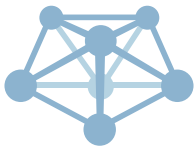


DDVTECH MEDIA SERVER PERFORMANCE MEASUREMENT INITIATIVE

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

Abstract. These performance tests were performed in response to client requests as well as our personal interest in how MistServer performs compared to other widely-used media servers. We have collected sensible performance metrics in a reliable way, repeatable under clearly defined conditions, thus allowing users to compare relative media server performance.

Our initial tests compare performance among three media servers: MistServer 2.0, Wowza Streaming Engine 4.0.0, and Nginx+RTMP module. Nginx is one of the top-performing web servers with streaming capabilities, while Wowza is one of the leading dedicated media servers.

We compare the test results with our theoretical expectations and while these are mostly in line with each other, some of the results are quite unexpected, warranting further research.

1. Research question

When setting up a media system generally one of two approaches is taken — each on opposite sides of the spectrum. Either dedicated media server software is used which is capable of doing advanced processing and control, or a relatively simple web server is used that serves out pre-prepared data without further processing. As you would expect from these setups, the web server approach is more usual where throughput and performance are key while the dedicated approach is more usual in systems that focus on features or specific client needs.

In theory dedicated media servers generally have more features but lesser performance than web-server-based solutions. MistServer, being somewhere in the middle of the two extremes, is expected to perform accordingly.

The question is: how does this theory hold in practice?

Additional to the main theory, we expect some specifics in terms of performance behaviour. Web servers are optimized for many small files, have a very mature caching ecosystem and have been in development for a very long time. Media servers are relatively new, optimized for larger files and have no real caching mechanism of their own, instead relying on existing HTTP caching implementations.

Segmented media delivery methods are HTTP based by design, so those will most likely be optimal on web-server-based systems. On the other hand, dedicated media servers are more suitable for true streaming delivery methods such as RTSP and RTMP. We expect dedicated systems to use more CPU power and memory, because of the additional processing overhead they must incur. Web servers are optimized for dealing with high load and many incoming connections, so most likely those will excel at higher loads.

To fairly assess performance we wanted to look at several areas at the same time. We specifically looks at viewer count, viewer experience, memory/CPU use,

Date: 2014/04/07 - revision 2.

and bandwidth use. Together we believe these give a clear picture of most important factors of media delivery systems, all the way from viewer experience to infrastructure costs.

2. Methodology

We created a simple tool that runs as a low-footprint system daemon, collecting generic system performance metrics. These metrics are for the whole system — not just the media server. We opted for this method because there is no reliable way to collect these metrics for a single server.

The daemon attempts to poll the system usage once per second, but if the system is really busy the daemon may not be able to poll at this interval and the interval may become larger. Each interval, the following metrics are collected:

- Time since measurement start, in whole integer seconds (so larger intervals can be detected and accounted for)
- A one-minute rolling system load average
- Total memory use in bytes, excluding operating system buffers and caches
- The percentage of time since the last interval that the CPU spend in non-idle states
- Total number of bytes uploaded through all network interfaces since measurement start
- Total number of bytes downloaded through all network interfaces since measurement start

Additionally, a client process is defined, for the to-be-measured protocol and stream. This client process is intended to be run from one or multiple separate hardware machines that should have enough bandwidth and processing power available to retrieve the stream as many times as will be tested, simultaneously. The client process simulates a real client connecting to the specified stream over the specified protocol. It analyses the data coming through and performs corruption detection checks. If the data stream at any point becomes corrupted, the client process disconnects and exits without logging anything. Once the stream ends or the client process is killed, it logs the timestamp of the last media packet received. This means the amount of received media data is logged (clean state), or nothing is logged at all (error state).

Each test follows the following steps, in order:

(1) Enable auto-boot for the to-be-measured media server software
(2) Reboot the to-be-measured hardware
(3) Remotely enable the metric collection daemon
(4) Wait 2 seconds
(5) Start X batches of Y client processes on one or more systems that are not measured, waiting one second between batches
(6) Wait for Z seconds
(7) Kill all client processes
(8) Wait 10 seconds
(9) Remotely stop the metric collection daemon and collect the stored measurements.

3. Test hardware

We ran our initial tests on a single hardware configuration. It's specifications are:

- AMD FX(tm)-8120 Eight-Core Processor
- 2X8GB RAM: Kingston KHX1600C10D3/8GX

- ASRock N68-VS3 FX motherboard
- Eminent EM4028 gigabit ethernet PCI-E adapter
- Western Digital Blue WD5000LPVX, 500GB HDD

The installed operating system is Arch Linux, updated to February 1st, 2014. All optional system services and daemons have been disabled, except for ntpd and sshd to make sure system time stays in sync and the system can be remotely administered.

4. The completed measurements

Following the above methodololy, we set up a 0.6Mbit RTMP stream, encoded from the Blender project's Big Buck Bunny movie. For reference, the exact sample clip used can be downloaded at the following URL: `http://dtsc.mistserver.com/example1_low.mp4` On the three configurations described in the introduction, the following tests were run:

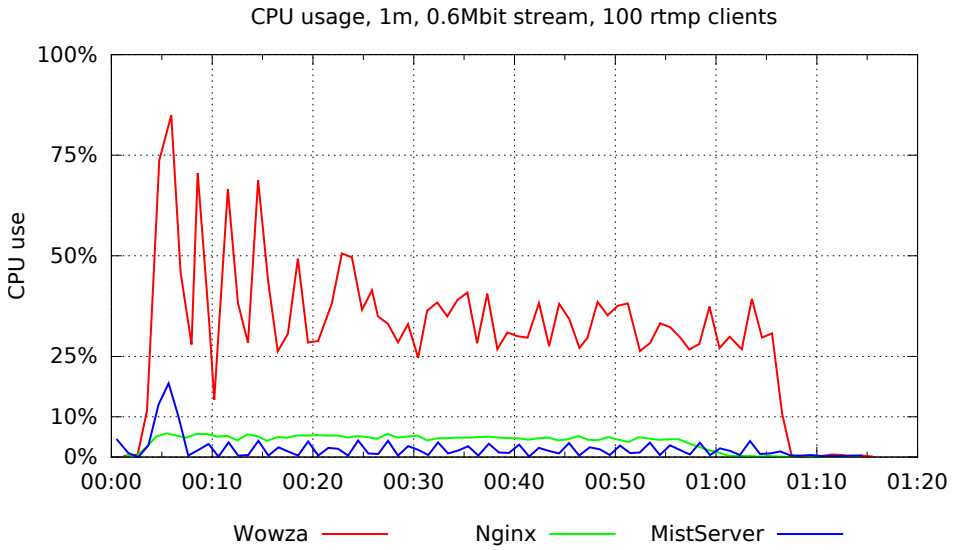
- Run 000: 2 batches of 50 clients, for 60 seconds
- Run 001: 2 batches of 50 clients, for 300 seconds
- Run 002: 2 batches of 50 clients, for 600 seconds
- Run 003: 4 batches of 50 clients, for 60 seconds
- Run 004: 4 batches of 50 clients, for 300 seconds
- Run 005: 4 batches of 50 clients, for 600 seconds
- Run 006: 8 batches of 50 clients, for 60 seconds
- Run 007: 8 batches of 50 clients, for 300 seconds
- Run 008: 8 batches of 50 clients, for 600 seconds
- Run 009: 16 batches of 50 clients, for 60 seconds
- Run 010: 16 batches of 50 clients, for 300 seconds
- Run 011: 16 batches of 50 clients, for 600 seconds
- Run 012: 24 batches of 50 clients, for 60 seconds
- Run 013: 24 batches of 50 clients, for 300 seconds
- Run 014: 24 batches of 50 clients, for 600 seconds

The raw collected data can be retrieved at this URL: `http://releases.ddvtech.com/raw_data_201403.zip`

This archive contains a folder for each media server software, containing all the test runs in the format "run000_100_rtmp_60.csv". The first part is the number of the run, 000 through 014 as listed above. The second part is the total number of clients. The third part is the protocol tested and the last part is the test duration in seconds. Another file with the extension .times is available, containing the measurements taken by the client side and listing how many milliseconds of media data was received. Finally, a third file ends in a dash followed by another number. This number is equal to the contents of the file and lists the count of times that are at least 90% of the test duration. Only these are considered to be successful client connections.

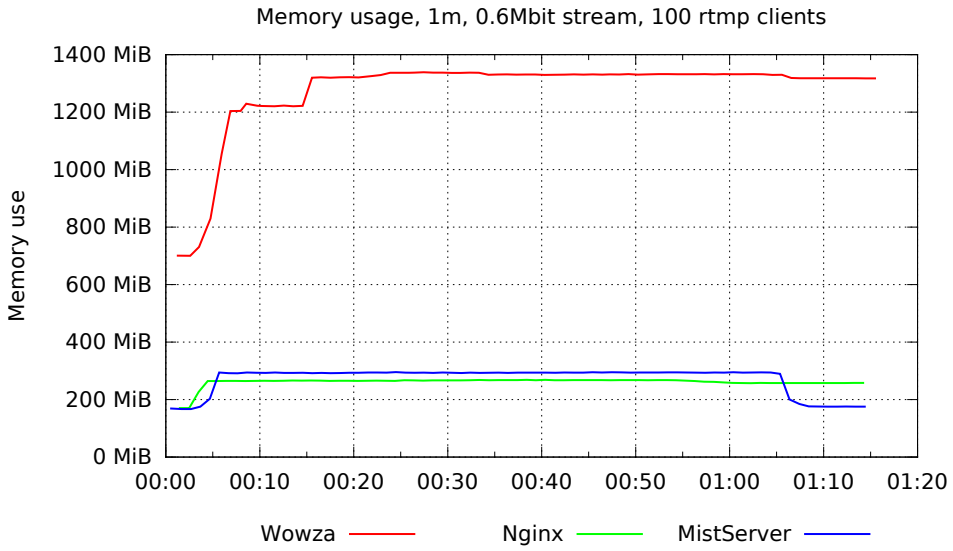
5. Results

We created graphs out of the raw data collected above, to simplify analysing this huge amount of data. Let's go over these and note the more interesting points, starting with the graphs for 100 viewers at one minute test duration.



This is the CPU usage for a one minute test of 100 viewers, for all three tested applications. **Less is better.** Nginx is clearly the more stable of the three, showing no spikes of any kind. This is to be expected, as Nginx is the most generic and optimized but least sophisticated of the three. Both MistServer and Wowza show spiking at similar moments, but MistServer has a much lower average CPU usage than the others.

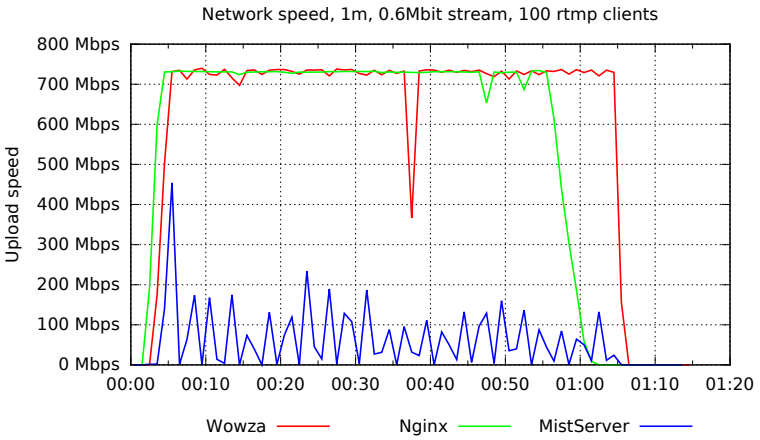
On the whole nothing is very surprising about this graph. Nginx shows the best startup behaviour, smoothly accepting all 100 clients without showing much effort at all. Wowza is most bothered at the beginning, spiking over 80% CPU usage for just 100 clients already.

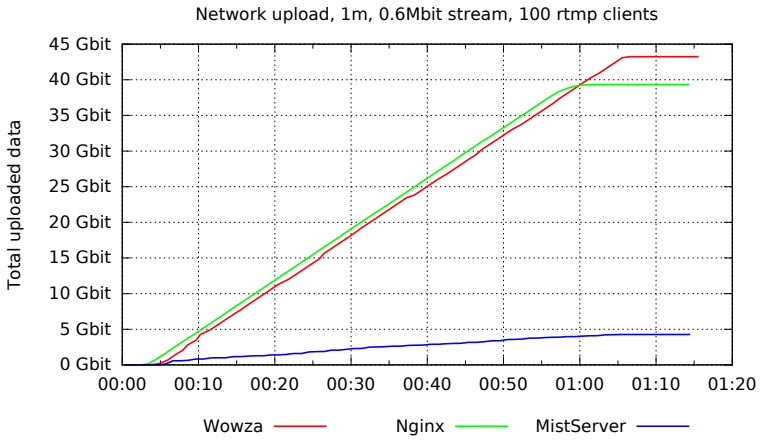


Memory usage. Again, **less is better.** Here we also do not see anything unexpected. Note how MistServer's memory usage rises and drops when clients connect and disconnect, while Nginx' usage rises, but doesn't drop again. This could be considered both an advantage and a disadvantage, depending on who you ask. Keeping the memory in use means the startup time of more incoming clients afterwards will

be shorter, while freeing the memory means it can be used for other tasks running on the system, or will not be billed in cases where memory use is rented.

Wowza clearly uses much more memory than even the others combined — most likely due to the fact that they use Java, which is known for its poor memory management capabilities. This also explains the sudden rise around 15 seconds.





Finally, these last graphs represent the uploaded data. Here too, **less is better.** We also recorded downloaded data, but since uploaded data accounts for almost all of the data usage in all cases, we only graphed the uploaded data here. The left graph shows the average data rate per second, while the right shows the total of data uploaded over time.

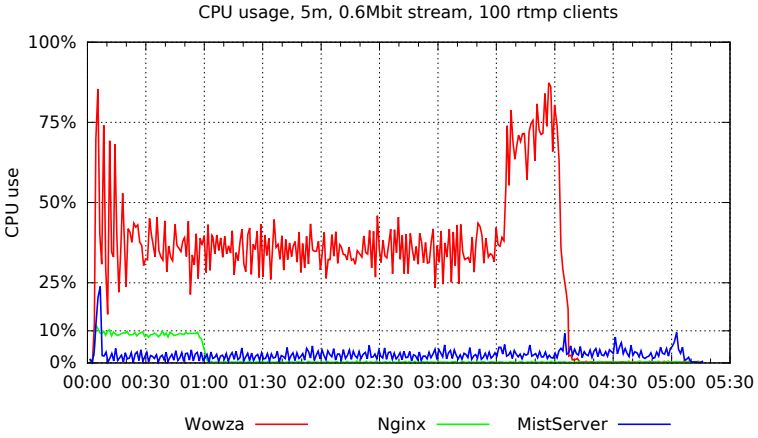
Both Nginx and Wowza fully saturate the gigabit network card (1024Mbps is a gigabit, but this is not a practically reachable speed because of TCP overhead and other factors), sending all 100 viewers the stream as fast as they possibly can.

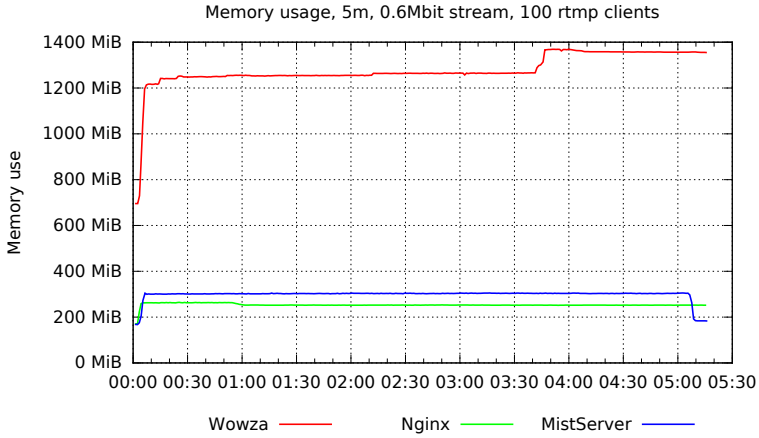
MistServer instead sends out the stream in real-time, generating much less network traffic. Nginx completely drops out just before the minute is over — this is because at that point Nginx has finished sending the complete 10 minute video file. Wowza however has not finished yet at that point and keeps sending data until the clients are forced to disconnect after a minute. Naturally, MistServer's clients have only received one minute of the stream after a minute, and they also keep receiving data until they are forced to disconnect at the end of this test run.

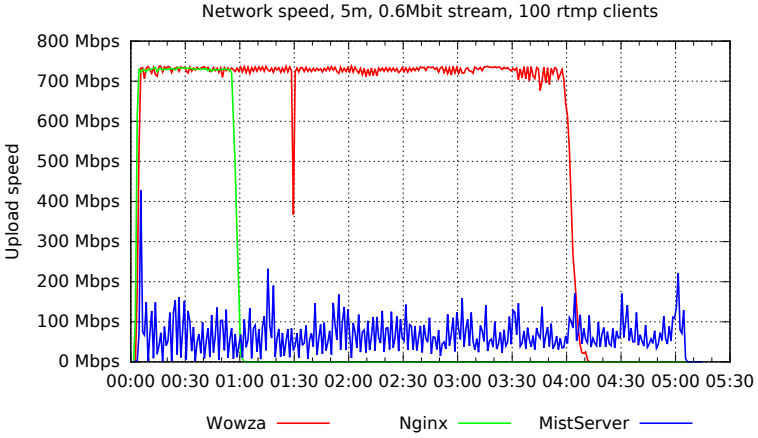
In the end, MistServer uses just under 5 gigabits of bandwidth to send a minute of this stream to 100 viewers, while Nginx and Wowza use around 40 gigabits instead. Quite a large difference, entirely explained by the fact that Nginx and Wowza sent the data as quickly as possible while MistServer sends it at real-time speed.

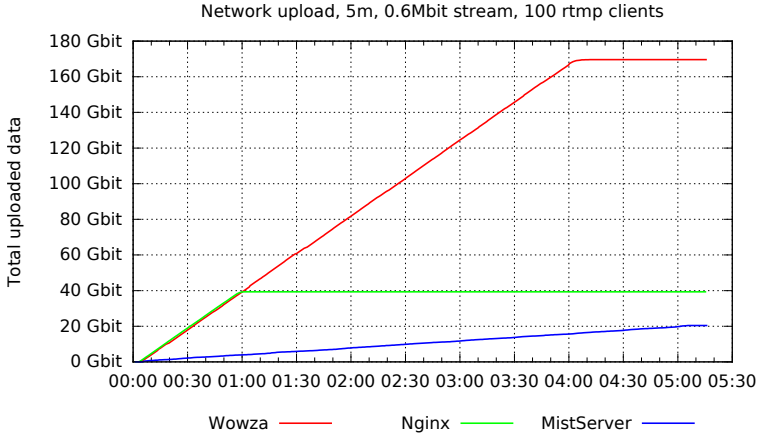
So — what happens when we increase the duration of this test?

Here are the same graphs, for a 5 minute test run. Remember, for all of these **less is better**.





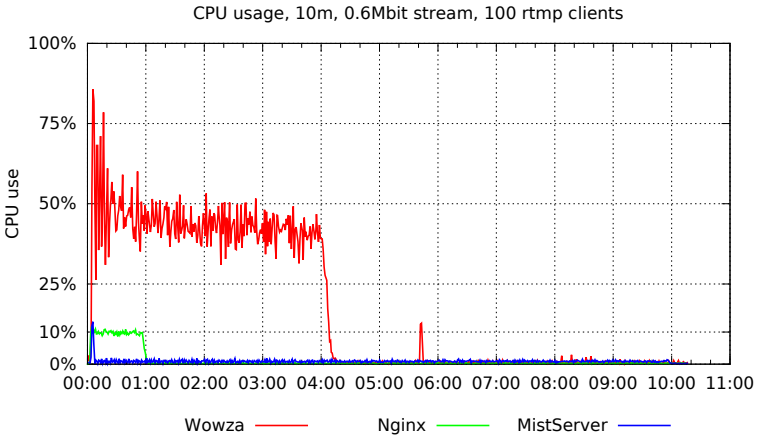


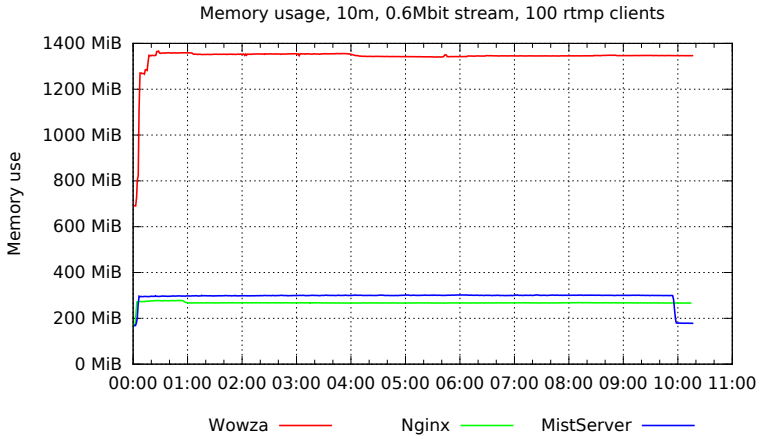


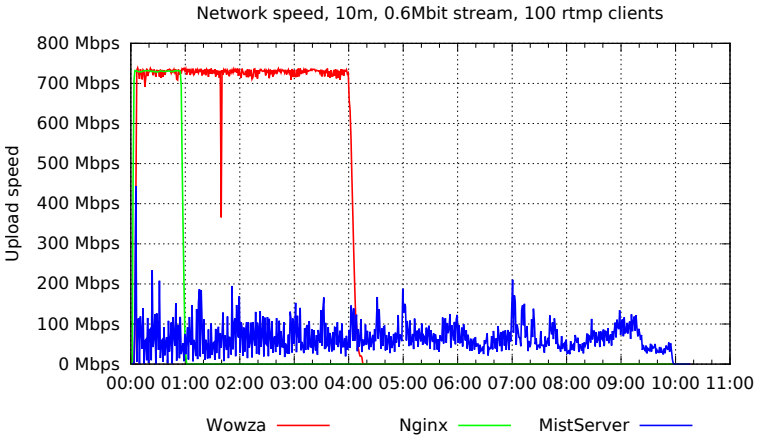
In many ways, these are simply longer versions of the one minute graphs. Worthy of note are the sudden CPU usage spike Wowza shows around 4 minutes into the test, as well as the sudden increase in memory usage as it finally finishes sending the entire 10 minutes to all connected clients.

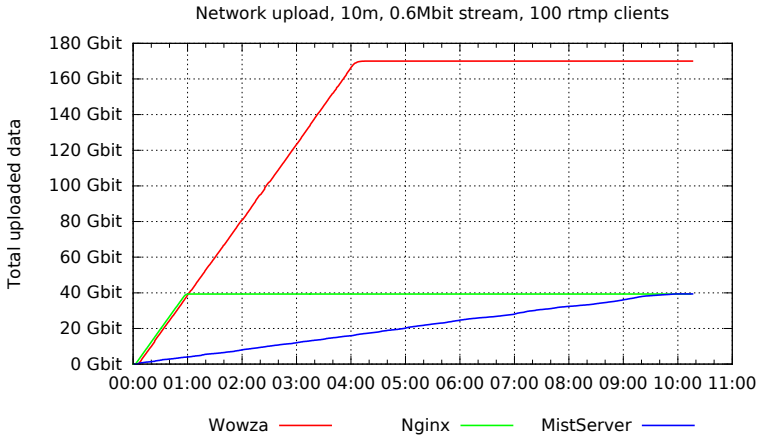
Really odd here is the bandwidth usage. This is a stream that averages around 0.6 Mbps over the whole length of the video. (51 MiB = 408 mbit. Divided by 596 seconds equals 0.68 Mbps, some of which is container overhead.) That would mean it should take roughly 40 gigabits to send 100 clients the entire stream. Nginx clearly shows a near exact match with this theory. MistServer does as well, since halfway into the video it's at 20 gigabits: exactly half of what Nginx used to send the entire stream. However, Wowza uses over 160 gigabits to send 40 gigabits of data. That's approximately 75% overhead! We didn't believe this at first, and double-checked that we were indeed sending out the same stream through Wowza, and confirmed that indeed it was the exact same file. An in-depth check of the streamed data showed us that Wowza sends out several redundant packets, increasing the stream overhead.

Let's move on to the 10 minute run:





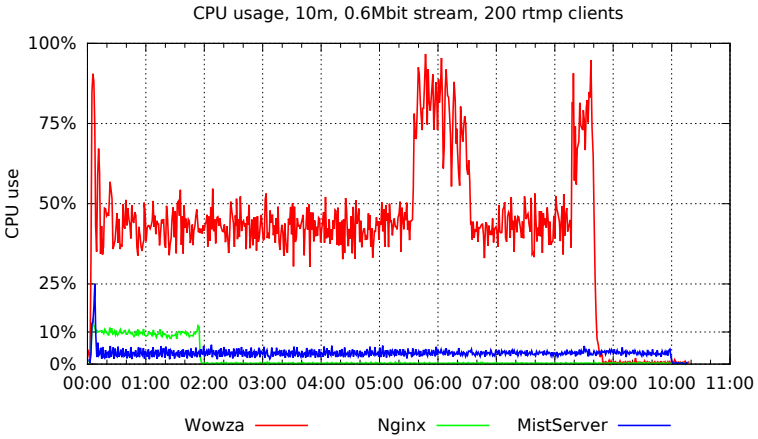


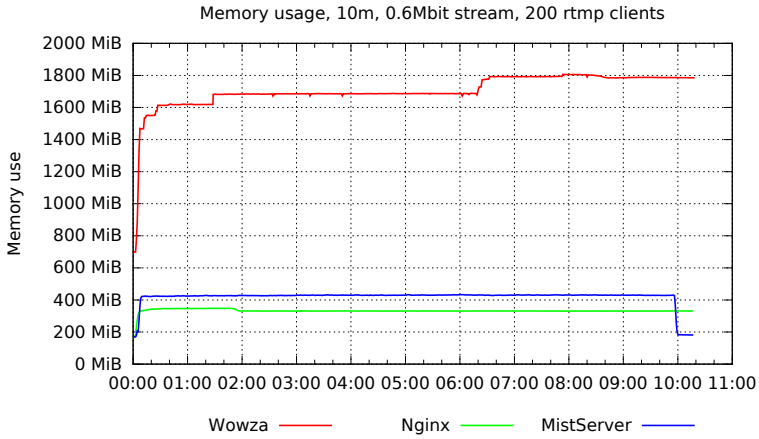


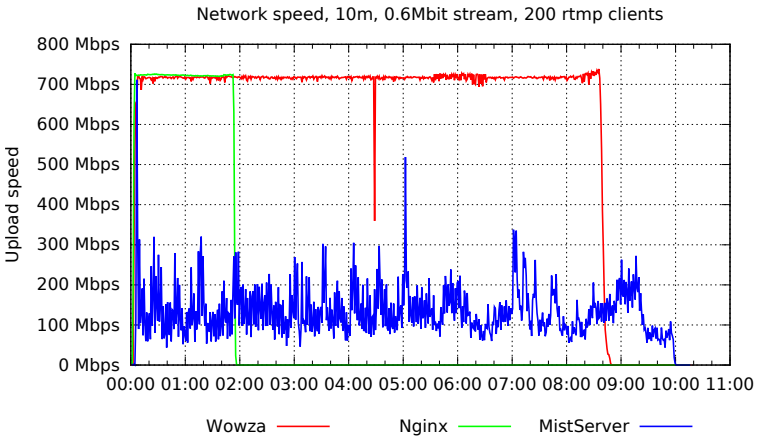
These ten minute graphs really tell the same story as the one and five minute graphs did, but all the way to the end. The scaling causes some of the effects that were already visible in the shorter graphs to be much more pronounced: you can clearly see MistServer's CPU usage being far under Nginx' usage, for example. You can see MistServer's total upload matching Nginx' right at the ten minute mark, as expected. The memory used by Nginx and Wowza is still not freed, even after several minutes of serving zero clients.

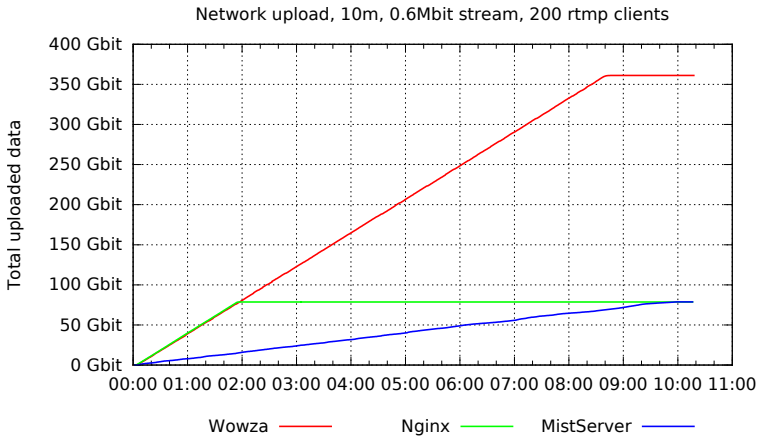
Something that wasn't shown in any graph, but is worth noting anyway: in all of the above cases, all media servers delivered 90% or more of the testing time in media data to all 100 clients. In other words, they all pass our measure for stream quality completely. *For simplicity and to account for measuring error, we'll call a run "perfect" if no more than 5 clients fail to receive 90% or more of the testing time.*

Let's see what happens when we double the number of clients to 200. The effects between one, five and ten minutes are mostly the same, so to save space here are only the graphs for 10 minutes:





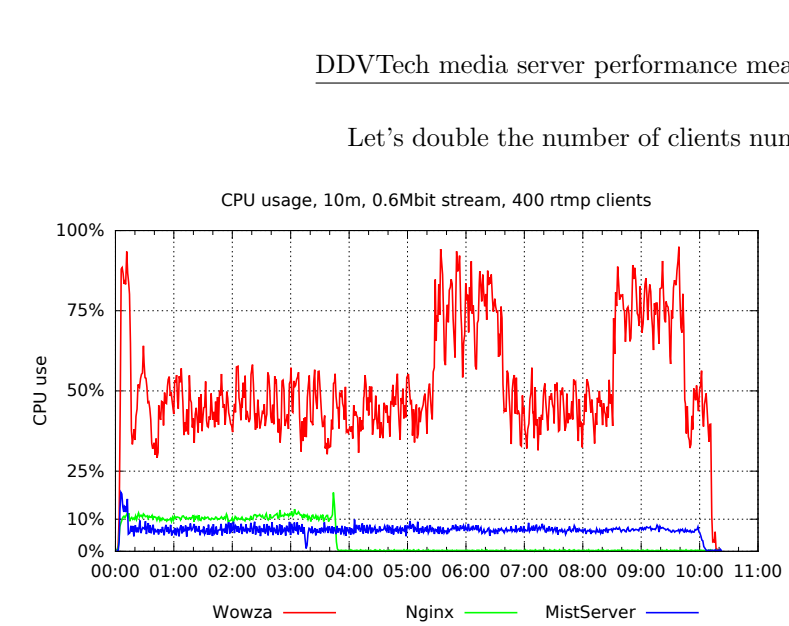


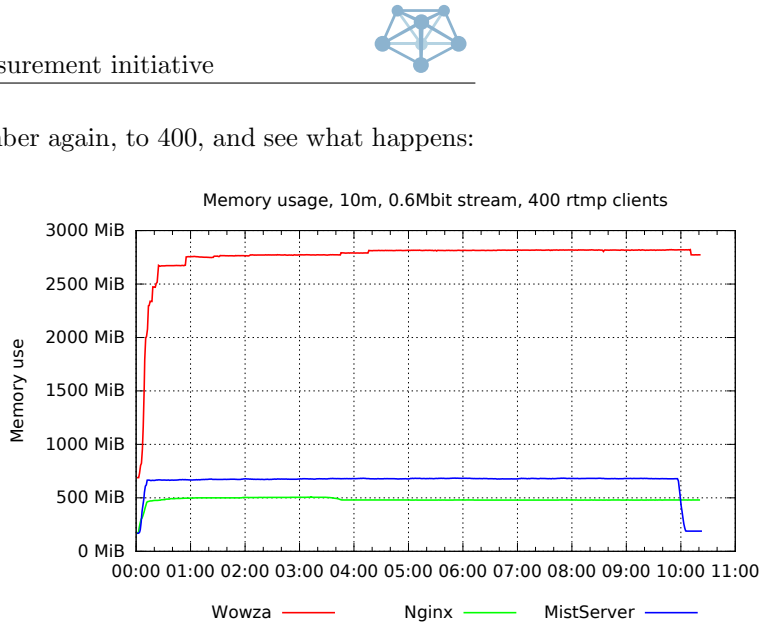


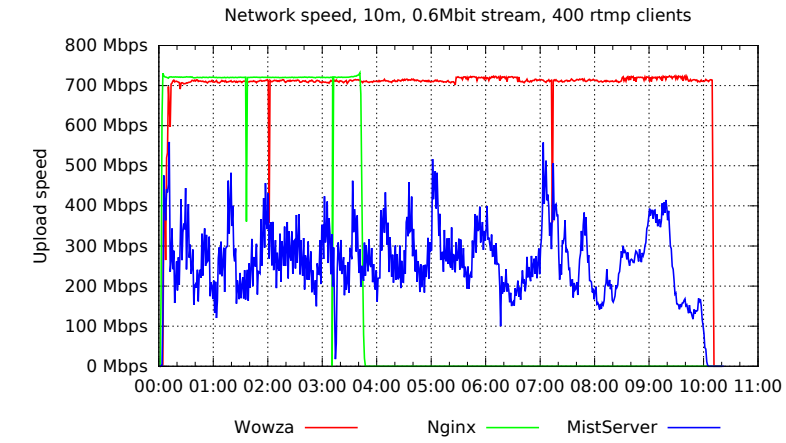
First of all, let's note that again all runs for all servers got a perfect count for our stream quality measure: all 200 clients served successfully.

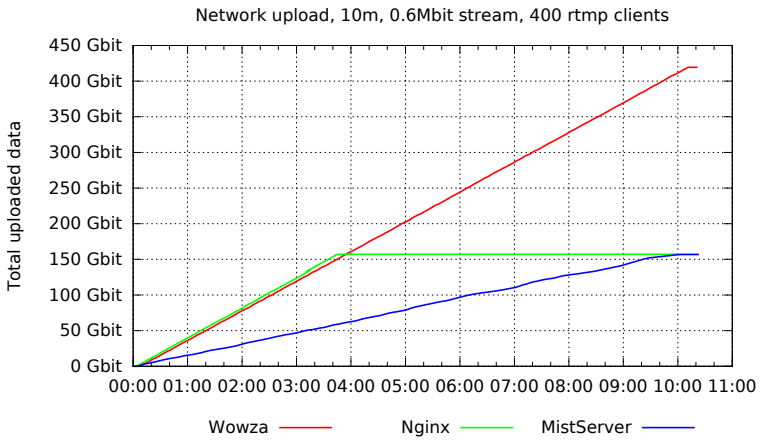
Most things here are simply doubled. Wowza and Nginx both still saturate the entire available bandwidth (as expected), but they both take twice as long to completely send the 200 clients the entire stream (again, no surprise there). MistServer's upload speed has doubled, compared to 100 clients. The total amounts of bandwidth used at the end are exactly doubled for all three. In terms of memory usage, both MistServer and Nginx roughly doubled their use, while Wowza only added about 400 megabytes to its previous 1400 megabytes. It still uses more memory even when idle than either MistServer or Nginx use while they are serving 200 clients, though.

It's a little hard to tell because the numbers are so small, but MistServer's CPU usage has roughly doubled compared to the 100 clients run. Nginx and Wowza's CPU usage have stayed nearly the same. They both take twice as long as before, causing the total usage over time to have approximately doubled for all three.





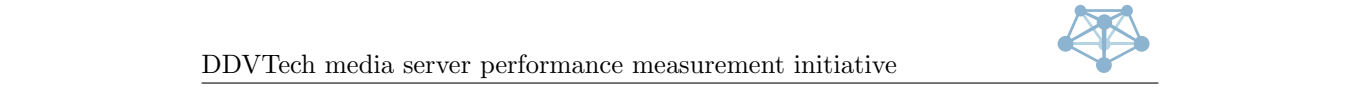




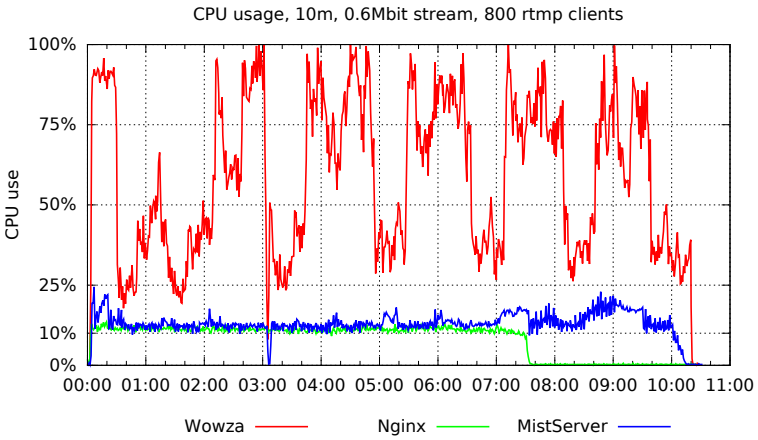
Let's double the number of clients number again, to 400, and see what happens:

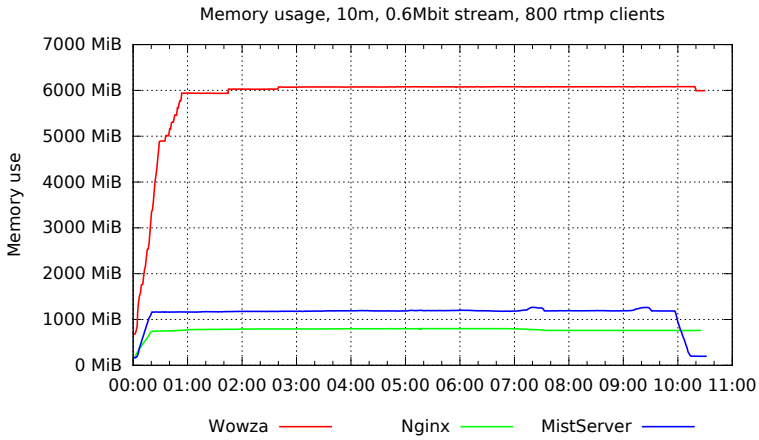
The quality measure counts are no longer perfect at 400 clients, already. Nginx and MistServer got a perfect in all cases, but Wowza only managed 166, 0 and 0 successful connections out of 400 for the 1, 5 and 10 minute runs, respectively.

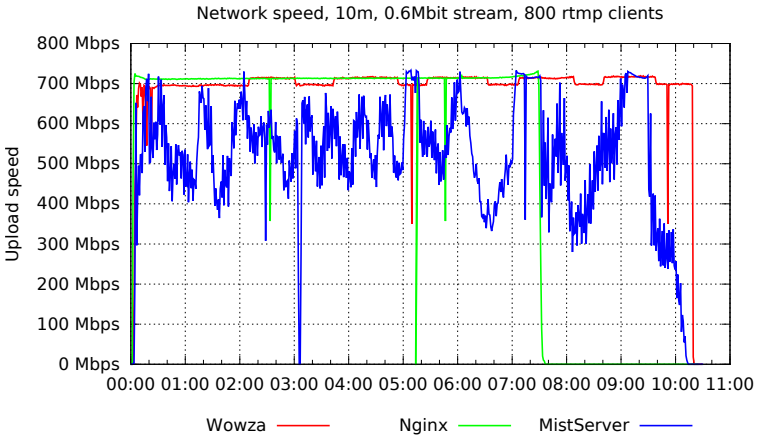
Other than that, exactly what you'd expect by looking at the 100 and 200 client runs has happened: bandwidth usage again doubled for all three in the exact same way. Both Nginx and MistServer doubled their CPU and memory usage in the same way again, but Wowza shows slightly less than double CPU use (most likely due to failing to serve all 400 clients). Wowza's memory usage did double in the same way, again, adding approximately another 800 megabytes on top of the previous 1800.

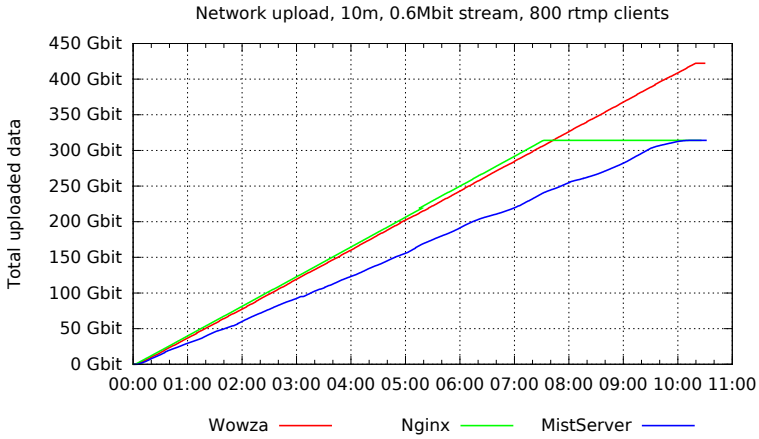


Let's double that client count again, to 800:





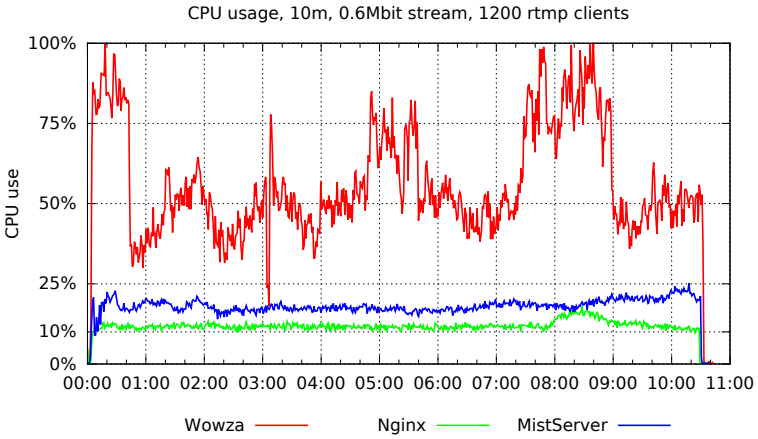


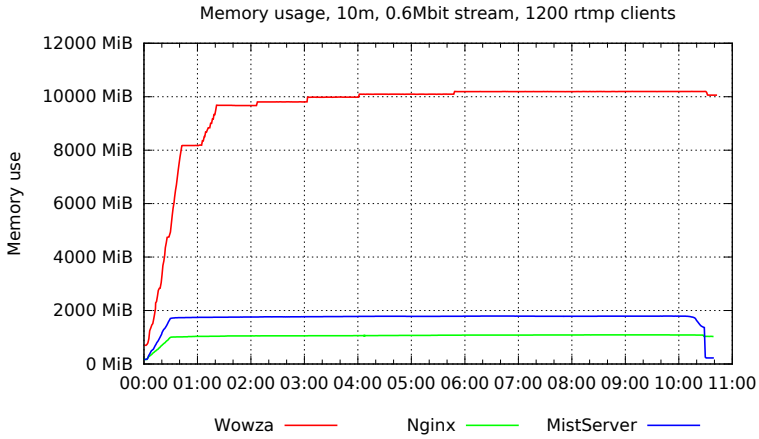


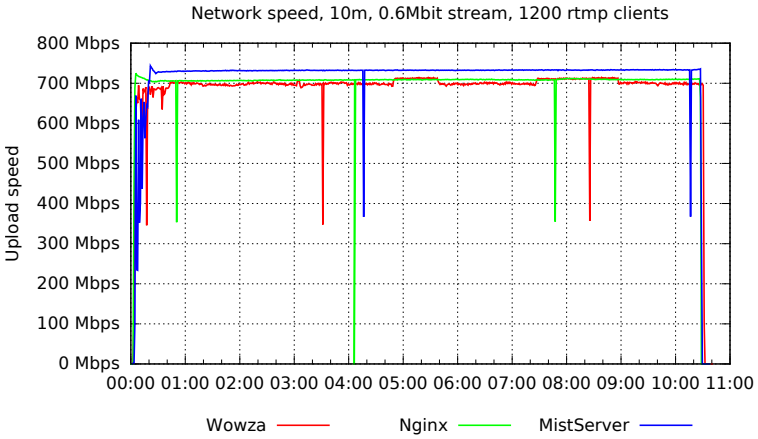
Nginx and MistServer still both get a perfect quality measure count here. Wowza got a 167 on the one minute run (almost exactly the same as at 400 clients), and again a zero on the other runs.

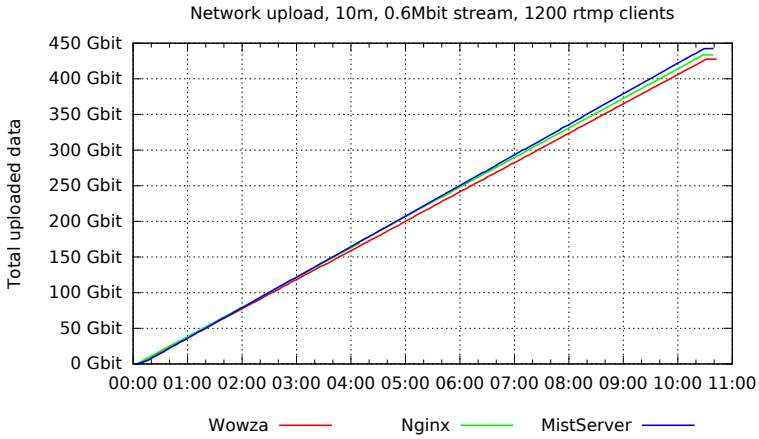
Roughly the same things happened again, with some exceptions: Wowza now spikes repeatedly to 100% CPU usage, and more than doubled its memory use to a whopping 6000 megabytes total. Additionally, MistServer now saturates the network card momentarily where the test video has the highest bitrates. This seems to be "sweet spot" where Nginx and MistServer are at near-equal performance on almost all measured fields.

We would double the number of clients again to 1600, but since this is an 0.6 Mbit stream and the network card is saturated at roughly 700Mbps, we theoretically can only serve about 1165 clients reliably. Let's attempt to push past that maximum just a little bit and see what happens at 1200 clients:









Let's talk quality measure counts again. We're not expecting perfect counts anywhere here, since this run is theoretically impossible. As a reminder, we consider receiving at least 90% of the test duration in media a successful connection, and no more than 5 connections may be unsuccessful for a run to be considered perfect. Nginx got 1079, 858 and 809 out of 1200. MistServer got 1182, 728 and 818. Wowza got 343, 0 and 0.

All the previously established patterns in terms of CPU and memory usage are still visible here. Bandwidth is now fully saturated by all servers for the full ten minutes, exactly as expected.

This makes the quality measure counts the main point of discussion. Since most high-bitrate moments appear later into the video, MistServer has an advantage in the first minute and gets a near-perfect 1182, compared to Nginx' 1079. At five minutes, MistServer can't keep up anymore as the bitrate needed to sent the stream in real-time pushes past the limit of the network card, and it drops to 728 while Nginx almost maintains its count at 858. Of course both applications run into limits here, but Nginx got some of the viewers a head start while it was still able to and MistServer did not, causing MistServer's count to drop more quickly. The ten

minute counts turn back into MistServer's favor as bitrates drop again, allowing MistServer to "catch up" to 818 and Nginx drops further to 809.

The above data is also summerized in some easily readable tables, below.

Table 1. Successful client counts

Clients	400			800			1200		
Time	1m	5m	10m	1m	5m	10m	1m	5m	10m
Wowza	166	0	0	167	0	0	343	0	0
MistServer	400	400	398	795	798	798	1182	728	818
Nginx	400	400	400	800	800	800	1079	858	809

Table 2. Total bandwidth used per client in KiB

Time	1m	5m	10m
Wowza	51943	221935	221935
MistServer	5287	26490	51583
Nginx	51518	51549	51549

Table 3. Memory usage per client in MiB

Client count	Wowza	MistServer	Nginx
100	5.49	1.34	0.95
200	5.91	1.30	0.80
400	5.23	1.27	0.77
800	6.56	1.27	0.79
1200	7.52	1.34	0.75
Average	6.14	1.31	0.81

Table 4. CPU usage per client in percentage

Client count	Wowza	MistServer	Nginx
100	0.170	0.009	0.011
200	0.219	0.017	0.010
400	0.106	0.017	0.010
800	0.067	0.017	0.010
1200	0.042	0.015	0.010

6. Analysis

What do all these results mean with regards to our theories and research question?

The theory that dedicated media servers, which are more feature-rich than web servers, exhibit lower throughput performance than web-server-based solutions seems to hold up pretty well. The theory that the tested protocol (RTMP) would perform worse on a web-server-based system doesn't seem to hold at all.

MistServer is doing better than we expected it to — its performance is very similar to Nginx' (even exceeding it in some of the test cases), with a scaling curve that is dissimilar to any of the other products. Both MistServer and Nginx seem to scale almost linearly in memory, while Wowza uses a very unpredictable amount of memory. Both MistServer and Wowza scale almost linearly in CPU time while Nginx has a very steady CPU usage. Both Nginx and Wowza scale equally with

regards to bandwidth used, trying to send data as fast as possible while MistServer only sends what is needed to have a working stream.

Nginx was the best performer of the tested products in most of the conditions, only being beaten by MistServer in some of them. This is most likely due to the lack of extra features in Nginx, allowing it's implementation to stay minimal and efficient compared to the others.

Our predictions about CPU and memory usage do hold true - specifically that dedicated media servers use more CPU and memory than web-server-based systems. The exception, in some cases, is MistServer, which uses less CPU than Nginx to complete the delivery especially for the lower viewer counts and durations. For memory the prediction was always correct, though MistServer's memory usage was only slightly higher than Nginx' while Wowza's usage was much higher.

The theory that Nginx would excel at higher loads is partially true: it is able to accept many incoming connections more smoothly than the others. However, during the 1200-client overload testcase, MistServer was able to consistently serve a larger number of viewers for two out of three scenarios (the one- and ten-minute tests), while Nginx won out in the five-minute test scenario. This is most likely a side-effect of the streaming behaviour of Nginx, where it tries to send as much data as possible as fast as possible. When the connection is being oversaturated like this, Nginx tends to serve some of the clients more quickly than others. This causes a bandwidth shortage for the remaining clients, making them unable to receive the stream at real time speed or higher.

The bandwidth usage measurements were a bit of an afterthought: we weren't expecting to see anything strange there. However, we were quite surprised at the diversity of behaviour here. MistServer was the only server not trying to maximize use of the data connection, and the amount of overhead incurred when using Wowza is odd to say the least.

7. Conclusion

Summerizing the above analysis, most of the test results are nicely in line with our theories. It clearly shows that not all media servers are created equal, even when attempting to deliver the same content over the same delivery method. Compared to what we expected, Nginx is much more adept at handling true streaming protocols and MistServer performs better in overload conditions.

Naturally we are simplifying many factors here — possibly viewer experience is the most simplified. Additionally, we were not expecting such varying results in bandwidth use and we would have preferred to not be limited to just three server software products.

We believe these results warrant further research, perhaps additional metrics would reveal further differentiating factors as well. We're encouraging anyone with the means to do so to repeat these measurements and share them.