Functional Requirements

Erlang Router Team: Davis, George, Graham, Joseph, Paul

Last Edit: April 29, 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

Our software operates upon a network (or cluster) of interconnected servers. Clients connect to the cluster in real time, each is attached to a particular server. Clients that communicate with each other are subdivided into 'groups'.

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

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

We then denote client positions c in terms of servers and groups, such that $c(s_1, g_1)$ refers to clients on server s_1 that are also in group g_1. Or the set $c_{s_1,g_1} = \{ \textit{ client} : \textit{client} \in s_1 \wedge \textit{client} \in g_1 \}$

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

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

SxG, as a matrix:

$$\begin{array}{c c} & \begin{array}{cccccc} s_1 & s_2 & s_3 & s_4 & s_5 & ... \end{array} \\ \begin{array}{c} g_1 \\ g_2 \\ g_3 \\ g_4 \\ g_5 \\ ... \end{array} & \begin{pmatrix} c_{1,1} & c_{2,1} & c_{3,1} & c_{4,1} & c_{5,1} & ... \\ c_{1,2} & c_{2,2} & c_{3,2} & c_{4,2} & c_{5,2} & ... \\ c_{1,3} & c_{2,3} & c_{3,3} & c_{4,3} & c_{5,3} & ... \\ c_{1,4} & c_{2,4} & c_{3,4} & c_{4,4} & c_{5,4} & ... \\ c_{1,5} & c_{2,5} & c_{3,5} & c_{4,5} & c_{5,5} & ... \\ ... & ... & ... & ... & ... & ... \end{pmatrix} \end{array}$$

By this example it should be obvious that selection of a server s_n offers a series of c values indicating the clients per group within s_n.

$s_1 = [c_{1,1}, c_{1,2}, c_{1,3}, c_{1,4}, c_{1,5}, ...]$

We may examine a group across servers in the same manner.

$g_1 = [c_{1,1}, c_{2,1}, c_{3,1}, c_{4,1}, c_{5,1}, ...]$

Since this paper primarily discusses techniques based upon analysis, it does not concern itself with the individual designations of each client, merely the number of clients in each location.

As such, s_1 can be assumed to represent $[|c_{1,1}|, |c_{1,2}|, |c_{1,3}|, |c_{1,4}|, |c_{1,5}|, ...]$ wherever it appears in this paper, and the same follows for groups and any other such designations.

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 a server on to $\mathbb{N}$ [cap: $S \rightarrow \mathbb{N}$]. The capacity of server s_1 is denoted $cap(s_1)$. This gives us an absolute maximum of client connections that a server may never exceed.

For a server s_1, $TotalClients(s_1) = \sum_{g_i \in G} c_{s_1,g_i}$

We can subtract this from $cap(s_1)$ in order to determine how many additional clients s_1 may accommodate. This will be imperative for allocation.

Similarly, we can total the clients in a group across as servers:

$TotalClients(g_1) = \sum_{s_i \in S} c_{s_i,g_1}$

2 Methods of Analysis

We detail our methods of analyzing and quantifying 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.

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.

$Server1: \quad F_1$
$Server2: \quad F_2$
...
F_1 represents the number of servers Server 1 is connected to due to some common group(s).

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.
$Group1: \quad F_1$
$Group2: \quad F_2$
...
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 intend to run this basic solution through our simulator to generate average run-times under a variety of circumstances, this data will serve as a reference point for improved algorithms.

3.2 Re-Allocation Worst-First

The Worst-First 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:

1. First, the group-to-servers fragmentation is calculated and stored as a list. It indicates how badly each group is split, ideally each group should have a value of 1, meaning they reside on only one server.

2. The algorithm will proceed through every group from order of highest fragmentation to lowest.

3. The total number of clients in the selected group is determined.

4. The algorithm checks to see if most empty server can hold all of these clients.

5. 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.

6. Regardless of whether this attempt to reduce/resolve group fragmentation succeeded, the algorithm chooses the next worst group and continues until it has attempted each group that has a fragmentation value greater than 1. This concludes a single pass of the Worst-First algorithm.

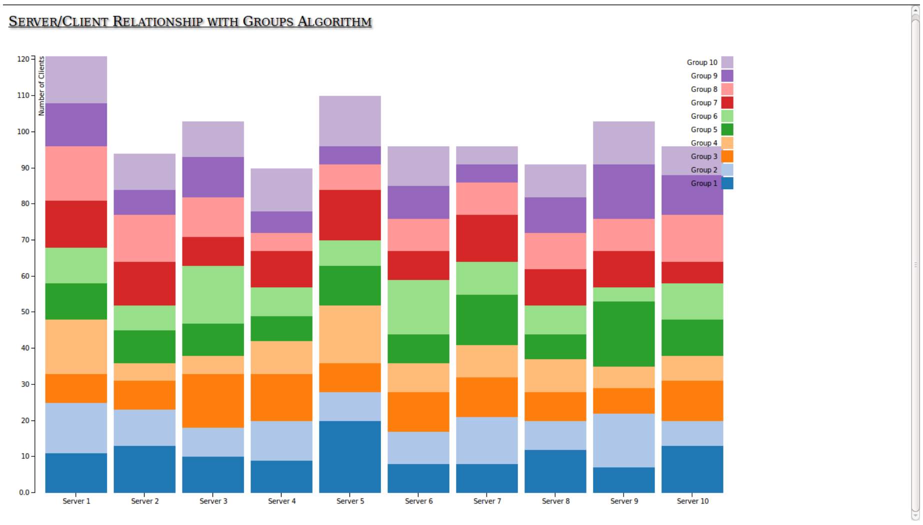

Figure 1: The starting allocation in the 1000 client simulation.

4 Results

For a basic demonstration, the simulator ran the algorithmic solutions with a 10 x 10 Server-Group configuration. We will variate these parameters to suit real-world conditions once we are privy to such information.

4.1 1000 Client 200 Capacity test

Clients: 1000
Servers: 10
Groups: 10
Capacity: 200

Initial GTS: [10,10,10,10,10,10,10,10,10,10]

Greedy allocation result: [4,3,3,3,1,2,1,1,2,2]
Greedy run-time: 11827 microseconds

Worst-First allocation result: [1,1,1,1,1,1,1,1,1,1]
Worst-First run-time: 9296 microseconds

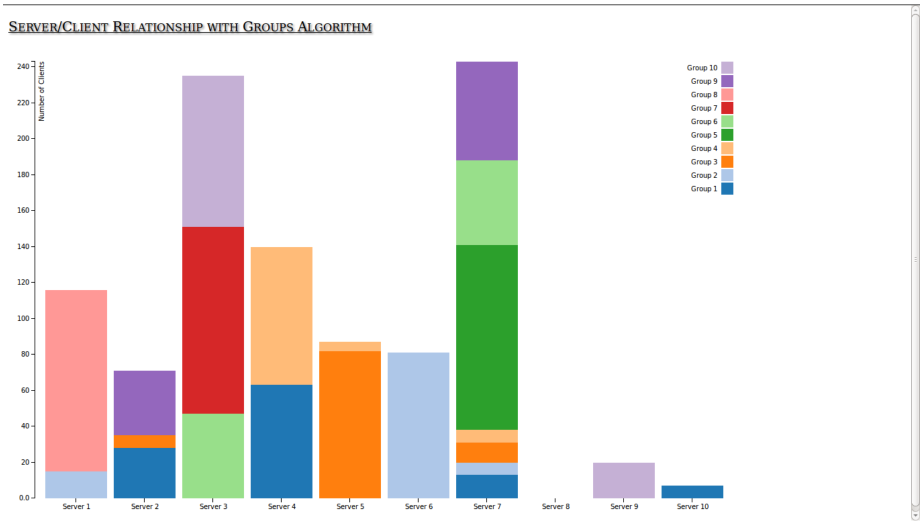

Figure 2: The resulting Greedy allocation in the 1000 client simulation.

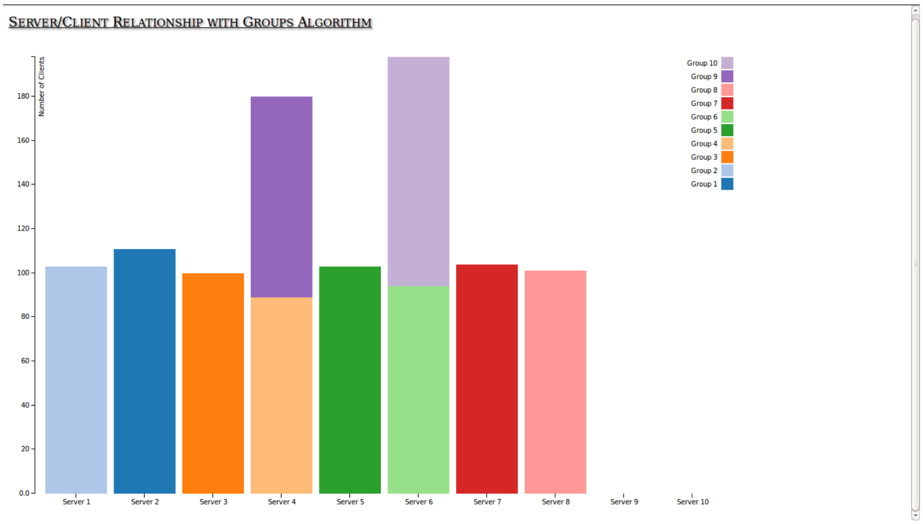

Figure 3: The resulting Worst-First allocation in the 1000 client simulation.

4.2 10000 Client 2000 Capacity test

Clients: 10000
Servers: 10
Groups: 10
Capacity: 2000

Greedy allocation result: [3,2,1,1,1,1,1,1,2,2]
Greedy run-time: 90963 microseconds

Worst-First allocation result: [1,1,1,1,1,1,1,1,1,1]
Worst-First run-time: 75523 microseconds

5 Working Conclusion

The advantage of the Worst-First Algorithm should be clear from the above results. It is faster than the Greedy attempt and forms a superior allocation. It should therefore manage a cluster with greater efficiency, though that remains to be demonstrated. Additionally, our solution development is far from finished. Both solutions will need to be optimized for better experimental accuracy, and expanded to handle a broader degree of situations. We will need to augment the simulator accordingly.