{2,4} & c_{3,4} & c_{4,4} & c_{5,4} & \dots \\ c_{1,5} & c_{2,5} & c_{3,5} & c_{4,5} & c_{5,5} & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \end{bmatrix}$$

1.3 Server Capacity

In reality, servers maintain up to a finite number of client connections. We call this limit capacity.

Capacity is a function from servers on to \mathbb{N} [$cap: S \rightarrow \mathbb{N}$] such that the capacity of server n will be denoted as $cap(n)$. This gives us an absolute maximum of client connections that a server cannot exceed.

Then, $|C_n|$ is the number of clients in server n , and $|C_n| \leq cap(n)$ is an implicit rule.

2 Methods of Analysis

We detail our methods of analyzing and quantifying the stated problems/tasks.

2.1 Fragmentation Intro

We know that server to server communication is expensive relative to communication within a server. By this logic, all the clients in a group should be confined to a single server whenever possible. Taken further, the minimization of group splitting instances between servers is ideal. We use these assumptions and the following logic to inform our Re-allocation solution.

The Fragmentation Principle operates on this logic, that fewer instances of group members present on different servers is preferable, as it would minimize the instances of clients communicating across different servers. It is worth mentioning that we developed two ways to understand fragmentation, server-to-servers and group-to-servers. It is advisable to skip the discussion of the former, as we have abandoned it for the time being.

We will use the adjectives of magnitude and breadth to describe fragmentation. Magnitude concerns the volume of clients involved, whereas breadth evokes the degree to which they are dispersed.

2.2 Fragmentation: Server-to-Servers

Server-to-servers fragmentation ignores the magnitude of fragmentation between two servers, and instead examines the breadth across servers in the whole. Server A and server B have some of clients in conversation, or none. This can be understood as a binary classification, either two servers are connected (by way of grouped clients), or they are not. Each server's respective fragmentation indicates how many servers it is connected to.

The server-to-servers fragmentation of server s_n is simply the number of servers that it is connected to by way of common groups. It can be modeled as a summation of binary variable i across all other servers j , i corresponding to 1 if the servers s_n and s_j have any common groups, 0 if none.

$$server\ frag(n) = \sum_{j \in S; j \neq n} i = 1 \text{ if } |G_n \cap G_j| > 0, \text{ else } i = 0$$

Calculating the server-to-servers fragmentation of every server provides a corresponding sequence of values that depicts each server's relation to other servers in a broad sense. It enables one to determine the degree to which servers communicate by way of group association.

$$\begin{aligned} Server1 : & F_1 \\ Server2 : & F_2 \\ & \dots \end{aligned}$$

F_1 represents the number of servers Server 1 is connected to due to some common group(s).

Server-fragmentation can be interpreted as a mapping of vectors. We represent each connection between two servers as a vector $[n, j]$. It is obvious that if $n = j$, we have a point, the same position mapped to itself. Furthermore, the total number of vectors is equal to half of the total server-fragmentation, the

sum of $serverfrag(n)$ for every server n .

2.3 Fragmentation: Group-to-Servers

Group-to-servers fragmentation examines the degree to which each group is fragmented. It provides a list of elements that correspond to the groups, with values indicating their degree of fragmentation. The group-to-servers fragmentation of a group g_n is the number of servers that contain clients of g_n .

Group1 : F_1

Group2 : F_2

...

F_1 represents the number of servers Group 1 is split among.

3 Algorithmic Solutions

Here we detail the Algorithm(s) we developed. In order to analyze the efficiency of these methods, we need to consider the number of client re-allocations accordingly, as transferring clients will require server interaction at the minimum. Furthermore, the calculations used to orient optimization may delay in cases involving thousands of clients.

3.1 Re-Allocation: Naively Greedy

For this approach, we work greedily to expunge all group fragmentation from each server, one server at a time.

We compare two servers (A and B), counting the number of clients in a groups split between the two (there are other servers as well- C,D etc). This greedy algorithm deals with four basic situations:

- If server A has more members of common groups than server B, we move the clients from B to A. However, if A does not have the capacity for all of those clients, we search for other instances of that group on other servers. If no match is found, the clients remain on B.
- If server B has more members in a particular group than server A, we move the clients from A to B with the same protocol for capacity as above.
- If they have the same number of clients from a particular group, we move all clients to whichever server has the most capacity available, if possible. Otherwise nothing happens.
- If either server doesn't have any members from a group, no action is required, as no fragmentation is present.

We repeat this operation for every server B that isn't A to eliminate the all fragmentation involving A. To complete this greedy method, we would do that total process for every server, except the last one (we can assume it is fragmentation free if every other server is by the definition thereof)

As a possible addendum: If no server can hold them all, we divide the clients in half and repeat the search for space to accommodate each half. This would continue recursively until every client is allocated. I am thinking of rescinding this process for the greedy approach, as it would reduce fragmentation but it may not be a suitable optimization.

This requires $(n-1)*|G|$ comparisons per server, if n and $|G|$ are the total number of servers and groups, respectively. That would be $(n-1)^2*|G|$ comparisons for the complete implementation.

We will construct and average of client re-allocations this method evokes as we test it with out simulator, and this data will serve as a reference point for improved algorithms.

3.2 Re-Allocation Worst-First

This approach orients itself through group-to-servers fragmentation. It iterates through the groups, rather than the servers, and attempts to corral their disparate clients into a single server. It prioritizes the groups of highest (or worst) fragmentation.

The sequence of operations that occur within a single 'pass' of first-worst:

- First, the group-to-servers fragmentation is calculated and stored as a list. It indicates how badly each group is split, it would be ideal for each group to have a value of 1, meaning they reside on only one server.
- The group with the highest value is chosen, the algorithm will attempt to resolve it to one server, or reduce it to occupy fewer servers.
- The total number of clients in the group is determined.
- The algorithm checks to see if most empty server can hold all of these clients.
- If this is not possible, it tries the two most empty servers. It continues this process until the number of servers sought exceeds the original amount, as we don't want to divide the group further.
- Regardless of whether this attempt to reduce/resolve succeeded, the algorithm chooses the next worst group and continues until it has attempted each group that has a fragmentation value greater than 1.

4 Experimental Data

This is where we will examine and compare the results from different implementations, and hopefully make a case for a particular method.

5 Sources

Honestly not sure what these would be *smugly clears throat*. I'm just going to jot down what comes to mind for later consideration.

- The wonderful paper Prof. Czaparka provided us? I'm not sure how much of that remains, but it definitely help us get started.
- Basic source material on bin packing?
- Basic definition of IOT if that seems salient?
- Source material on other visual aids (bi-variate graph)?
- can't think of anything else immediately