SmartFoxServer 2X
Running 100K concurrent users

January 2015

Author

Marco Lapi
The gotoAndPlay() Team

Introduction

In this short article we will describe a performance stress test that was run on a single **SmartFoxServer 2X** instance.

The objective of the test was to assess the ability of the server to scale in the tens of thousands of users, possibly reaching and surpassing the 100 000 concurrent users (CCU) mark.

We measured the amount of consumed resoruces (CPU, RAM, Network) and the overall scalability, looking for possible bottlenecks or uneveness of the performance.

Testing approach

For this benchmark we have designed a simple login Extension that groups every client in a Room of random size, between 3 and 8 users.

When a new client logs in the Extension searches for a free Room and either joins the User in the Room just found, or creates a new one and then joins the client.

After this step each client will start sending messages at a rate of 1 every 4 seconds, which is a medium-high message rate for any Lobby, chat or turn-based game.

Hardware setup

We run the test in the Rackspace Cloud infrastructure using 3x “OnMetal” machines which are dedicated, physical servers with no active virtualization.

We built a 2 node cluster for the stress testing tool (codename BitSmasher) to ensure we could reach at least 100K connections without running into ephemeral ports issues.

The problem is that a client application can open a max of 50-60K TCP connections towards the same server host, depending on the OS settings, after which it will run out of client port numbers.

By running two nodes with approximately 50K clients each, we made sure to avoid this hurdle.

The 3 servers used the same hardware configuration:

- 1x CPU Intel Xeon E5-2680, 10 cores, 2.8Ghz
- 32GB RAM
- 10 Gbit fast ethernet connection
- Linux CentOS 6.5
- Linux *ulimit* configured to 150000

SmartFoxServer configuration

SmartFoxServer 2X 2.9.x was used for the test with a minimal set of JVM tweaks to exploit the larger memory available in the system.

We used the following conservative settings:

- -Xms4G -Xmx8G to ensure a large heap size
- 64 threads for the System Controller
- 16 thread fot the Extension Controller
- 8 threads for the SocketWriter thread pool

Everything else was from the standard configuration.

Stress tester configuration

The stress client tool run on two nodes with the following parameters for the test:

- Client generation: 40ms (x2 instances = 2 users every 40ms) = 50 users/sec
- Max clients: 50 000 CCU (x2 = 100K CCU)
- Public message speed = 4 seconds

When all the clients are generated there are 100 000 requests every 4 seconds = **25K req/sec.**

Since the requests are public messages and the average Room size is 5.5 = (3+8/2) the total amount of broadcast messages is 25K * 5.5 = **137K responses / sec**.

This generated a pretty high traffic:

- **Incoming average**: ~150Mbit/sec (min: 98, max 195)
- **Outgoing average**: ~444Mbit/sec (min: 201, max: 781)

Running the test

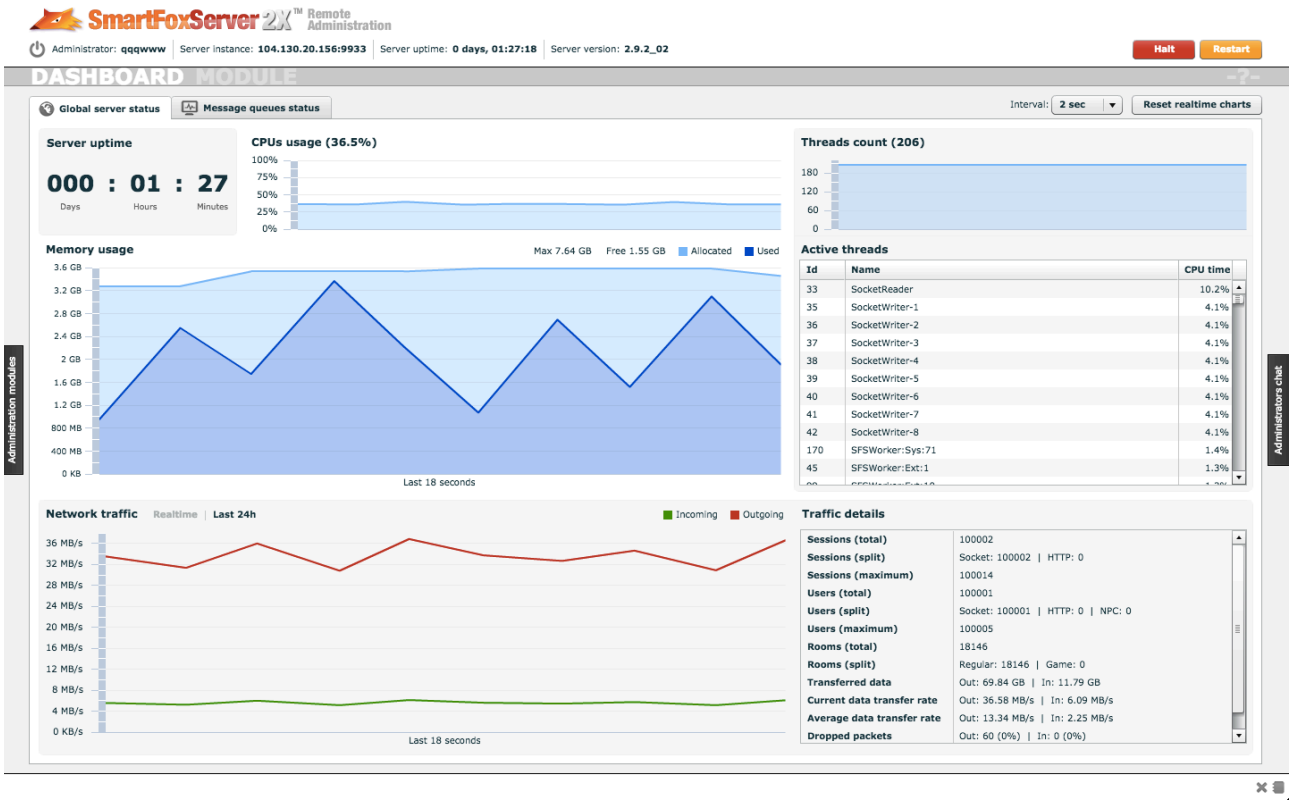
In order for all 100K clients to be generated it took ~35 minutes, during which over 18100 Rooms were created to accomodate all the users.

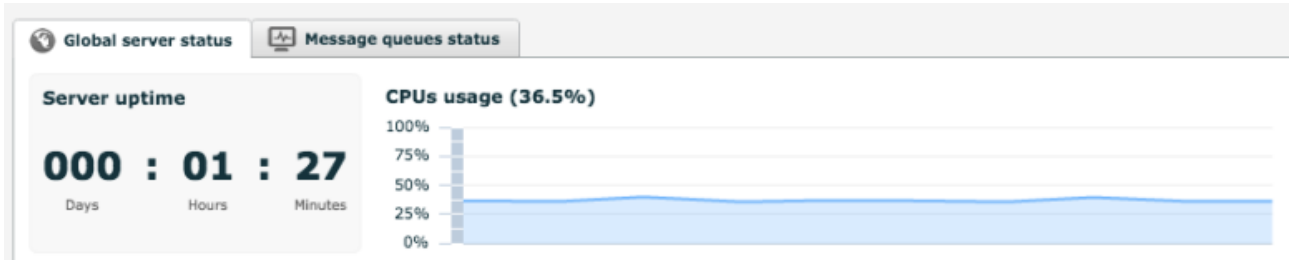
We saw the CPU consumption rarely spike over 50% and then settling at ~37% after all clients where joined.

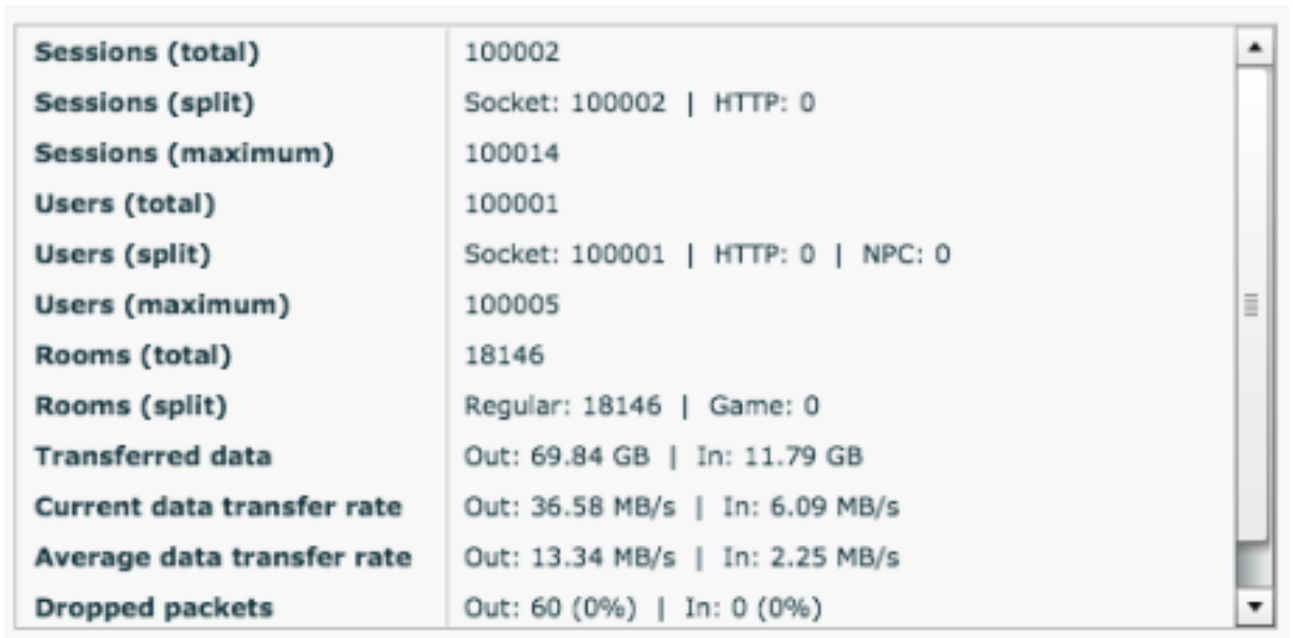
Memory consumption didn't event reach the minimum heap size configured for the JVM (4GB) leaving plenty more room for other data.

The test was left running for ~12 hours then we shut down all clients at a speed of 100/sec. The test run stable for all the time.

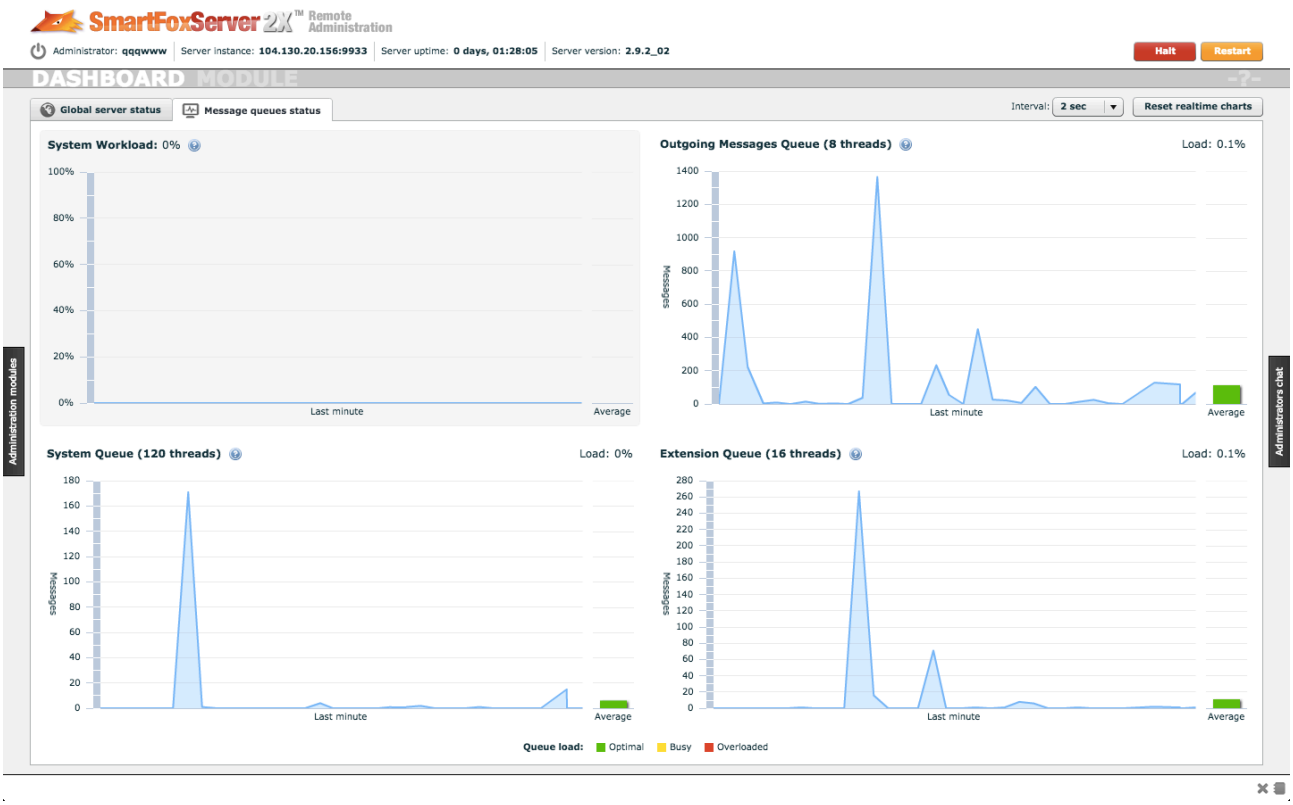
Runtime Dashboard view







Server's queues



Conclusion

Embarking in a stress test for 100K CCU was relatively a simple operation. We only needed to tweak the JVM heap and the OS ulimit parameter.

We didn't encounter any scalability issues, and recorded a linear consumption of resources (CPU, RAM, Network) without "bumps". The single CPU, 10-core machine appears to be generally oversized for this task, especially on the RAM department, which is a bonus since it offers a lot of room for extra traffic, more complex server side logic or the ability to host other processes, such as a database engine.