SmartFoxServer 2X Performance And Scalability White Paper

March 2012

Author

Marco Lapi
The gotoAndPlay() Team

Introduction

Performance and scalability are among the most common topics that customers want to learn about when evaluating a server product for the project they have in mind. Depending on the type of multiplayer application in development, there can be very different requisites that developers might look for.

In this white paper we are going to address most of the common questions that we often get asked about scalability, performance and server side architecture.

Before diving in the benchmark numbers it is important to examine the many variables at play and learn how to interpret them correctly. This will enable the developer to evaluate the results presented in this white paper and more importantly to analyze their own custom tests. In fact one of the most crucial phases in the development of multiplayer applications is the pre-production benchmark testing aimed at finding performance glitches and bottlenecks.

» The common questions

Among the usual questions we receive there are:

- How many concurrent clients SmartFoxServer can handle?
- Will my hardware be capable of handling <AnyNumber> of concurrent users?
- Will SmartFoxServer scale if we will reach <AnyNumber> of users?
- What hardware configuration would you recommend for our project?

It is usually difficult to provide a realistic answer to these questions because they heavily depend on a number of important variables: the type of application (MMO, realtime game, turn-based game, etc), hardware resources, average message rate, complexity of the game logic, efficiency and scalability of the custom server code, proper configuration of the server, available network bandwidth, etc...

The variables at play

One of the most discussed subject in multiplayer games is the amount of concurrent users (often abbreviated as CCU) which expresses the number of players simultaneously interacting with the server.

This parameter is usually not fully understood and it can bear very little meaning when used outside of an application context. When confronted with claims of thousands of hundreds or even millions of CCUs we should always look for additional details, the lack of which is usually indicative of commercial hype.

In fact it is very simple to declare extraordinary feats in terms of CCU when there is few or no elements at all describing the context in which these results where obtained. The first question one should ask is "*one million CCUs doing what?*".

Reliable benchmarks should describe the following parameters for each test.

- **CCU**: total number of users (real or automated) participating in the test at the same time
- **Message rate**: the amount of messages per second exchanged between a single client and the server. It can be expressed either as a constant value or an average in case the rate is variable.
- **Message size**: the average size of the messages sent in the test. It can be of lesser interest if the next parameter is included.
- **Bandwidth usage**: it is a fundamental value, probably the most interesting one because it shows the mass of data that the server can handle and the relative costs in terms of CPU and Memory consumption.
- **Process resource usage**: it shows the amount of **CPU** and **Memory** utilized during the test.
- **Hardware specifications**: a description of the server machine hosting the server including CPU type, RAM, system architecture (32/64 bit), OS type, network card, and any other setting that might be relevant.
- **Test description:** a detailed description of the benchmark test, the objective of the test (e.g. show the performance of broadcast-type messages), the length of the test (was it run for an hour? a day? a month?).

» The CCU problem

In the course of the past 8+ years we have been asked many times how SmartFoxServer would compare to ProductX or ProductY claiming to handle 100K CCU, 500K CCU, 1M CCU, etc. Responding to these requests is usually complicated because we know very little about ProductX and generally the CCU parameter alone is not significant without the extra information about the other parameters we have just mentioned.

A real world example came from a potential customer asking how SmartFoxServer would compare with ProductZ, claiming 300,000 CCU in their benchmarks. While this result is remarkable and certainly capable of attracting the attention, on further investigation it easily found that the test was just running a message rate of 60K msg/sec and a mere bandwidth occupation of ~30Mbit.

Doing the simple math shows that users are just sending 1 message every ~5 seconds and the bandwidth usage is very indicative of the relatively low traffic generated by this test. This is comparable to 1-2000 CCU playing an ordinary multiplayer action game, where the typical message rate is ~10-20msg/sec.

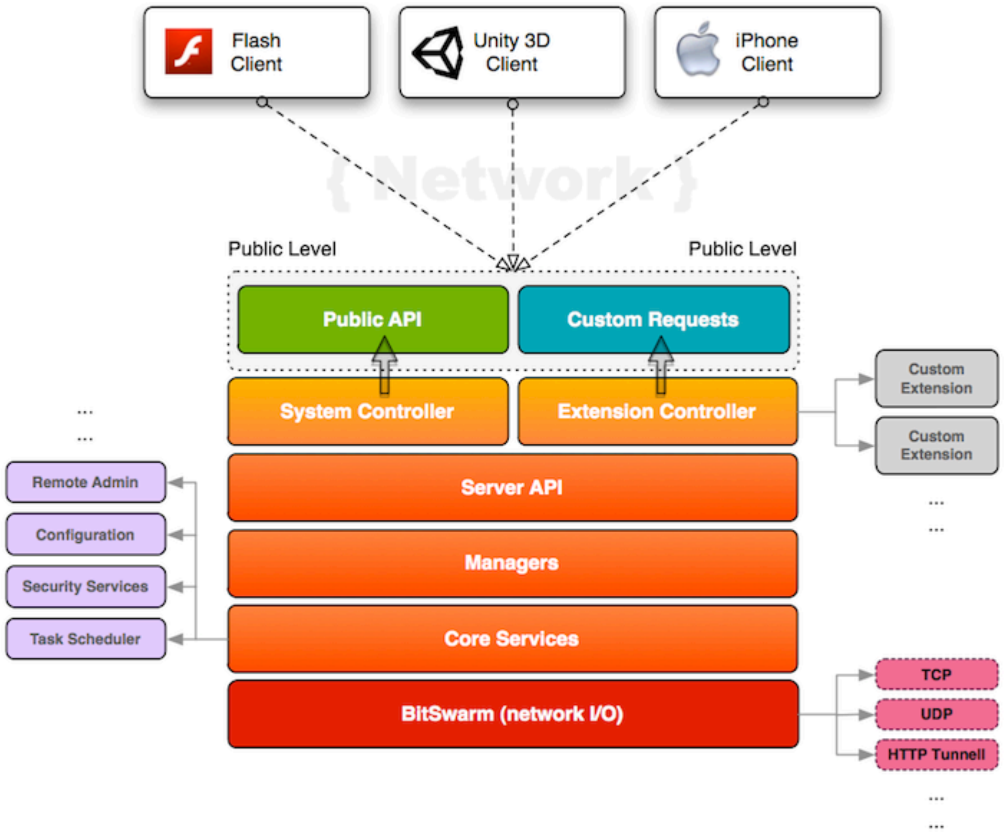
In conclusion the CCU parameter alone is a great tool for advertisement but has little value in describing the ability of the server to perform and scale. As we will proceed through the benchmarks proposed in this paper we will analyze more in detail how to take into account the bandwidth and the other parameters to offer a detailed picture of what you can expect from a single instance of **SmartFoxServer 2X**.

Introducing BitSwarmEngine 3.0

One of the key performance element in **SmartFoxServer 2X** is the core network engine, codename **BitSwarm**, currently at his 3.x version which provides several advantages over most of the other competing game servers.

BitSwarm is specifically designed with massive multiplayer games in mind and provides a remarkable edge over the usual all-purpose socket libraries employed by other server products (typically Apache Mina 1.x).

In essence **BitSwarm** provides SFS2X with TCP/UDP connectivity, session management, network security tools, the HRC (Highly-Resilient-Connections) system, pluggable HTTP tunneling, monitoring and more, using an highly scalable non-blocking design.



The benchmarks

In order to provide better value to our benchmarks we have executed the same tests using two popular open-source socket engines: **Apache Mina** and **JBoss Netty**.
These two products are excellent all-purpose socket libraries that are widely employed as the core for http servers, ftp servers, ssh services and more. Additionally Mina 1.x seems to be the most popular choice in most commercial java-based solutions competing with SmartFoxServer.

We created two alternative servers by plugging the same protocol used by BitSwarm 3.x and using Mina 1.1.7 and Netty 3.2.3.

We then proceeded to run our benchmark tests on all three solutions and compare the results to show the differences that exist between a finely tuned core engine and a generic one.

» Test environment and methodology

The tests were run on a dedicated 1Gbit/s local network using one server machine and several client computers all running Linux (Ubuntu 9.x).

For the main server hardware we used an inexpensive dual core (Intel Core2 Duo) machine running at 2.2Ghz, with 4GB RAM. The reason for this is that scalability can be demonstrated on any adequate hardware. However, if we chose to use a monster server (e.g. 16 or more cores), it would have been almost impossible to fill its total capacity.
In fact we were able to **saturate the 1Gbit/s** line even with this inexpensive setup; if we had to do it with a 16-core server it would have been impossible. Finally in our setup we were able to bring the server machine to very high workloads and observe its behavior under "high pressure".

Each test was run for approximately 2 hours in order to detect possible stability issues, unexpected memory growths, disconnections, etc. The vital informations (CPU, RAM, etc) were all recorded towards the end of the test, allowing the JVM to "warm up" (in other words allowing the JIT compiler to optimize the code at runtime).

For all tests we used the OpenJDK Server VM 1.6.0_0-b12 with no additional settings.

All the CPU/RAM/Network measurements were recorded using the OS system tools such as the *top* command and the visual Resource Manager. The CPU reading goes from 0% to 200% (two cores).

» Benchmark #1 - Discard message

The *discard message* test aims at measuring the socket reading abilities of the Server. The target is bombarded with client messages that are read from the network and decoded from binary into the high level protocol. At this point the obtained object is discarded and rendered eligible for garbage collection. No response is sent back to the client.

The test is repeated 3 times with increasing network pressure: 16Mb/s, 128Mb/s and 512Mb/s. At the end of the cycle (2 hours) all clients are abruptly disconnected at once, providing an extra test in stability and resilience.

Server machine
Intel Core2 Duo, 2.2Ghz
4GB RAM
Linux Ubuntu 9.04
OpenJDK Server VM 1.6.0_0-b12

Round 1
Msg Size: 1024 bytes | Rate: 1 msg/s | Clients: 2000 | Target traffic = 16Mbit/s

	CPU %	RAM %	IN (MB/s)	OUT(MB/s)
BitSwarm	4-5%	2,2%	2,1	0
Netty	6-7%	2,1%	2,1	0
Mina	14-15%	2,5%	2,1	0

Round 2
Msg Size: 8192 bytes | Rate: 1 msg/s | Clients: 2000 | Target traffic = 128Mbit/s

	CPU %	RAM %	IN (MB/s)	OUT(MB/s)
BitSwarm	22-23%	7%	16,3	0
Netty	33-36%	6,9%	16,2	0
Mina	80-82%	7,2%	16,2%	0

Round 3
Msg Size: 16384 bytes | Rate: 1 msg/s | Clients: 4000 | Target traffic = 512Mbit/s

	CPU %	RAM %	IN (MB/s)	OUT(MB/s)
BitSwarm	97%	17%	65	0
Netty	135% for a while then crashes	N/A	~8	0
Mina	Crashed before reaching target traffic	N/A	N/A	0

NOTE
The **IN**/**OUT** values in the above tables are expressed as MBytes (Mbit = MByte * 8).

While memory occupation is similar for all three engines the performance and ability to scale of BitSwarm is evident especially at very high traffic. Thanks to a *zero-copy-buffer* strategy the SFS2X core engine is **extremely light on the CPU** and shows excellent scalability under all three scenarios. Take also in consideration that the last test failed for both Mina and Netty because they couldn't keep up with the massive traffic while BitSwarm is only using 50% of the CPU resources (2 cores = 200%, BitSwarm uses 97%).

» Benchmark #2 - Echo message

The *echo message* test measures both incoming and outgoing traffic by sending back to each client their respective messages. The simulation mimics a scenario where most users are engaged in private chatting or 2 players turn-based games.

Server machine
Intel Core2 Duo, 2.2Ghz
4GB RAM
Linux Ubuntu 9.04
OpenJDK Server VM 1.6.0_0-b12

Round 1
Msg Size: 1024 bytes | Rate: 1 msg/s | Clients: 2000 | Target traffic = 16Mbit/s

	CPU %	RAM %	IN (MB/s)	OUT(MB/s)
BitSwarm	11-12%	4,2%	2,2	2,1

	CPU %	RAM %	IN (MB/s)	OUT(MB/s)
Netty	13-14%	4%	2,2	2,1
Mina	23-24%	4,9%	2,2	2,1

Round 2
Msg Size: 2048 bytes | Rate: 1 msg/s | Clients: 3000 | Target traffic = 50Mbit/s

	CPU %	RAM %	IN (MB/s)	OUT(MB/s)
BitSwarm	25%	5%	6,7	6,5
Netty	30%	5,8%	6,7	6,5
Mina	65%	5,9%	6,7	6,5

Round 3
Msg Size: 2048 bytes | Rate: 1 msg/s | Clients: 6000 | Target traffic = 100Mbit/s

	CPU %	RAM %	IN (MB/s)	OUT(MB/s)
BitSwarm	52%	7,2%	13,5	13,1
Netty	62-64%	7,5%	13,5	13,1
Mina	145-150% (*)	8,9%	13,3	13

(*) Crashed at the end of test when all clients are disconnected at once

This is a fairly simple test that shouldn't pose any particular problem to the server engine because there is only one message created for each message received. Even in this scenario BitSwarm shows significant edge over the other solutions and an excellent ability to scale without heavy performance costs.

Take in consideration that **Round #3** represents and unreal situation, as it is impossible that any human player can actually send 2KBytes of chat text every second.
Typically a chat or turn-based games uses small messages in the range of 50-100 bytes per request.

If we take 80 bytes as an average for chatting/gaming we could very well be able to handle the same traffic (100Mbit) with approximately **120 thousand users**:

instead of	**6000 user** x **2048 bytes = ~100Mbit**
we have	**150000 users** x **80 bytes** = **~100Mbit**

We conservatively trimmed down the mathematically equivalent 150,000 users to **120,000** because each new connection has a certain resource cost in the system so we need to keep that in account too.

» Benchmark #3 - Broadcast echo message

The *broadcast echo* test is the most aggressive of the set as it subjects the socket engine to an extreme pressure. Each message received generates a new message for each connected client. In other words if we have 1000 users sending 1 message per second we end up with 1000*1000 = 1 million messages per second.

In this test we aimed at saturating the 1Gbit/s ethernet network.

Server machine
Intel Core 2 Duo, 2.2Ghz
4GB RAM
Linux Ubuntu 9.04
OpenJDK Server VM 1.6.0_0-b12

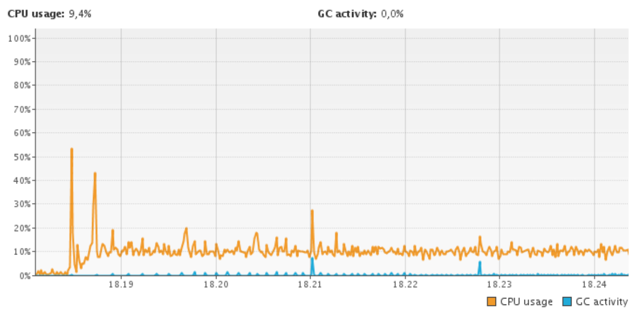
Bandwidth	BitSwarm		Mina		Netty	
	CPU	RAM	CPU	RAM	CPU	RAM
80Mbit	14%	1,40%	55%	N/A	50%	4,60%
320Mbit/s	45%	1,30%	139%	3,50%	crashed	N/A
840Mbit/s	91%	2,20%	crashed	N/A	N/A	N/A

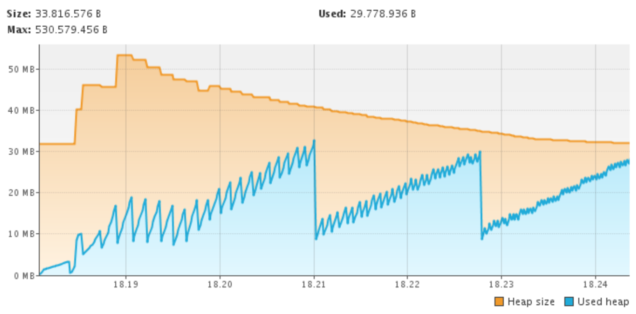
With 840Mbit/s of actual data (not counting the TCP/IP packet data) we were able to fully saturate the 1xGigabit line. Only BitSwarm was able to sustain this throughput while the other engines produced several crashes or extreme slow downs.
We repeated the test several times and what you see in the above table represents an average of the multiple passes.

We also ran a version of the test monitoring the activity of the JVM Garbage Collector for each socket engine. You can notice how the BitSwarm *zero-copy-buffer* strategy makes a remarkable difference in avoiding continuos creation and destruction of objects which ultimately leads to massive GC activity and poor performance.

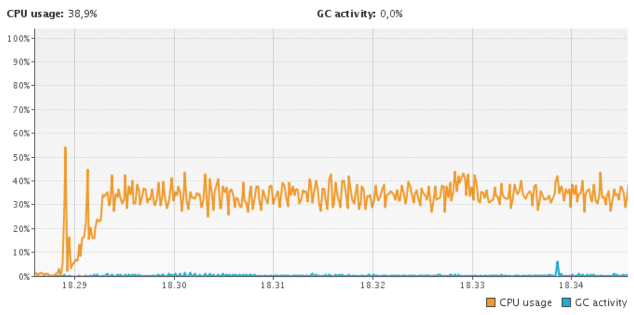
The GC and heap memory activities were recorded by plugging VisualVM in the JVM at runtime, before starting the test.

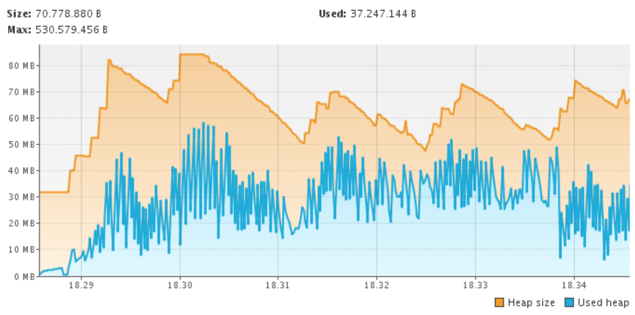
BitSwarm



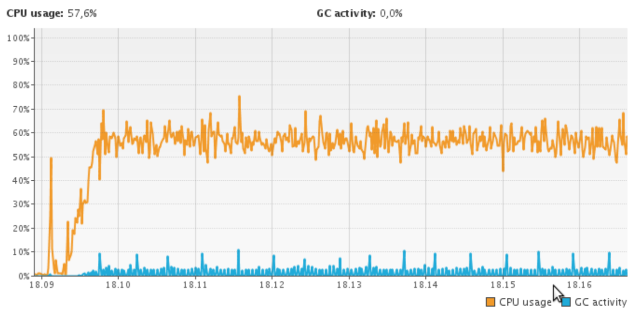


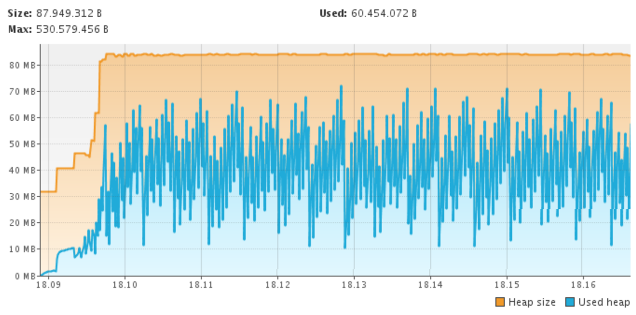
Netty





Mina





You can notice a dramatic difference in the JVM heap allocation patterns (right side) where Mina shows the most inefficient memory usage and Netty displays a slightly better situation but it is still engaging the GC too often (left side), with consequent performance loss (the **blue area** represents the objects allocated in memory, the **orange area** shows the size of the heap).

» Conclusions

In conclusion the ability of SmartFoxServer to deliver top-notch performance even on commodity hardware provides more value for your money. You won't need to spend thousands and thousands of euros/dollars in hardware to get started and if your application will grow quickly you won't be forced to immediately move to new hardware.

In general we like to recommend a dedicated Quad-Core server with 4GB of RAM running Linux (64bit) as a very good system to get started with any medium-sized SmartFoxServer project. The server editions of Windows also work great with SmartFoxServer although we have noticed that Linux delivers the best overall performance.

Outside the core

The philosophy of careful optimization employed for the core engine is also applied in many other components of the **SmartFoxServer 2X** platform.

» Protocol

The binary protocol was designed from scratch with the same approach: we tested a few well known solutions and noticed inadequate performance so we concentrated on a custom made solution optimized for our needs and goals.
The SmartFoxServer 2X protocol is **6 times faster** than its predecessor in terms of serialization/deserialization and highly efficient in terms of bandwidth occupation.

You can learn more about the protocol performance on our website.

» Server Extensions

SmartFoxServer 2X allows developers to plug in their code and implement all kinds of sophisticated interactions just like a http-server allows to create any type of web application.

The architecture behind the Extension system is extremely light, fast and simple to understand. In order not to compromise performance we dropped the use of scripting languages (on the server side) which unfortunately are not able to scale under high workload. By leveraging the Java language, developers will be able to fully take advantage of the scalability and performance of the JVM (version 6 and higher) including concurrent collections, atomic types, locks, thread coordination tools, etc...

» Scalability via visual configuration

SmartFoxServer 2X is entirely built with a non-blocking architecture allowing for high concurrency and scalability. This approach allows the different stages of the server to be fine-tuned for different requirements via the server's own visual AdminTool.

Whether you are running on dual core machine or on a multi-core/multi-CPU monster you will always be able to allocate the necessary resources so that the Server can use the machine's full power.

» Client API

The same approach was also applied to the client API where performance issues can be even more critical. In fact most action games require considerable amount of resources in order to keep steady animations at 25-30 frames/sec. while the network data continue to stream and the game logic is running.

Not all game server solutions out there seem to consider this aspect and they deliver API components that can eat the computer's resources too quickly.

Here's an anonymous test done with SmartFoxServer 2X Flash API and another commercial game server's Flash API.

Server machine
MBPro dual-core i7@2.6Ghz, 8GB Ram
Java Runtime: 1.6.0_20
Java HotSpot(TM) 64-Bit Server VM (build 16.3-b01-279, mixed mode)

Client machine
iMac Core2 Duo @3.0Ghz 4GB Ram
Flash Player standalone version 10.0.2.54

Single Message test
The test shows a typical real-time game scenario where lots of messages arrive each second to the client. One message is sent to the client every **<interval>** amount of milliseconds as specified in the table.

Interval	Msg count	Bandwidth	Server CPU	Client CPU
10 ms	100 msg/s	33KB / 38KB	3% - 4%	6% - 22%
5 ms	200 msg/s	66KB / 78KB	4.7% - 7%	8% - 40%
2 ms	500 msg/s	167KB / 190KB	8% - 14%	12% - 90%
1 ms	1000 msg/s	332KB / FAIL	13% - 58%	13% - FAIL

In red are the values obtained with **SmartFoxServer 2X** and related API.
In blue are the values obtained with the other game-server and related API.

At 200msg/sec you can already see how the alternative product is eating 40% of the client resources just to deal with the network traffic, while the SFS API have a very light impact (less than 10%).

Final thoughts

The previous pages clearly showed how well SmartFoxServer 2X scales as the number of concurrent users grows. In closing we will also address two more of the common questions we have mentioned in the opening of this article.

Q: How many concurrent clients SmartFoxServer can handle?

We have seen there are many variables that can dramatically affect the amount of CCU, however we can provide an approximation by using different categories of games.
With a dedicated machine running 1-2 quad-core CPU(s) and 4-8GB RAM:

- **chatting / turn based games** typically don't hit the server performance significantly so you can expect thousands of hundreds of clients running on a single SmartFoxServer 2X instance;
- **fast real-time action games** typically hit the server with high message rates so you can expect the number of CCU to be 1/5th to 1/10th of the previous scenario, supposing that we're using the same hardware.

Q: Will my hardware be capable of handling <AnyNumber> of concurrent users?

This is the most difficult question to answer without the details of the application. Also one very important variable is how scalable is the server-side code plugged into the server that rules the game logic.

Supposing that the Extension code is bottleneck-free we recommend a dedicated Xeon (or equivalent) quad/eight-core machine for any medium complex application/game. As regards RAM, usually 4-8GB are enough. This will guarantee enough resources and scalability to support your growth for quite a long time, so that you don't have to switch too soon in case there's an unexpected growth. Also this type of hardware is currently (as of 2012) very affordable.

Should the game popularity skyrocket and become the next *Club Penguin*, you will probably need more server instances and more powerful machines, but in that case the business will be so successful that such investment should be of no concern.

Appendix

CCU
The total number of concurrent users connected.

Non-Blocking architecture
It refers to a programming technique that never blocks the current thread of execution. This approach allows for high concurrency and performance because it doesn't force to use an excessive number of threads.
For example the Java Development Kit provides a non-blocking API for network I/O. A non-blocking architecture is a software design that applies this concept throughout all its components.

Scalability
The ability of the software to take work efficiently under different workloads. In the context of our test it is also the ability to run effectively taking advantage of the hardware in use (single-core vs multi-core cpu, small amount of RAM vs large amounts of RAM, etc).
Sometimes this term is also referred to as "vertical scalability" as opposed to "horizontal scalability" which is used in distributed environments.

Server Extension
In the context of the SmartFoxServer platform, it represents the developer's custom code plugged in the server that handles specific game/application logic requests.

Socket Engine
It is the low-level code of a Game Server that handles raw I/O from the network. A finely tuned socket engine can make a big difference int the overall Server performance and ability to scale.

Zero-Copy Buffer strategy
A programming technique targeted at reusing memory buffers as much as possible in low-level network code. This technique avoids continuos generation of byte arrays and byte buffers which in tun can severely hit the server performance.