DDVTECH RESEARCH INITIATIVE, LOAD BALANCING, 2016 Q2

JARON VIËTOR, CTO, MISTSERVER / DDVTECH B.V.

About this paper

The following paper is a university-level research project, done under supervision of Dr. Michael S. Lew of the University of Leiden, the Netherlands.

In it, two load balancing strategies are compared: algorithm 1 and algorithm 2.

Algorithm 1 is a recently published state of the art example of load balancing, while algorithm 2 is DDVTech's new load balancing algorithm.

To paraphrase the results, DDVTech's algorithm is significantly better in all tested load simulations, providing a less crash-prone method to balance overload, and generally resulting in smoother and more predictable distribution of load over servers.

Read on for full details, or skip straight to the conclusion at the end for a summary.

Date: 2016 Q2.

A streaming media-specific load balancing algorithm

J. Viëtor
Supervisor: Dr. Michael S. Lew
Reader: Dr. Kristian Rietveld

June 15, 2016

Abstract

Load balancing, the process of spreading out server load over multiple server instances, is a problem that has many known solutions already in a wide variety of specific use cases. There are also generic methods available that tend to work well in most situations. Streaming media delivery is a very special case however. It places a relatively high requirement on bandwidth as well as available server RAM, while not requiring all that much CPU power, while in most existing algorithms CPU is the main deciding factor. A new method to load balance multiple servers specifically geared towards streaming media delivery is proposed, with special consideration given to the effect of new incoming load on bandwidth as well as being cache-friendly. The algorithm is then tested against a recently published alternative in several simulations, followed up by a detailed analysis of the measured results.

1 Introduction

Load balancing is not a new problem. It has been widely researched for use cases such as grid computing and automated processing pipelines, but also web servers and databases in particular have receive a lot of attention in this area. Streaming media applications are however a relative newcomer to the server ecosystem, and there are not all that many publications or even practical results that can be compared to each other fairly for the specific use case of streaming media delivery.

This paper aims to change that, both by providing a new load balancing algorithm that is specifically tailored to streaming media delivery and by providing a repeatable set of load balancing algorithm tests that can serve to compare such algorithms objectively.

The new method specifically geared towards streaming media delivery that is proposed in this paper is based on the combination of an agent that collects up-to-date status information on the servers being balanced amongst with a scoring algorithm that attempts to predict future network load and is cache-aware in the sense that is will attempt to group clients for same assets together on same servers. The full details of this algorithm are given in Section 4.

A literature study has been done to find a recent state-of-the-art example of a load balancing algorithm, to compare our new method against. In the end, the work by X. Jiang, et al., [JLY13] was selected to be a representative candidate to compare against because it is both recent and specific to streaming media delivery. More on the details of this algorithm in Section 3.

Other related work and candidates for comparison are discussed in the following section.

2 Related work

The work by Gupta, et al., [GGG15] also describes load balancing techniques, focussing more on future reliability as opposed to the current state. It would have made a good candidate for a comparison as well, as it provides a good description of the algorithm and is a more recent publication than X. Jiang, et al., [JLY13], but is not specially geared towards media delivery and instead more appropriate for grid processing. The main reason for this is that it makes the assumption that higher load means a higher fault ratio, while when it comes to media higher load instead tends to mean a higher cache hit ratio and thus more optimal delivery. Though the methods described in it look very solid, this makes them unsuitable for a media-centric load balancer.

The work by Diallo, et al., [DFAEA14] is another more recent publication. The method it describes is actor-based and splits the load balancing work over three distinct groups (content provider, operator and client). Since this method requires three actors as opposed to just one to properly implement and test, it could not be used in a fair comparison. It could however theoretically be combined with this method, as the formulas described for the operator role are very similar to both algorithms discussed in this paper. It would seem feasible to replace just this part of the method with either of the two algorithms and apply the full three-actor technique as-is.

The work by Han [Han12] is an older but still interesting publication. It focusses more on the task of balancing network load and making sure to avoid overloading certain connections as opposed to certain machines/servers. Since it practically ignores server load and has such a big focus on network load, it is however unsuitable for a comparison. The techniques used to prevent overloading certain network connections could easily be combined with either of the algorithms discussed in this paper for an even more complete load balancing solution, however.

Two publications by Q. Jiang, et al., [JXY07, QHBC08] seem to be covering roughly the same technique. This is a very interesting approach that has a clear parallel with our Algorithm 2: it too attempts to encourage cache hits. This technique instead of the relatively simple approach taken in our Algorithm 2 uses Markov-based methods. While it is very interesting and likely will score well, these publications were considered too dated to use in a state-of-the-art comparison, being almost ten years old.

The work by Ma, et al., [MDW12] describes a method based on simulated annealing for load balancing media applications, and looks very promising. Sadly the publication does not provide any implementation details or formulas of any kind, thus unfortunately making any kind of comparison impossible.

The work by Espeland, et al., [ELS+08] is a very novel approach to the load balancing problem. It proposes a network appliance (such as a router) to do both protocol translation and load balancing of the servers that are behind it. The publication however focuses mostly on the protocol translation and only touches the subject of load balancing very briefly. Likely a straightforward load balancing algorithm was used, which means it could possibly benefit from implementing (some of) the techniques mentioned in this paper.

3 Details of Algorithm 1

The method described by X. Jiang, et al., [JLY13] we shall call Algorithm 1. This method describes a Server Weight, scaled from 0 to 1, where 1 means fully available and 0 means fully utilized. It defines weights for `CPU` (processor utilization), `MEM` (RAM utilization), `T` (network bandwidth), `IO` (input/output operation utilization), `SC` (buffer) and `P` (process count) which must all sum up to 1.

These weights are dynamic, changing over time. For each measurement interval, the new weight is calculated from: $a_1 * \texttt{oldWeight} + a_2 * (\texttt{oldVal} - \texttt{newVal})/(\texttt{oldSum} - \texttt{newSum})$.

The values of the coefficients a_1 and a_2 are not mentioned in the text, besides their sum needing to be 1, so we will pick some sensible values, namely $a_1 = 0.9$ and $a_2 = 0.1$, which complies with that condition. These values were picked because a high amount of retention of the original value (90%) versus a smaller change to the value (10%) will cause uncommon changes in the resource use to not affect the algorithm too much, while common changes still will have a significant effect.

Our testing method only load-balances full sessions at a time, so the cookie method described by X. Jiang, et al., [JLY13] is not implemented. All allocations are assumed to be permanent for the duration of a single resource use (session), meaning the dynamic feedback load balance method is always used and there is no caching of these values.

Additionally, we will not use the signal processing method that is described, but only work with raw values from the algorithm. This is because streaming media specifically has a steady load usage pattern and not a wave-like pattern, so signal analysis does not add to the usefulness of the algorithm.

Since there is no mention of how `T` is used versus `IO`, we will assume `T` is the maximum available bandwidth and `IO` is the currently used bandwidth, scaling the value from 0 to 1 between 0 and `T`.

To score a single host, on a scale from 0 to 1:

```
score = cpuRate * cpuModif + ramRate * ramModif + bwRate * bwModif;
```

Every data refresh, perform:

```
if (numbersChanged){
  cpuModif = 0.9 * cpuModif + 0.1 * ( (cpuPrev-cpuCurr) / (totPrev-totCurr) );
  ramModif = 0.9 * ramModif + 0.1 * ( (ramPref-ramCurr) / (totPrev-totCurr) );
  bwModif  = 0.9 * bwModif  + 0.1 * ( (bwPrev -bwCurr ) / (totPrev-totCurr) );
  //ensure the total is always 1.0 for the total weights
  if (cpuModif + ramModif + bwModif != 1){
    tot = cpuModif + ramModif + bwModif;
    cpuModif /= tot;
    ramModif /= tot;
    bwModif  /= tot;
  }
}
```

Figure 1: Pseudocode for Algorithm 1

There is no mention of what "buffer" is, so we will not use the value. Since number of processes and CPU usage tend to go up and down simultaneously with equal measure, we will simplify these to just CPU usage, leaving only the CPU usage, network bandwidth and RAM usage as variables in the load balancing.

Finally, to make sure the weights of all three maintain a sum of 1, after each new weight is calculated after each measurement step, each weights is divided by the sum of all weights.

Figure 1 displays the pseudo-code for Algorithm 1.

4 Details of Algorithm 2

The new method described here we shall call Algorithm 2. The underlying principle is that we assume that CPU, RAM and network bandwidth are all equally important. In other words, if any of these three resources start to run out, this is equally bad. Furthermore, we assume that if an asset is already cached on a server, it will take less resources to fulfil a request for it. The scoring is altered in such a way that only a 20% disadvantage in any of the resources will offset the advantage of using an existing cache. This value was chosen as a sensible default (not too high to be dangerous, not too low to have no effect) but may turn out to need further tweaking depending on the type of network, traffic and/or servers involved. Finally, we assume that a new client will use the average amount of bandwidth of current clients, and temporarily add this as an offset to the current bandwidth. CPU and RAM are not taken into account here, as in practice often bandwidth is the only limiting factor to how many media clients a single server can handle.

This method uses a score that does not scale from 0 to 1, but instead from 0 to 3200. The scoring works as follows:

A thousand points each are given for CPU, network and RAM usage, all scaled so that 0 means fully utilized and 1000 means fully available.

An additional 200 bonus points are given if the requested resource is already available on a specific server, and not given out if this is not the case (there is no scale: either 200 or 0 is used).

Furthermore, the score for network usage is lowered using a predictive model of the network usage. The way this works, is that for every allocation of a session to a server, an estimate is made of the bandwidth that this new session will be using in the near future. The estimate is based on either the average bandwidth used for that particular asset (if known) or an average of all sessions currently active on the server. If neither is available (for example because the server was just (re)booted), 0.5 megabit per second is assumed. This estimate is then clamped between 0.5 and 8.0 megabit per second to prevent huge or tiny values from throwing off the algorithm, and applied to the server as if it were real load, lowering the network value (but not below 0). The total of this faked load is then lowered by 10% for every measurement interval, under the assumption that this way actual load will slowly take over from the fake load.

Figure 2 displays the pseudo-code for Algorithm 2.

5 Comparative analysis method

It is somewhat complicated to benchmark load balancing techniques. Any approximation of real traffic, no matter how well thought-out is always going to be a poor substitute for real traffic generated by real users. Even so, real systems usually have dozens of servers to balance load over, and using dozens of servers for a benchmark is both expensive and complicated to set up. Load balancers often have to deal with situations outside of normal operating conditions, such as servers going down or a significant portion of the current connections unexpectedly reconnecting all at once. Finally, there is no way to realistically model various assets being in high or low demand.

The work we're comparing against ([JLY13]) also does not help us select a good testing method, as the description of the tests they did was extremely minimal and left out almost all the implementation details.

Because of the above reasons, a simplistic setup was chosen with only a single 10-minute duration media asset being requested from three server machines. This setup was then tested with both algorithms discussed in the previous two sections, under three different load conditions. During the tests, the CPU, RAM and network utilization information that the load balancers used and that was

To score a single host, on a scale from 0 to 3200:

```
//First, add current CPU/RAM left to the score, on a scale from 0 to 1000.
score = cpuRate * 1000 + ramRate * 1000;
//Next, we add 200 points if the asset is already in cache.
if (assetAlreadyCached){score += 200;}
//Finally, account for bandwidth. We again scale from 0 to 1000 where 1000 is perfect.
bwScore = (1000 - ((currentBandwidth * 1000) / availBandwidth));
//Prevent values under 0 during overload
if (bwScore < 0){bwScore = 0;}
bwInfl = ((bandwidthInflation * 1000) / availBandwidth);
if (bwScore - bwInfl < 0){bwInfl = bwScore;}
score += (bwScore - bwInfl);
```

After a host is chosen, perform:

```
toAdd = currentAverageBandwidth;
//ensure reasonable limits of bandwidth guesses
if (toAdd < 64*1024){toAdd = 64*1024;}//minimum of 0.5 mbps
if (toAdd > 1024*1024){toAdd = 1024*1024;}//maximum of 8 mbps
bandwidthInflation += toAdd;
```

Every data refresh, perform:

```
bandwidthInflation *= 0.9;
```

Figure 2: Pseudocode for Algorithm 2

collected for graphing both came from an internal statistics module embedded in the media server software used.

Machine A is a powerful machine, dedicated to streaming and nothing else. This simulates a dedicated server. It features an AMD FX-8120 Eight-Core Processor at 1.4Ghz, 16GiB of RAM and gigabit Ethernet.

Machine B is a slightly less powerful machine, with many background tasks creating interference in the load pattern. This simulates a shared server. It features an AMD FX-4130 Quad-Core Processor as 1.4Ghz, 8GiB of RAM and gigabit Ethernet.

Machine C is a severely underpowered machine with limited bandwidth (100 megabits per second). This machine simulates remote and/or temporary and/or cloud-based resources. The load balancing algorithms are not informed about the limited bandwidth, instead led to believe this machine can handle a gigabit per second. The configuration was then updated with the correct bandwidth limits and the failed tests were repeated. This machine is in actuality a Raspberry Pi 2 model B, featuring a 900MHz quad-core ARM Cortex-A7 CPU and 1GiB of RAM.

All three machines are running a fully updated minimal text-mode Arch Linux installation, with all non-critical services disabled with the exception of `ntpd`. As media server software, all three are running the open source edition of `MistServer 2.6`.

Machine B runs some additional services on top of this, generating background load: `prometheus`, `grafana`, `docker` and `nginx`, all handling a consistent flow of tasks to keep the tests equal and fair.

All the media requests sent to the three servers a coming from a single (separate) machine with a gigabit wired network connection to the other three machines. A single request means a single complete play through of the 10 minute media asset at real-time speed. The machine sending the requests was the same machine that was running the load balancing algorithms.

The **trickle** test will send a new request once per second, for 1500 seconds total (a total of 1500 requests). Afterwards, the system pauses for 10 minutes to let the pending requests finish. This test simulates expected day-to-day use of a media system.

The **waves** test will send 100 new requests every 5 seconds, for 75 seconds total (a total of 1500 requests). Afterwards, the system pauses for 10 minutes to let the pending requests finish. This test simulates sudden popularity spikes, as are common in TV series and similar content.

The **burst** test will send 1500 new requests all at once. Afterwards, the system pauses for 10 minutes to let the pending requests finish. This test simulates mass reconnects when an server goes down unexpectedly and the entire load of a whole server must suddenly be re-balanced over the remaining servers.

Since the available bandwidth of machine C was misconfigured during the tests, both the waves and burst test overloaded machine C's network connection for both algorithms (as one would expect to happen). To see what happens with the correct configuration, these two tests were repeated as waves-2 and burst-2, with the correct configuration given to the algorithms. This allows us to directly see the effects of a misconfiguration versus a correct configuration for both algorithms.

The entire test setup was repeated three times in full. All three runs had the same usage patterns with only minor differences, so only the last run was converted into graphs.

While this setup is hardly realistic, it should provide us with good estimates of how the load balancing algorithms will behave in various real situations, particularly under both normal and abnormal working conditions.

6 Comparing trickle test results

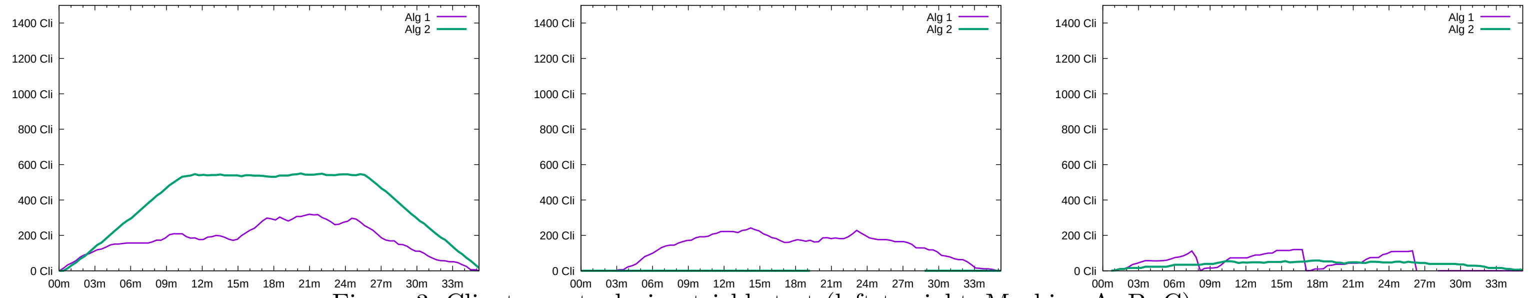

Figure 3: Client counts during trickle test (left to right: Machine A, B, C)

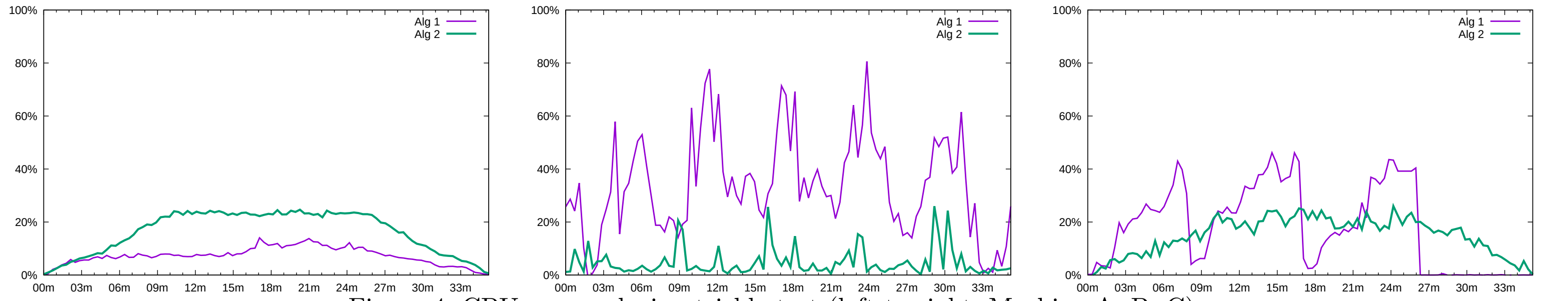

Figure 4: CPU usages during trickle test (left to right: Machine A, B, C)

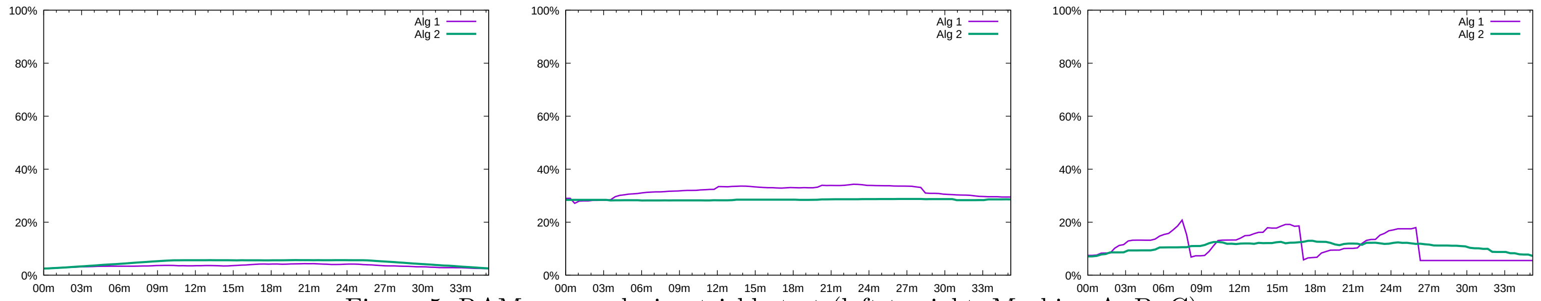

Figure 5: RAM usages during trickle test (left to right: Machine A, B, C)

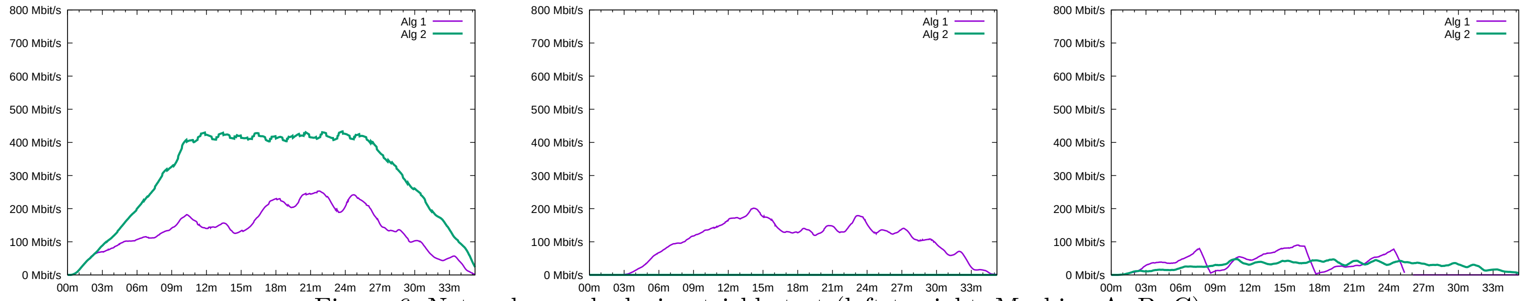

Figure 6: Network speeds during trickle test (left to right: Machine A, B, C)

Table 1: Summary of system vitals during trickle test

Machine	Avg. CPU		Avg. RAM		Avg. MB/s		Client total		Successful	
	Alg 1	Alg 2	Alg 1	Alg 2	Alg 1	Alg 2	Alg 1	Alg 2	Alg 1	Alg 2
A	7%	16%	4%	6%	17.90	39.00	629	1370	629	1370
B	35%	5%	32%	28%	13.29	0	467	0	467	0
C	18%	15%	11%	11%	3.96	3.73	404	130	404	130
All	20%	12%	16%	15%	11.72	14.24	1500	1500	1500	1500

Figures 3 - 6 and Table 1 show measurements taken during the trickle test. Being the least taxing of all tests but at the same time the most representative of a normal type of load, the trickle test sets the baseline for expected behaviour of the algorithms under normal situations.

A few things immediately jump to attention here: Algorithm 1 has much more erratic CPU usage and client spread patterns than Algorithm 2, and Algorithm 2 neglects to send any load whatsoever to machine B.

It is clear that Algorithm 1, being a dynamically reactive algorithm, is thrown off by both the background usage of machine B and the limited resources of machine C. It starts off by sending most load to machine A (as expected), but then notices the bandwidth rapidly decreasing and starts sending load to the other two machines as well. Machine B is immediately affected by the sudden increase in load, spiking its CPU usage heavily in reaction. Machine C gradually builds up, and the algorithm decides to stop sending new clients to machine C for a while as it becomes more used (again, as expected). This happens three times in a row. The result is heavily spiking CPU usage patterns for machines B and C, while A is quite underutilized even though it was expected to take the brunt of the load.

Meanwhile, Algorithm 2 notices that machine B is already quite busy and decides not to send any load to it since machines A and C can easily handle the incoming trickle of clients. Most load goes to machine A, because as machine C starts taking some of the load the algorithm notices that C is more quickly affected by this load. The result is light CPU usage for all machines throughout the entire test period.

In both cases, RAM usage is mostly stable and the network usage closely follows the client spread, showing that all clients received roughly equal bandwidth. This is an indication that they were all able to successfully retrieve the media asset.

While both algorithms “pass” in the sense that all clients were served successfully, Algorithm 2 was able to do so with significantly less CPU usage on average (20% for Algorithm 1 versus 12% for Algorithm 2).

7 Comparing waves test results

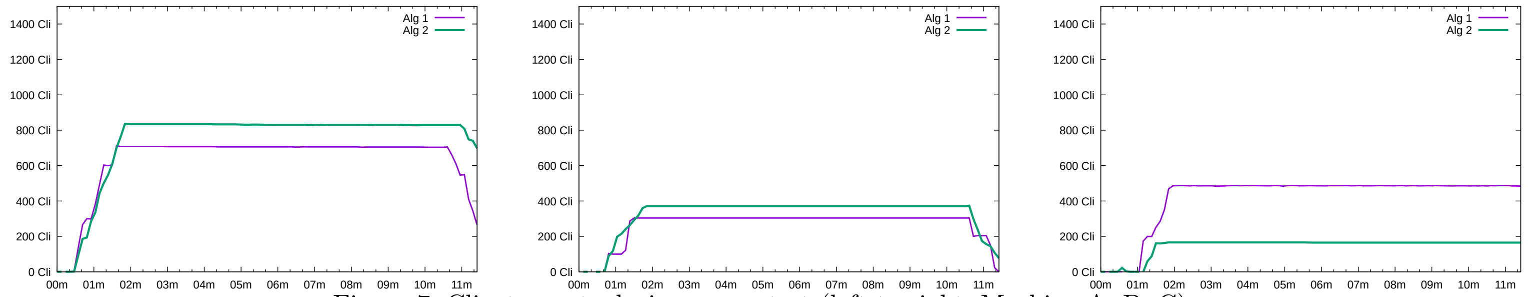

Figure 7: Client counts during waves test (left to right: Machine A, B, C)

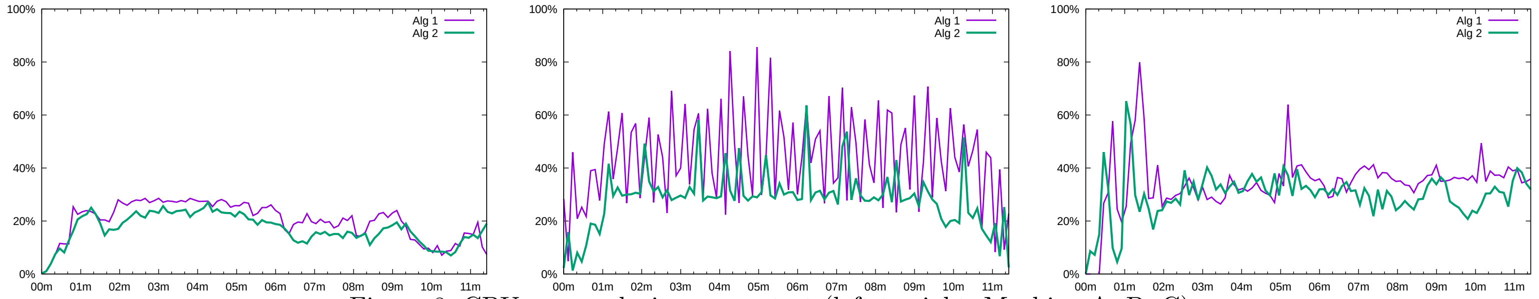

Figure 8: CPU usages during waves test (left to right: Machine A, B, C)

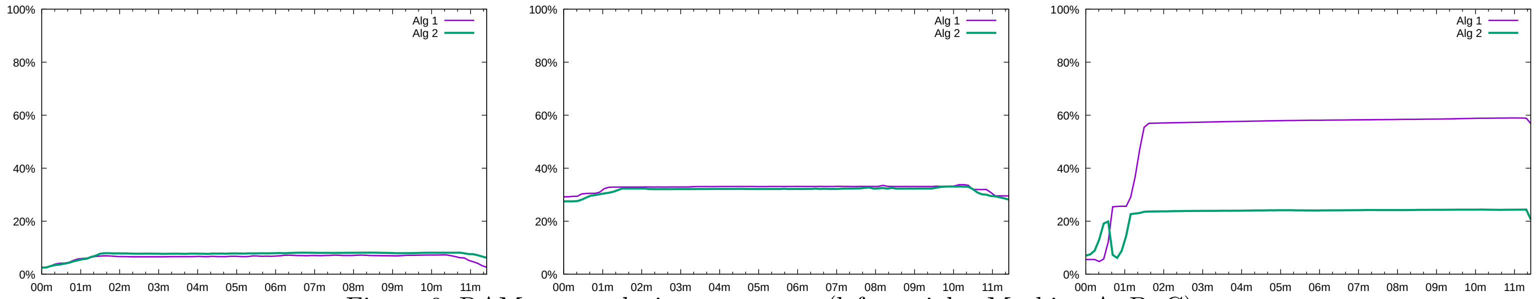

Figure 9: RAM usages during waves test (left to right: Machine A, B, C)

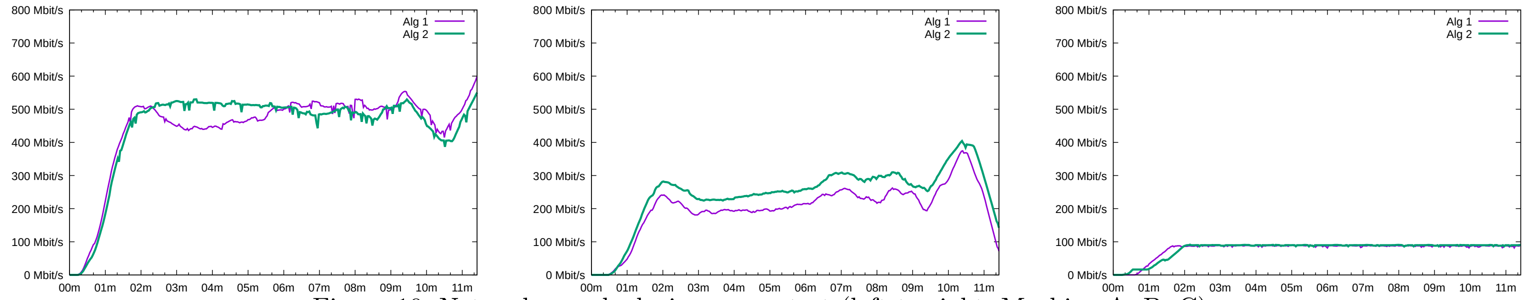

Figure 10: Network speeds during waves test (left to right: Machine A, B, C)

Table 2: Summary of system vitals during waves test

Machine	Avg. CPU		Avg. RAM		Avg. MB/s		Client total		Successful	
	Alg 1	Alg 2	Alg 1	Alg 2	Alg 1	Alg 2	Alg 1	Alg 2	Alg 1	Alg 2
A	20%	17%	6%	7%	60.20	59.90	708	834	708	834
B	44%	29%	34%	32%	26.70	32.50	304	371	304	304
C	34%	30%	53%	23%	10.88	10.65	488	295	0	0
All	33%	25%	31%	21%	32.59	34.35	1500	1500	1012	1138

Figures 7 - 10 and Table 2 show measurements taken during the waves test. The waves test contains several waves of small bursts of clients. It would be expected that in this test, differences between the two algorithms become much more clear as the situation is less standard and changes rapidly.

Oddly, this turned out not to be the case at all. In fact: the two algorithms perform almost identical on all fronts.

The only noticeable differences in client spread and CPU usage are that Algorithm 2 had a higher preference to send load to machine A, and that machine B was slightly more erratic in CPU usage under Algorithm 1. On average, however, the CPU usage patterns are pretty much identical.

There is a bigger difference in RAM utilization, as machine C's limited resources got very close to running out under Algorithm 1. However, they did not, and the bandwidth graphs for the two algorithms are amazingly similar to each other.

However, neither algorithm can be said to "pass" in the sense that all clients were served successfully: in both cases, machine C ran out of bandwidth and the connections that were made to it were not able to complete as a result. However, Algorithm 2 sent less clients to machine C and more clients to machine A, causing the this problem to be partially mitigated. Algorithm 1 has 1012 successful clients (67%) and Algorithm 2 has 1138 successful clients (76%).

8 Comparing burst test results

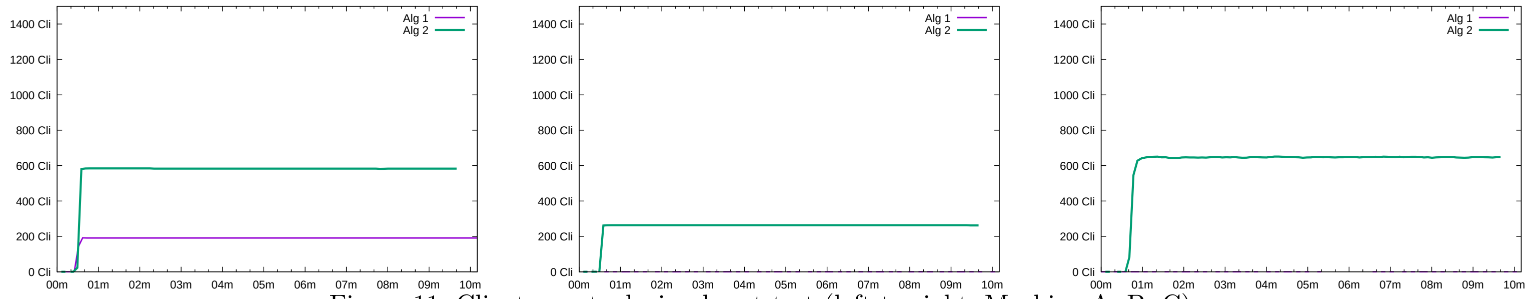

Figure 11: Client counts during burst test (left to right: Machine A, B, C)

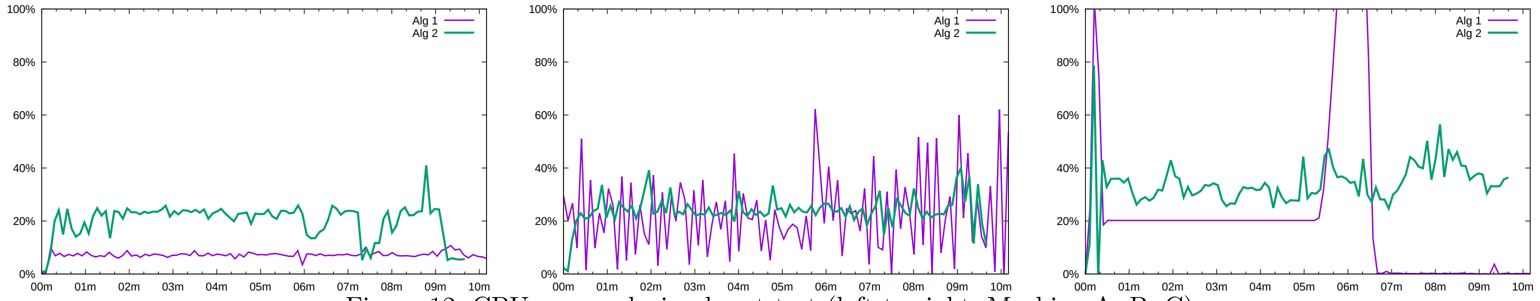

Figure 12: CPU usages during burst test (left to right: Machine A, B, C)

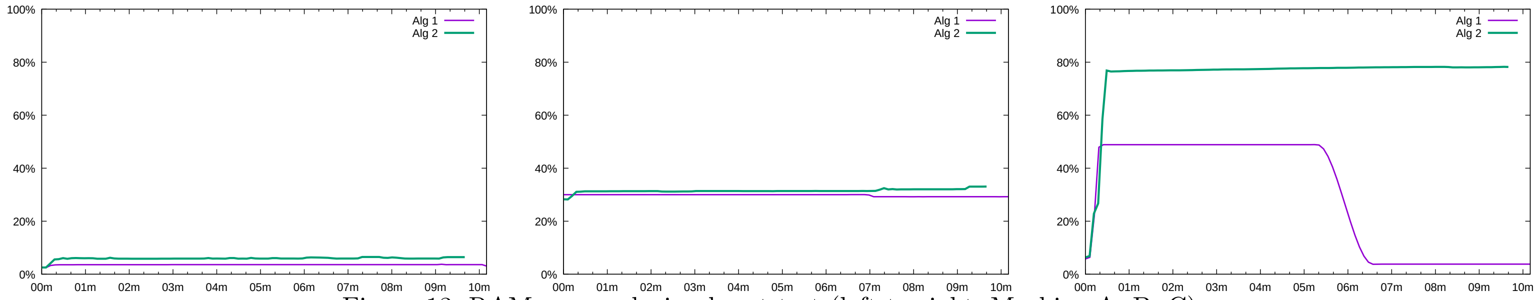

Figure 13: RAM usages during burst test (left to right: Machine A, B, C)

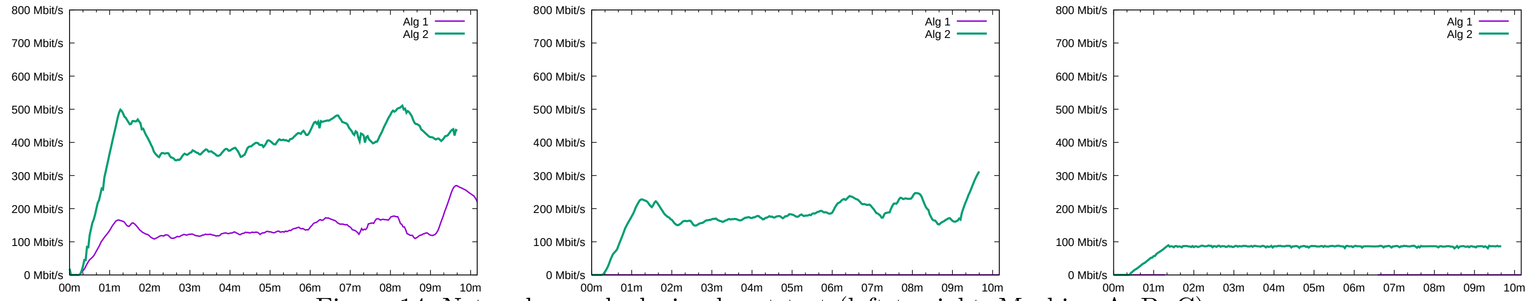

Figure 14: Network speeds during burst test (left to right: Machine A, B, C)

Table 3: Summary of system vitals during burst test

Machine	Avg. CPU		Avg. RAM		Avg. MB/s		Client total		Successful	
	Alg 1	Alg 2	Alg 1	Alg 2	Alg 1	Alg 2	Alg 1	Alg 2	Alg 1	Alg 2
A	7%	20%	4%	6%	18.80	53.40	191	584	191	584
B	22%	25%	30%	32%	0	25.6	0	263	0	263
C	—	34%	—	75%	—	10.47	1309	653	0	0
All	—	26%	—	38%	—	29.82	1500	1500	191	847

Figures 11 - 14 and Table 3 show measurements taken during the burst test. Perhaps the most interesting of all tests, the burst test is the most difficult test. Spreading an entire server's worth of load all coming in at once is a realistic situation that is a common cause of cascading outages, where a failing server sends too much load to the other servers, which then start to fail in turn, making the problem worse and worse.

Here the most clear differences finally show. Algorithm 1 sends the 191 clients to machine A, but then notices machine A getting more loaded and switches over to sending the remaining 1309 clients all to machine C. Machine C cannot handle this load, and almost immediately crashes. The reason for this crash is the kernel running out of memory during high network load, a known problem for this machine (see `http://elinux.org/R-Pi_Troubleshooting#Crashes_occur_with_high_network_load`). It is restarted and comes back online about two thirds into the test, but by then all those connections have already failed.

While it could be said a crash of a system invalidates the test, the crash is directly caused by the high network load. Most machines do not crash because of high network load, but almost all media serving systems will exhibit unwanted or uncontrollable behaviour during overload situations and it can be safely assumed that even if the crash did not happen those clients would not have been served satisfactorily regardless.

Algorithm 2 on the other hand notices that machines A and C both are not very busy while machine B is, and decides to spread the incoming load roughly evenly over A and C, while sending a much smaller portion to B. While all machines stay online and accept the load without crashing, machine C quickly runs out of bandwidth and the approximately 600 clients it is handling receive sub-par delivery.

So, in case of this third test, again neither algorithm can receive a "pass" for serving all clients successfully. Algorithm 1 manages 191 successful clients (13%), while Algorithm 2 manages 847 successful clients (56%).

9 Comparing waves-2 test results

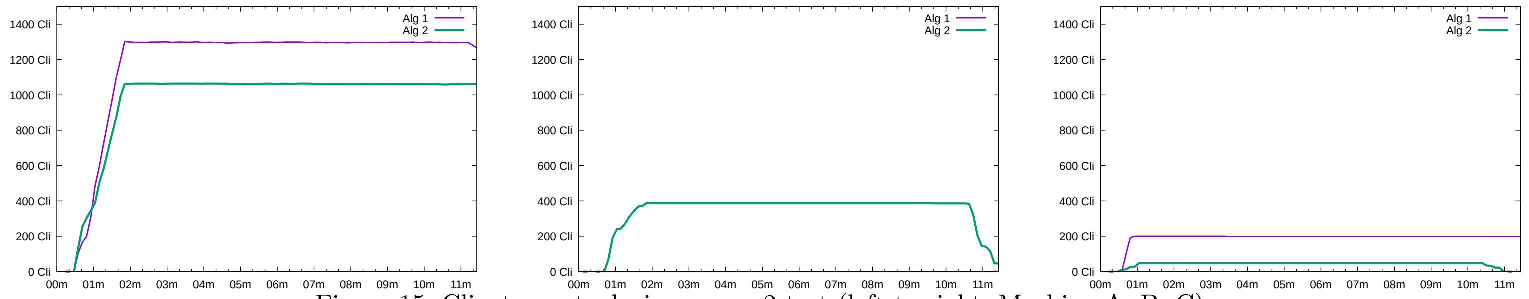

Figure 15: Client counts during waves-2 test (left to right: Machine A, B, C)

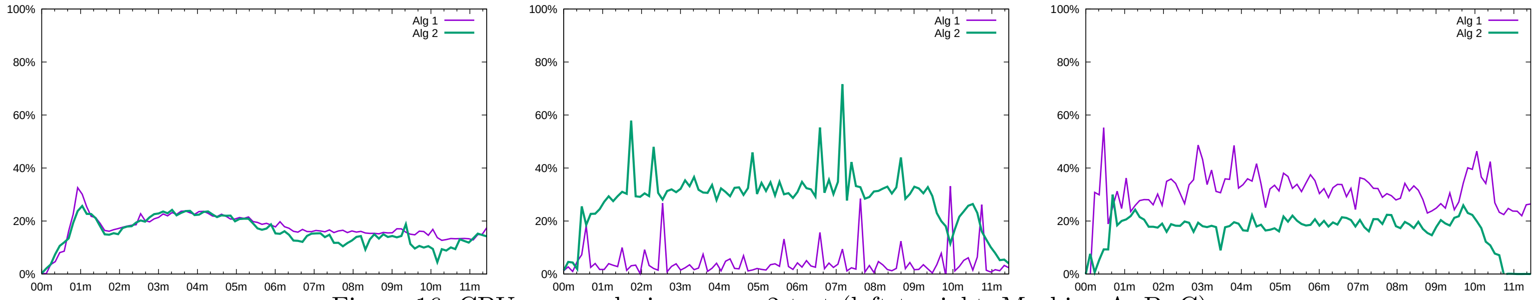

Figure 16: CPU usages during waves-2 test (left to right: Machine A, B, C)

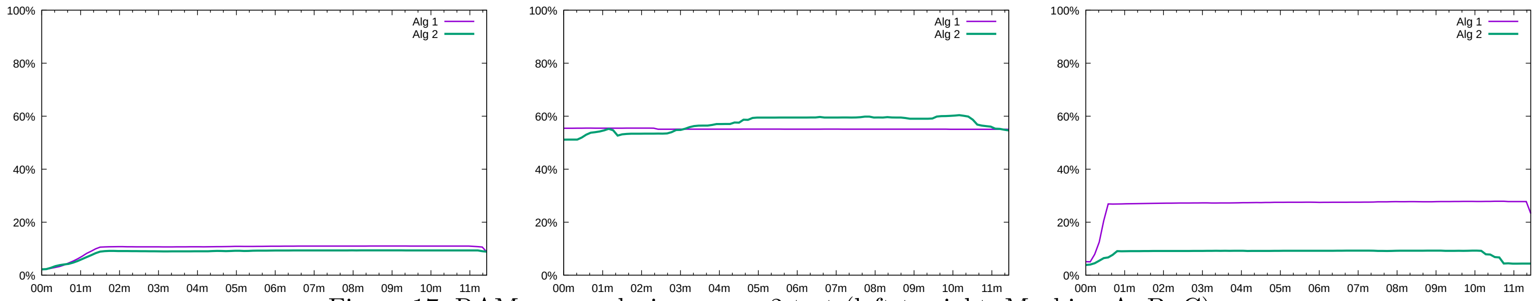

Figure 17: RAM usages during waves-2 test (left to right: Machine A, B, C)

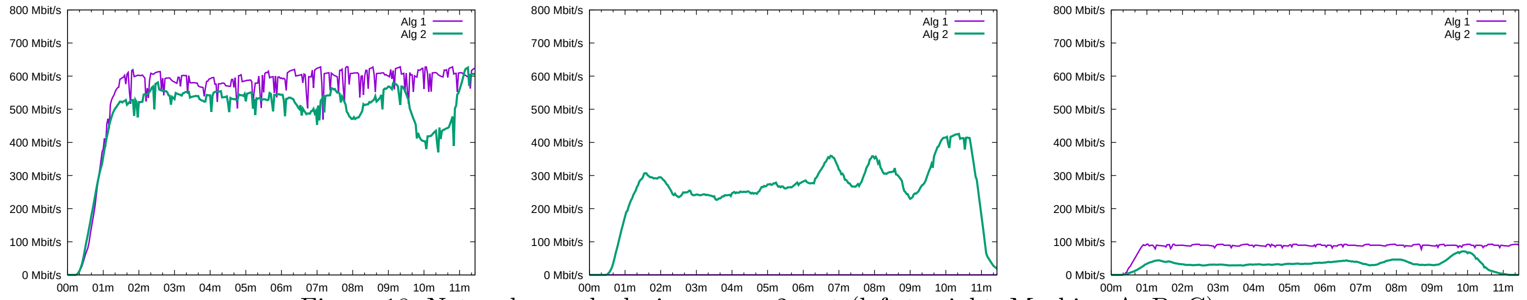

Figure 18: Network speeds during waves-2 test (left to right: Machine A, B, C)

Table 4: Summary of system vitals during waves-2 test

Machine	Avg. CPU		Avg. RAM		Avg. MB/s		Client total		Successful	
	Alg 1	Alg 2	Alg 1	Alg 2	Alg 1	Alg 2	Alg 1	Alg 2	Alg 1	Alg 2
A	17%	16%	10%	9%	70.20	63.50	1300	1064	1300	1064
B	5%	28%	55%	57%	0	33.70	0	387	0	387
C	31%	17%	27%	9%	11.70	4.20	200	49	0	49
All	18%	20%	31%	25%	27.30	33.80	1500	1500	1300	1500

Figures 15 - 18 and Table 4 show measurements taken during the waves-2 test. In this re-do of the waves test, providing both algorithms with the correct bandwidth limit information for machine C, the difference between the algorithms becomes even more apparent.

Algorithm 1 still overshoots the bandwidth limit for machine C, but sends significantly less traffic to it than it did without correct bandwidth limit information. Oddly, it has decided to not send any traffic to machine B, unlike the original run of the waves test where it did. Because machine C was much less overloaded in this do-over, the successful client count is much higher at 1300 successful clients (87%).

Algorithm 2 no longer overshoots machine C's limits, and only sends very little traffic to it at all, sending most to machine A and a sizeable chunk to machine B. The result is a roughly equal utilization of all three machines in terms of CPU percentage. This time around Algorithm 2 does receive a "pass" for serving all clients successfully.

10 Comparing burst-2 test results

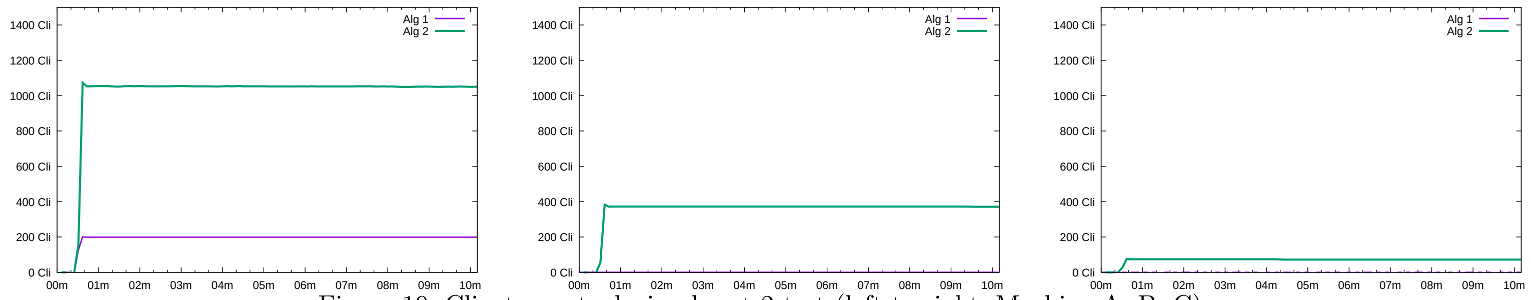

Figure 19: Client counts during burst-2 test (left to right: Machine A, B, C)

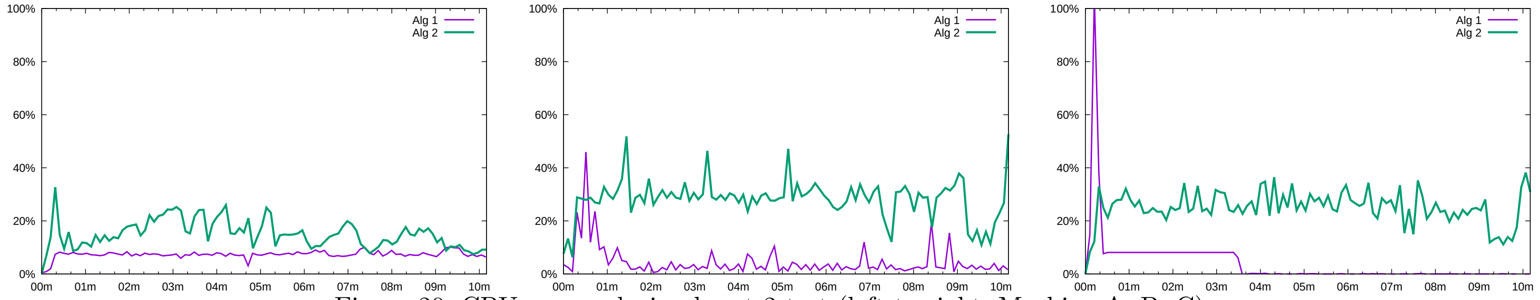

Figure 20: CPU usages during burst-2 test (left to right: Machine A, B, C)

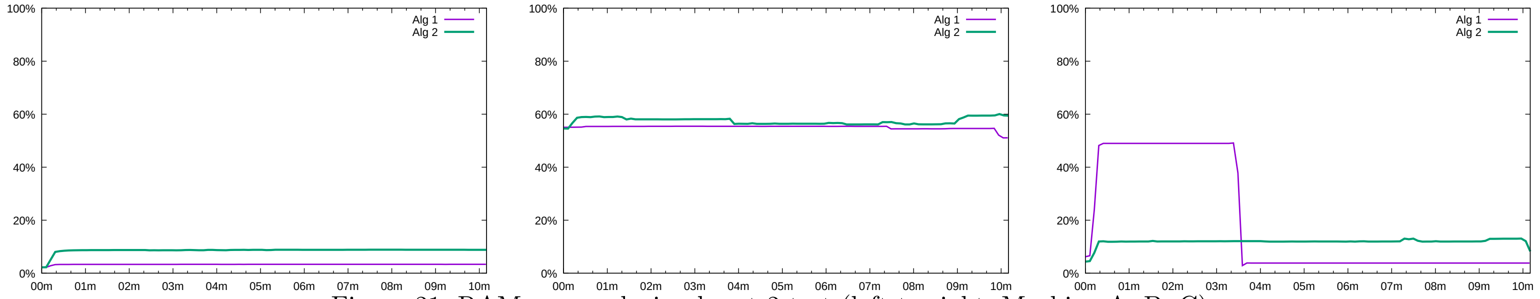

Figure 21: RAM usages during burst-2 test (left to right: Machine A, B, C)

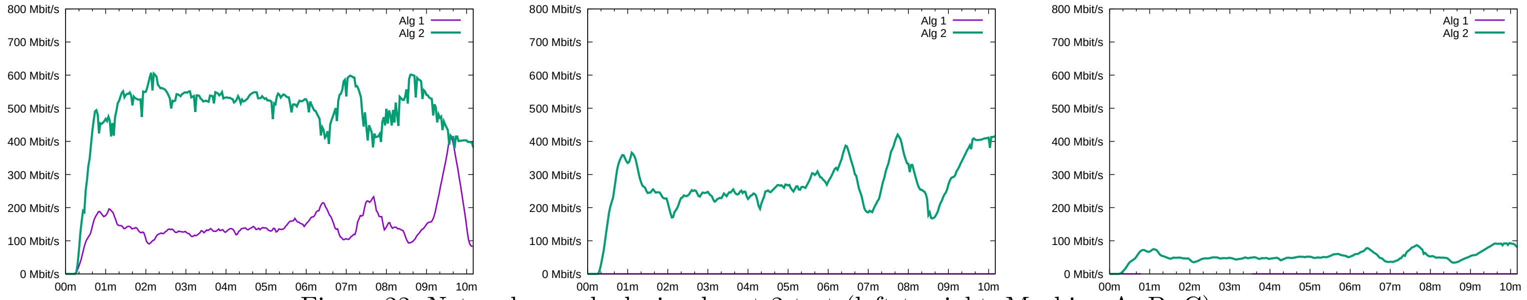

Figure 22: Network speeds during burst-2 test (left to right: Machine A, B, C)

Table 5: Summary of system vitals during burst-2 test

Machine	Avg. CPU		Avg. RAM		Avg. MB/s		Client total		Successful	
	Alg 1	Alg 2	Alg 1	Alg 2	Alg 1	Alg 2	Alg 1	Alg 2	Alg 1	Alg 2
A	7%	15%	3%	9%	19.60	62.60	199	1054	199	1054
B	5%	28%	55%	57%	0	35.60	0	372	0	372
C	—	25%	—	12%	—	7.08	1301	74	0	74
All	—	23%	—	26%	—	35.09	1500	1500	199	1500

Figures 19 - 22 and Table 5 show measurements taken during the burst-2 test. This re-do the burst test, providing both algorithms with the correct bandwidth limit information for machine C, has the most dramatic difference between the two algorithms.

Since Algorithm 1 is reactive, it behaves just like on the first run: almost all traffic is sent to machine C and a small portion to machine A. Again, machine C immediately crashes from the network overload causing the kernel to run out of memory (just like in the first burst test) and only a mere 199 clients has a successful connection (13%).

Algorithm 2 does significantly better this time around: detecting the limited bandwidth of machine C, nearly no traffic at all is sent to it. Instead, machine A takes the brunt of the load while machine B takes the rest of it. The result is roughly equal utilization of the CPU, and another "pass" for serving all clients successfully.

11 Conclusion

For all five tests, Algorithm 2 is the clear winner. In case of the trickle test it achieves delivery with the least CPU cycles on average, while in the first waves and burst tests it is able to mitigate the misconfigured network speed of machine C the most effectively, resulting in the most successful client connections in both cases. During the second waves and burst tests, in was able to successfully serve all clients in both tests while Algorithm 1 was not. Additionally, machine C never crashed under Algorithm 2, while it crashed during both burst tests under Algorithm 1.

Besides the results of the tests, we can also consider how the algorithms would perform in real situations, and particularly in which situations each would perform best and worst.

Algorithm 1 is particularly well-suited to systems and situations where the load on each of the resources generally does not change very quickly. Any quick change can then be seen as a significant event, warranting paying more attention to whatever statistic is changing rapidly. This means the algorithm would be especially apt at reacting to both slow or sudden spikes in use of a particular resource or all resources together. However, it will also react to sudden decrease in resource use and will then start paying more attention to the resource that has just become plentiful. In situations where, for example, CPU slowly builds and then bandwidth suddenly flat lines, the algorithm would start paying most attention to bandwidth and ignore the possibly high CPU statistic. In our tests this didn't appear to be a problem, but one can imagine it could become one.

Algorithm 2 is particularly well-suited to systems and situations where changes may happen faster than the monitoring interval of the servers being balanced. For example in the case of a sudden burst of traffic as was simulated in the burst tests it is clear this algorithm performs remarkably well while Algorithm 1 can only react once at least a single monitoring interval has passed. On the other hand, the cache-encouraging mechanism may cause a too high score to be given to nodes that would otherwise be considered too crowded, and thus may require tuning of this parameter to not be potentially dangerous to the overall health of the network. Additionally, if clients cost percentually more CPU or RAM than bandwidth, Algorithm 2 will not perform as well as it does in situations such as tested, where bandwidth is the main deciding factor.

All things considered, we can say that despite their apparent simplicity, both Algorithms 1 and 2 proved to be adequate to appropriately handle the situations that are common in the average media distribution networks as we know them today. On top of that, Algorithm 2 does handle situations that are often feared and/or the cause of downtime, as they are not adequately handled by current load balancing algorithms.

References

[DFAEA14] Mamadou Tourad Diallo, Frédéric Fieau, Emad Abd-Elrahmane, and Hossam Afifi. Utility-based approach for video service delivery optimization. In *ICSNC 2014: International Conference on Systems and Network Communication*, pages 5–10, 2014.

[ELS+08] Håvard Espeland, Carl Henrik Lunde, Håkon Kvale Stensland, Carsten Griwodz, and Pål Halvorsen. Transparent protocol translation and load balancing on a network processor in a media streaming scenario. In *Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '08, pages 129–130, New York, NY, USA, 2008. ACM.

[GGG15] Punit Gupta, Mayank Kumar Goyal, and Nikhil Gupta. Reliability aware load balancing algorithm for content delivery network. In *Emerging ICT for Bridging the Future-Proceedings of the 49th Annual Convention of the Computer Society of India (CSI) Volume 1*, pages 427–434. Springer, 2015.

[Han12] S. C. Han. Network load-aware user grouping for internet media streaming systems. In *2012 IEEE 10th International Symposium on*

Parallel and Distributed Processing with Applications, pages 262–268, July 2012.

[JLY13] X. Jiang, S. Li, and Y. Yang. Research of load balance algorithm based on resource status for streaming media transmission network. In *Consumer Electronics, Communications and Networks (CECNet), 2013 3rd International Conference on*, pages 503–507, Nov 2013.

[JXY07] Q. Jiang, H. S. Xi, and B. Q. Yin. Dynamic file grouping for load balancing in streaming media clustered server systems. In *2007 International Conference on Information Acquisition*, pages 498–503, July 2007.

[MDW12] J. Ma, G. Ding, and R. Wang. A new load balancing method based on simulated annealing algorithm in streaming media system. In *Wireless Communications, Networking and Mobile Computing (WiCOM), 2012 8th International Conference on*, pages 1–4, Sept 2012.

[QHBC08] Jiang Qi, Xi Hongsheng, Yin Baoqun, and Xu Chenfeng. An event-driven dynamic load balancing strategy for streaming media clustered server systems. In *2008 27th Chinese Control Conference*, pages 678–682, July 2008.