CloudPrime

Sérgio Mendes
Instituto Superior Técnico
MEIC-A
sergiosmendes@tecnico.ulisboa.pt

Carlos Faria
Instituto Superior Técnico
MEIC-A
carlosfaria@tecnico.ulisboa.pt

Diogo Abreu
Instituto Superior Técnico
MEIC-A
diogo.perestrelo.abreu@tecnico.ulisboa.pt

1. Introduction

In this report we present our implementation of an elastic cluster of web servers that receive a post request with a single parameter, a semiprime that is going to be factorized and replied with the request back to whoever made the request.

Our main goals were to make sure that the workers were not overwhelmed with requests, therefore a good scheduling and scaling algorithm were necessary to ensure that and provide a reasonable client experience. Our other main goal was not to lose any request, therefore making the cluster fault-tolerant. We start by describing the important components of our cluster in Section 2. In Section 3 we present the metrics that we gather to estimate the cost of a request like explained in Section 5. In Section 4 we explain what other data structures we use besides the Metrics Storage System. In Section 6 we define our scheduling algorithm which is responsible for keep the cluster well balanced while in Section 7 we present our scaling algorithm that has the role of dealing with the creation and removal of Workers. Finally in Section 8 we show how our cluster is resilient to faults. In Section 9 we conclude and in Section 10 we have an Appendix that has illustrations that help our explanations.

2. Architecture

In the Appendix Section, 10.1, we present our overall system architecture and dataflow. In the following subsections we will describe what operations are performed in each of the system components, Load Balancer, Workers and Metric Storage System.

2.1 Load Balancer

The Load Balancer is the core of our system. It's this component that is responsible for assuring our main goals described in 1.In the following subsubsections we will describe what scaling and scheduling operations are performed by the Load Balancer.

2.1.1 Auto Scaling

The most important operating of scaling performed by the Load Balancer is to decide whether to create or remove instances, this will be explained in more detail in Section 7, in this section we will describe the basic, but necessary, scaling operations performed by the Load Balancer.

The Load Balancer has a list of all currently active workers, and keeps track of the following information about each instance that will be helpful for the Task Scheduling Algorithm (Section 6) and Auto-Scaling algorithm(Section 7):

- Basic instance information: instance ID, instance IP, etc;
- Current CPU usage;
- Number of requests being processed;
- Number of Max rank requests being processed (explained in Section 6);
- Number of Intermediate rank requests being processed (explained in Section 6);

Workers might fail, and we need to detect those fails in order to compensate that fail, transferring requests from the failed worker to others or starting another instance to compensate for the crash (Section 8), for this purpose, we have threads performing periodically health checks to the workers to check if they have crashed or not.

Each 30 seconds, we ping the worker and wait 3 seconds for the ping. This was carefully studied, we performed several tests when a worker was under a heavy load to check how long it took to respond to a ping and it took on average 1003 ms. This study was important to decide how long to wait for the ping because if we waited for a too short period, the instance could be detected has crashed when it actually wasn't and it was just delayed due to the heavy load. We chose the value 3 because we decided to add another 2 seconds for possible network delays or other types of delays that might occur.

In order to avoid false positives, that is, declaring an instance has crashed when its not, we decided to include an unhealthy threshold. Each time an instance fails to respond to a ping, we increment the unhealthy threshold,if it eventually responds, we put the unhealthy threshold back to 0, if it reaches the value 3 then we declare this work has crashed. This value is low due to the importance of detecting a failed worker because at each moment a worker is down, we are losing overall performance of the cluster. For the same reason, we ping the workers every 30 seconds.

Due to the importance of knowing the current CPU usage of each worker, we've enabled Detailed Monitoring for our workers and we have specific threads to query the CloudWatch monitor to get CPU usage of each worker. Since Detailed Monitoring updates the CPU usage every minute, these threads query the CloudWatch every 1:15 minutes to update the CPU usage of each worker.

2.1.2 Task Scheduling

In Section 6 we will explain in more detail how we perform the scheduling, here we will just mention the basic flow of a request when the Load Balancer receives it.

Each request that the Load Balancer receives is treated by a different thread we then run it through our scheduling algorithm which is responsible for assigning a worker to perform the factorization of the request. The Load Balancer then forwards the request to this worker and after the worker is done, the Load Balancer receives the result and sends it back to the user. In the following subsection we will demonstrate how Workers deal with requests and how they gather metrics.

2.2 Workers

The workers are the entities responsible for calculating the factorization of the request they received to factorize from the Load Balancer, collect metrics from that request, send metrics to the Metric Storage System and send the result from the factorization back to the Load Balancer The Workers are multi-threaded, each request they receive to factorize, is treated by a different thread. We start with 2 Workers so we can perform some load balancing right at the beginning.

2.2.1 Gathering Metrics

While they are calculating the factorization, at the same time, they are collecting metrics and when they are done, they calculate the Rank of the request(explained in Section 5) and they send both the metrics and the Rank to the Metrics Storage System. However, we also retrieve intermediate metrics every 100400724 basic blocks that are executed, we performed some tests and that amount of blocks are roughly executed in between 1.30 minutes and 2 minutes depending on the workload of the Worker so we think it's a good time window to gather intermediate metrics. These metrics are useful because they are normally associated with long operations, so if another request like this appears and we have intermediate metrics about it, we know it's a long operation and we can schedule it accordingly.

2.2.2 Optimizations

When gathering intermediate metrics, we have to perform some operations and these operations will introduce additional overhead which will influence the metrics retrieved, to avoid this situation, we create a thread specifically for dealing with sending the intermediate threads back to the Metrics Storage System.

When gathering intermediate metrics we have to be careful because in parallel, in another thread or even in another instance, may be ocurring the calculation of the same request and this can cause conflicts. For example lets imagine Thread 1 starts factorizing a big number and a while after, Thread 2 starts factorizing the same number as Thread 1. Thread 1 will gather intermediate metrics twice, and then Thread 2 will gather intermediate metrics and it would overwrite the intermediate metrics gathered by Thread 1. And even a worse case, they could overwrite metrics that were already fully calculated.

To avoid these situations, before gathering intermediate metrics, we query the Metrics Storage System (explained in more detail in Section 2.3) asking if the number we want to gather intermediate metrics were already full factorized or not, if not we need to perform two additional checks. We need to check both the thread ID and instance ID to check if this is the thread that first gathered intermediate metrics about this request, if it's not, we don't update it, if it is then we update it. If the number was already full factorized, then we don't need to gather intermediate metrics. For better understanding, in Section 10.2 in the Appendix we have a data flow diagram that represents what was just explained

Before storing the final metrics in the Metrics Storage System, we check if there's already an entry for it there, if there is then we don't need to put it there. This way we save the overhead of storing redundant information in the Metrics Storage System and we also save storage space. Next we will present how the Metrics Storage System is architectured.

2.3 Metric Storage System

As Metric Storage System we decided to use Dynamo provided by Amazon. The reason we chose Dynamo was that a NoSQL database provided a good performance for the type of information we are going to store then and also for learning purposes.

We have one table that stores the following information:

- **Parameter:** number of the request that was factorized;:
- **Basic Blocks:** number of basic blocks executed;
- **Fields Loads:** number of field loads;
- **Instructions:** number of instructions executed;
- **Rank:** the estimation cost for the Parameter (explained later in Section 5);
- **Running:** a boolean that if it is true, then we know its intermediate metrics;
- **InstanceID:** used to avoid conflicts when calculating intermediate metrics as explained in Section 2.2.2;
- **ThreadID:** same as InstanceID;

Now that we know how metrics are gathered and stored, in the next section we explain why we chose this metrics.

3. Instrumentation Metrics

We chose four different types of metrics:

- Instructions executed;
- Basic blocks executed;
- Fields Loads;
- Loads from the Stack.;

In separate these metrics wouldn't be good to estimate the complexity and cost of a request, for example, if we used just the number of instructions to evaluate a request we wouldn't obtain a good result since more instructions isn't directly proportional with the time that a request takes to be executed. For this reason we believe that using all these four metrics together, can provide a good insight about the complexity of a request.

We also thought about using the number of stores as another metric since IO operations have a significant impact on the performance, but in this case the numbers of stores are low so it wouldn't bring anything new to the four metrics we've chosen. In the following Section, we will present what other data structures we use to hold important data.

4. Data Structures

We use the Metrics Storage System to hold the metrics gathered by the workers as already explained in Section 2.3. In the Load Balancer we also have storage, persistent and internal (in memory) storage. When the Load Balancer queries the Metrics Storage System for the cost of a request (Rank, Section 5), if it returns a result, we also store the Rank in memory and in a file so the next time the Load Balancer gets the same request, he doesn't need to query the Metrics Storage System for the cost of that request because he already has it in memory so we save a big overhead.

Like mentioned before, besides storing in memory, we also store it persistently (in a file), this is because if, for some reason, the Load Balancer is shut down, when its turned on again, while initializing he reads the costs stored in this file and loads them to memory in order to avoid queries to the Metrics Storage System.

In the following section we will explain how this Rank that we've been mentioning is calculated in order to estimate the cost to calculate a given request.

5. Request Cost Estimation

In order to estimate the cost of a request, we classify it using a Rank. This rank is calculated in the following way:

*Rank = (Number of Instructions * 0.4) + (Number of Basic Blocks * 0.15) + (Number of Field Loads * 0.05) + (Number of Loads from the Stack * 0.4)*

We chose the number of Instructions and Field Loads as the two that most significantly impact the performance and therefore assigned both of them bigger weights, since generally the greater the number of instructions the greater the complexity (although not always the case like mentioned before) and IO operations like loads from the stack also have a significant impact on the performance of a program. As for the Basic Blocks and Fields Loads, their impact isn't as significant as the other two but they still provide a helpful insight about the complexity of a request. The higher the rank, higher the complexity of the request and therefore the longer it will take to factorize the request. We performed several tests and in 100% of the cases, the higher the rank the longer it took to complete a request.

Since this a simple calculation, we perform the calculation of the Rank at the Workers just before they store the metrics in the Metrics Storage System and then we store the metrics and the rank associated with those metrics. Although the Workers are the biggest point of congestion in our cluster, we think that it compensates to do this calculation at the Workers and saving some work at the Load Balancer as it can also be a big point of congestion if many requests arrive at the same time.

5.1 Other cases

There are two other cases that can occur, when there's no Rank for a given request in the Metrics Storage System and when it's an intermediate Rank (from intermediate metrics).

When there's no Rank in the Metrics Storage System, the Load Balancer assigns the Rank as the number of the request, so a request of 983732 would have a rank of 983732. This is the best approximation we can do with no information about that request and it actually isn't that bad estimation as some of the tests we did demonstrated, as its usually a medium case estimation which is better than a best case estimation because a best case estimation could easily overflow the Workers with requests.

When we fetch a Rank from the Metrics Storage System and its flagged as being from intermediate metrics (identified by Running = True, as explained in Section 2.3) we assume the worst case, that is, we assume it's going to be a long operation and assign it a MAX classification (explained in the following subsection).

The procedure for assigning a Rank to an incoming request in illustrated in a dataflow diagram in Section 10.3.

5.2 Classification of Requests

Besides the Rank, we also decided to classify the requests based on their Rank. This classification will help us in our scheduling algorithm as it will be explained in next Section. We decided to use two classifications, MAX and INTERMEDIATE.

A request that is classified as having a MAX classification is considered to be a long operation, we consider a long operation in this case, more than 10 minutes to be factorized. To decide what Rank we should use to start classifying requests as MAX we did tests while counting the time until a request reached 10 minutes and then we went to check out his Rank. We did this tests with different types of loads on the Worker (same as for the test with the INTERMEDIATE classification) and reached a decision that a request that has a Rank equal or greater than *1272110814* will be classified as being MAX.

For us, an INTERMEDIATE request is a request that takes more than 1 minute up to 10 minutes (more or less), so we performed the same tests as we did for MAX but with a request that took 1 minute to be factorized and we reached an agreement of using the Ranks greater or equal than *161751155* and less than *1272110814* as being INTERMEDIATE. As for the remaining of the requests, the ones lower than INTERMEDIATE, we consider them to be short requests. Now we will see how this classification helps us when scheduling a request to a Worker.

6. Task Scheduling Algorithm

Before going into the scheduling algorithm it's important to mention some aspects that we considered for developing this algorithm. We defined a limit of MAX and INTERMEDIATE requests a Worker can handle and we don't allow them to coexist, that is, we don't allow Workers to be working with INTERMEDIATE and MAX requests at the same time. This is because the MAX requests impact very significantly the time it takes for INTERMEDIATE requests to be processed and INTERMEDIATE requests aren't supposed to take too long so by keeping them separate INTERMEDIATE requests can be completed faster than if they coexisted with MAX requests.

These limits were carefully tested based on the impact they had on the processing time of short requests (less than INTERMEDIATE). From our tests we saw that after 10 MAX requests, short requests would start to be impacted significantly (taking much more time to be processed than they supposed to) so we defined a limit of 10 MAX requests per Worker. For establishing a limit of INTERMEDIATE requests per Worker, we did the same procedure and we reach a limit of 20 requests. The Load Balancer knows how many MAX and INTERMEDIATE requests each Worker currently has.

6.1 Algorithm

For better understanding of the algorithm, in Section 10.4 we have a dataflow diagram of the algorithm. When a request arrives, after assigning it's Rank, of the list of all Workers currently available, we will fetch the one dealing with less requests as it is the most probable Worker that this request will fit. After we got the Worker, we see what Rank is this request, if it's a Rank of a short operation (less than INTERMEDIATE) then we can immediately send it to the Worker. We can do this because of the limits we mentioned before, these limits exist to guarantee that any short request will fit in any Worker so we simplify the scheduling algorithm a lot for short requests.

When the request is not a short one, more things must be taken into consideration. Our first consideration is the current CPU usage of the Worker, if its greater than 75% then we don't even consider this Worker and analyze another Worker (we fetch the next one with less requests). To note that if at this point, if this request has not fit in all of the Workers available, we create another instance and after its created we send the request to it (explained in more detail in Section 7).

If the Worker current CPU usage is less than 75% then we check if our Rank has a MAX classification, if it has we check if this Worker hasn't reached the limit of MAX requests and if it's not dealing with any INTERMEDIATE request, if both of these condition are true, then we can send it to the instance, otherwise we analyze another Worker (if all available Workers have been analyzed, create a new instance).

At this point, if our Rank isn't MAX or a short request, then it's an INTERMEDIATE one so we need to check similar conditions as a MAX request does. We check if this Worker hasn't reached the limit of INTERMEDIATE requests and if it's not dealing with any MAX request, if both conditions are true, we can send the request to this Worker, otherwise we analyze another Worker (if all available Workers have been analyzed, create a new instance).

We believe that this algorithm keeps the system well balanced and also provides a reasonable client experience, which was also one of our goals for this project.We provide a reasonable client experience by guaranteeing that short requests are quickly answered as they are supposed to.If a client sends a request to factorize a small semiprime, he expects a result in a matter of seconds and we assure this with our scheduling algorithm. Next we present our algorithm for creating and removing instances.

7. Auto-Scaling algorithm

Like it was made clear in the last Section, the criteria for starting a Worker is when a request doesn't fit in any of the current Workers. However, we've over simplified this in the last Section. We cannot simply start a Worker for every request that doesn't fit in a Worker as this would create a lot of Workers. For this reason, we have some rules for starting a Worker. As for starting Workers, we also have rules for removing a Worker which will also be explained afterwards.

7.1 Creating a Worker

To avoid a lot of Workers being created when requests don't fit in all current Workers, we only allow one Thread to start a Worker for a certain amount of time, a grace period amount of time. When a request doesn't fit in any of the Workers it passes through a control variable that is evaluated to True if a Worker is currently being started. If a Thread reaches this control variable and its value is true, then it goes into a queue where it will wait for a grace period amount of time until it exits the queue. In the other case, if a Thread reaches this control variable and this control variable is false, this Thread sets this control variable to true and starts the Worker and after a grace period amount of time it is sent to the Worker.

The grace period is a value calculated by us based on the time it takes to start a Worker, this value had to be carefully calculated because we could lose requests by sending requests to a Worker that hasn't been created yet. From our calculations in average it took, 2.30 minutes to start an Worker, to play safe, we apply a grace period of 3 minutes.

When a Thread starts a Worker it needs to wait for the grace period for the reason we just mentioned. The Thread responsible for creating the Worker is also responsible for starting the Threads that are responsible for monitoring the new Worker CPU usage and another Thread for performing Health Checks to that Worker.

Like mentioned before, when a Thread reaches this control variable and its value is true, it goes to a queue for a grace period amount of time. This is because we need to give time for the new Worker to be started but we don't immediately send them to the Worker. These Threads that are in the queue, after they leave the queue, they start the scheduling algorithm from the beginning. We do this because grace period is quite some time and in the meantime, the Workers that were otherwise full, can now be available to receive new requests.

7.1.1 Optimization

We performed an optimization to this queue. If the queue reached a size of 20, the 21st Thread that reached the queue will start a new Worker and reset the queue size to 0, therefore creating two Workers instead of one like mentioned previously. We do this because what's in the queue are INTERMEDIATE and MAX requests and since the number of those requests per Worker are limited, there's no point in keeping a huge queue because immediately after they leave the queue, they would quickly go back into the queue because the newly created Worker was already in full capacity (based on our limits). By resetting the queue to 0 after the 21st Thread we allow for more than two Workers to be created if we have a big amount of requests pending to be sent to Workers.

7.2 Removing a Worker

As for removing Workers, they can be removed in two ways, by failing a health check or by low CPU usage. Both cases are treated in different ways as it will be explained next.

7.2.1 Failing a Health Check

Like mentioned in Section 2.1.1, if a Worker reaches 3 unhealthy thresholds its declared has being down and actions must be taken to compensate for that fault.

It might happen to be a false positive, so to be safe, if the unhealthy threshold is reached we explicitly remove the Worker and we transfer all the requests this Worker was dealing with (explained in Section 8). If a Worker crashes, it makes sense to create a new Worker to compensate for the one just crashed but it might not compensate to create a Worker because at the moment we could have Workers with low CPU usage so it would not make sense to create another Worker to also have low CPU usage. What we do is then the following. Let's imagine the case where we have a Worker with 90% CPU usage and another with 20% CPU usage, if a third one crashes, we allow that one is created to compensate for the faulty one.

We always allow two Workers to be with low CPU usage because imagine we only allowed one, it could happen the case that it got immediately flooded and we would have to wait for a new one to be created and the requests would be pending for the time it took for the new Worker to be created (the grace period time, mentioned in Section 7.1). To summarize, after a Worker crashed, we only allow a Worker to be created if there are less than 2 Workers with 20% or less CPU usage.

7.2.2 Low CPU Usage

We consider a Worker with low CPU usage when its current CPU usage is less or equal than 20%. We have a counter that is incremented every minute if the CPU usage is less or equal than 20% and if this counter reaches 11, we remove the Worker (or not, explained next). We let the counter be so high before removing the Worker because we need to give it some time for it starting receiving requests.

As we mentioned in Section 2.1.1, we use Detailed Monitoring to keep a better track of the CPU usage and since Detailed Monitoring updates the CPU usage every minute, we query the CloudWatch for the CPU usage every 1.15 minutes. As we said in the last paragraph, we might not remove the Worker just because his current CPU usage is lower than 20% based on the following checks. The first check is that we only remove the Worker if there are more than 2 Workers. By default, we have 2 Workers like mentioned in Section 2.3.

If there are more than 2 Workers, we have some rules to let a Worker be removed. First we check if there's any Worker with high CPU usage (greater than 75%), if there isn't we can remove this Worker because the others Workers can handle the incoming requests. If there's a Worker with high CPU usage, it might not compensate to remove the Worker so we perform an additional check. We check if there's any other Worker with low CPU usage, if there are more than 2, then we can remove this Worker, remember that we allow two Workers to exist with low CPU usage for the reasons mentioned before. To finish this report, we will address the case to what happens to the requests if there's any fault in cluster.

8. Fault-Tolerance

There can be different types of faults in the cluster. A Worker might fail or be removed due to low CPU usage (the second not being a fault though), a request might fail from the Load Balancer to the Worker due to network reasons or the Worker might be able to cope with the request if heavily loaded and the request is lost.

Our cluster is resilient to all the cases mentioned in the previous paragraph and now we will describe how we deal with them. The Load Balancer knows what requests each Worker is dealing with, if a Worker crashes or its removed, we go to the list of the requests that Worker was dealing with and we simply transfer them to other Workers, more precisely, we send them through the scheduling algorithm again where they will be sent to a new Worker.

To deal with the case of a network error or a Worker cannot handle the request and it fails due to a timeout, for example, we have try and catch blocks that will deal with the exceptions thrown by these two cases. Like before, if an exception occurs for any of these two cases, we send the request that failed to the scheduling algorithm again.

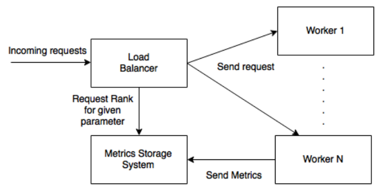
All this is dealt by the Load Balancer and doesn't require the clients to resend the request so it's completely transparent to the clients that a fault occurred in the cluster.

9. Conclusion

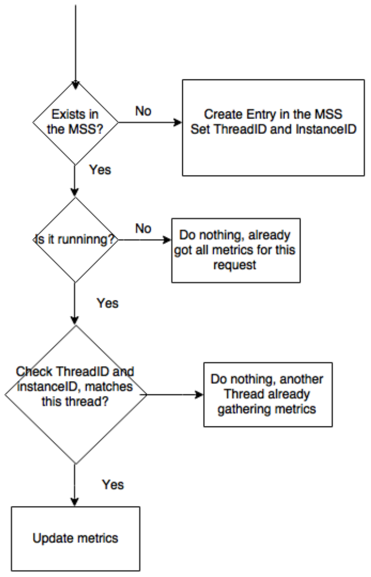
Like mentioned several times throughout this report, our main goal was to ensure a good load balancing of the cluster, a reasonable client experience and not losing any requests. All these objectives were successfully completed thanks to our scheduling algorithm that allows that short requests are quickly answered and by limiting the amount of big requests each Worker can cope we ensure that they are never heavily loaded delaying even more the answers of those requests. Our scaling algorithm also ensures that there's always enough Workers to deal with the incoming requests and if there's not, we put them in a queue while we create new Worker(s) to deal with those pending requests. Finally, our Fault Tolerance mechanism allows transparency to the users when a fault occurs in the cluster.

10. Appendix

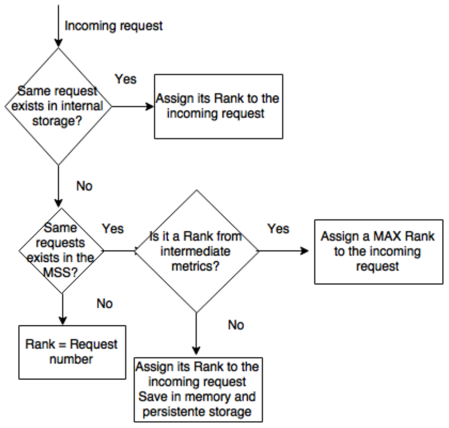
10.1 Overall system architecture/data flow



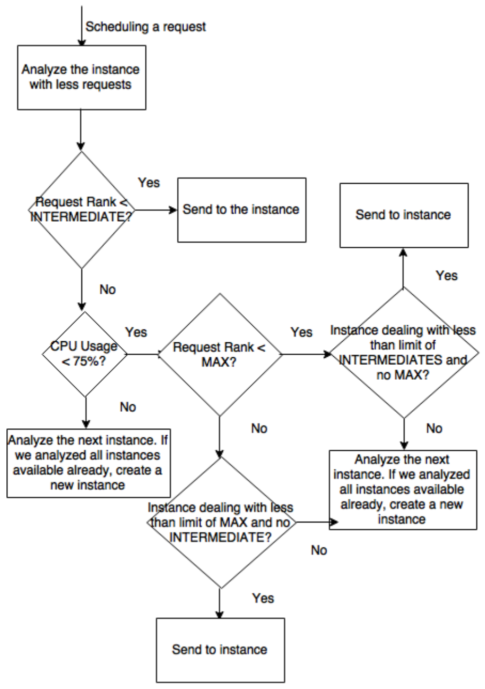
10.2 Steps for gathering intermediate metrics


Exists in the MSS?
No
Create Entry in the MSS Set ThreadID and InstanceID
Yes
Is it runninng?
No
Do nothing, already got all metrics for this request
Yes
Check ThreadID and instanceID, matches this thread?
Do nothing, another Thread already gathering metrics
Yes
Update metrics

10.3 Assigning a Rank to an incoming request


Incoming request
Same request exists in internal storage?
Yes
Assign its Rank to the incoming request
No
Same requests exists in the MSS?
Yes
Is it a Rank from intermediate metrics?
Yes
Assign a MAX Rank to the incoming request
No
Rank = Request number
No
Assign its Rank to the incoming request Save in memory and persistente storage

10.4 Scheduling


Scheduling a request
Analyze the instance with less requests
Request Rank < INTERMEDIATE?
Yes
Send to the instance
No
CPU Usage < 75%?
Yes
Request Rank < MAX?
Yes
Instance dealing with less than limit of INTERMEDIATES and no MAX?
Yes
Send to instance
No
Analyze the next instance. If we analyzed all instances available already, create a new instance
No
Analyze the next instance. If we analyzed all instances available already, create a new instance
No
Instance dealing with less than limit of MAX and no INTERMEDIATE?
No
Yes
Send to instance