Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 1342 – 1351

Procedia
Computer Science

2013 International Conference on Computational Science

Self-tuning multimedia streaming system on cloud infrastructure

Gheorghe Sebestyen[a]*, Anca Hangan[a], Katalin Sebestyen[a], Roland Vachter[a]

[a]Technical University of Cluj-Napoca, G. Baritiu 26-28, Cluj-Napoca, Romania

Abstract

In this paper we present an automatic performance tuning solution for the optimization of a high-demand multimedia streaming service built on a dedicated private cloud architecture. The optimization focuses mainly on two aspects: energy efficiency and quality assurance of streaming services. As QoS factors we take into consideration the minimum initial waiting time and the cumulative disruption time of continuous playing at the client side. To achieve the best performance, a cloud-based architecture is proposed with on-demand dynamic allocation of resources and a multi-level data flow with caching capabilities.

Keywords: automatic performance tuning; multimedia streaming; server optimization; cloud infrastructure, benchmarking; caching**;**

1. Introduction

In the last years, multimedia data transmission became dominant on the Internet and other communication networks (e.g. 3G and 4G cell networks). This tendency is technologically promoted through high bandwidth networks and high performance mobile devices with great graphical capabilities at an affordable price.

An exponentially growing number of Internet users are requesting multimedia streaming services such as movies and music on demand, Internet radio and TV or teleconferencing. Statistics show that the total number of Internet users reached one third of the world's total population and a single multimedia service provider like YouTube receives more than four billion video requests a day. Three billion hours of videos are viewed monthly, while the traffic from mobile devices has tripled in 2011.

* Corresponding author. Tel.: +40 264 401489; fax: +40 264594491.
E-mail address: gheorghe.sebestyen@cs.utcluj.ro.

In such conditions, where a huge number of users with high bandwidth capabilities are requesting multimedia content, the task of building high quality and efficient multimedia services is not a trivial job. Parallel and distributed solutions are required to fulfill all the incoming requests at an acceptable quality level. The optimization of distributed server systems is a very provocative topic, especially for multimedia streaming where the time constraints are critical and the number of requests is highly fluctuating.

Therefore, in this paper we propose a self-tuning system that takes into consideration quality and efficiency requirements. Through benchmarking, our resource manager determines the actual transmission capabilities of available computing elements and allocates tasks in order to fulfill some transmission time constrains (e.g. initial delay and cumulative disruption time).

The structure of the paper is organized as follows: the second section presents related research work; section three contains the overall system architecture; next section describes the evaluation process through benchmarking; section five presents the server management and optimization algorithms and finally, section six contains the conclusions and future work.

2. Related work

In order to facilitate efficient delivery of multimedia content over the network, several articles ([1], [2] and [3]) propose constructing a network of *proxy servers*. These servers represent an extra layer between clients and *storage servers* that preserve the multimedia content. The role of the proxy servers is to take over the load from the storage servers by caching certain video items and serving client requests. This approach has the potential of improving the overall performance of the system and of balancing the load between the available resources. The authors of [1] propose the use of proxy servers to accelerate data-centric operations in a distributed architecture. Their study shows that adding extra layers between the content suppliers and the clients can improve the overall performance of the system. Articles [2] and [3] propose the idea of dynamically varying the number of used servers based on the current system demands. Under heavy load, instead of rejecting new client requests or reducing quality of service, a new proxy server is allocated. In the opposite case, when fewer servers can handle the current client requests, the least useful server is de-allocated.

There are also proposals targeting cache replacement strategies. In [5] videos are replaced by taking into account their utility value. This value is calculated by a function that takes as parameters the size, time of last access, hit count and quality (with respect to the user preferences) of the video. The weights of the parameters are determined through an evolutionary algorithm.

Numerous articles recommend different video segmentation policies. In certain scenarios, caching only parts of a multimedia file is preferred over caching the entire content. The authors of [4] investigate the impact of five segmentation policies on the transmission delay: uniform video splitting, first-reduced splitting, linear splitting, Fibonacci-based splitting and truncated Fibonacci-based splitting. Article [6] compares the byte-hit ratio obtained for no-segmentation, mirror segmentation and exponential segmentation. In [7] authors divide videos into segments containing a number of blocks equal to powers of two. This approach is compared with the case of no-segmentation and prefix-caching presented in [8].

3. System architecture of the optimized video streaming system

The distributed system for media streaming proposed in this paper has six logical internal parts and one logical external one. The external part is represented by the Client applications that are requesting different media files. The internal parts are: Dispatcher, Resource Manager, Benchmark Client, Proxy Servers and Storage Servers. The Dispatcher listens to client requests and allocates them to proxy servers in accordance with a given policy that takes into account QoS restrictions and energy efficiency requirements. The

Benchmark Client measures the proxy servers' performance characteristics in order to optimize the use of the servers.

The Resource Manager identifies available proxy servers and allocates them to the Dispatcher. This component keeps track of the existing servers and their performance capabilities based on the results obtained from the Benchmark Client. The Storage Server preserves all the media files and when needed, supplies these files to the proxy servers. Proxy Servers provide video streaming services to clients; for this purpose they temporary store the media files obtained from the Storage Server in their cache.

4. Proxy server evaluation through benchmarking

In a server management and optimization system, the key challenge is to find a proper server configuration (possibly in an automatic way) that maximizes its performance. When using a server for media streaming, the limit of the server is reached when at least one client does not receive the amount of data needed for continuous playing of data.

The limit of the server is influenced by two types of factors: internal and external. Internal factors are the server's hardware components, such as the speed of the CPU, amount and speed of the internal memory, speed of the hard disk and the capabilities of the network adapter. The external factors are the characteristics of the network connection. Different hardware configurations have different optimal working parameters.

The simplest solution for detecting when a server reached its limit would be to monitor each of the server's clients and verify for a given period of time that the client received from the server enough amounts of data to play the media for that given time. This approach works when all of the clients have a higher network speed then the bit-rate of the video they are requesting. But when a client has a very small bandwidth, the server would interpret the failure to fulfill the client request as its own fault; in this case it will consider that its limit is reached. Even one client with a small bandwidth would give the wrong sense of overload.

To avoid such situations, there is a need of knowing exactly the limit of the server before accepting any real clients. To fulfill this requirement, during the server startup phase a benchmarking process is initiated. The Benchmark Client component generates virtual client requests and measures the server's overall capabilities. For a small number of clients the bottleneck in most of the cases is the network connection, but for a large number of clients the bottleneck could be the server's configuration. Therefore, the best configuration should be found and used before measuring the overall capabilities of the server. In order to find the best configuration of a server, different streaming strategies are tried out through "localhost" (with no network communication involved).

The experimental results (see section 6.1) have shown that in order to achieve the best performance a limited optimal number of threads must be used. In this case a larger number of clients should be served through the same thread in a concurrent manner; clients receive small data packages in sequential order.

This approach raises two questions: what is the optimal number of threads and what is the optimal size of the data packages. These parameters could differ from one hardware configuration to another. Therefore, these numbers should be obtained by trying multiple combinations of possible values in the benchmarking process. This process calibrates the server to an optimal configuration.

The next step is to measure the external factors that influence the streaming. In order to have an overview of the network connection using the server's optimal configuration, an external Benchmark Client system is used. This generates a large number of clients outside of the server and measures the overall throughput as the amount of data received over a given period of time.

Using this double process of benchmarking gives a confident evaluation of the server's capabilities and therefore the client diversity does not have an impact on the server management and streaming process.

5. Management of Proxy Servers

5.1. Server allocation

In order to build a scalable platform that can serve fluctuating request streams, an extra layer needs to be introduced between the end-users and the Storage Server. This layer is represented by the collection of proxy servers (referred to as simply *servers* from now on). A server is responsible for handling the client requests that it receives and caching recently used video files. Through this extra layer the system becomes more scalable and a significant amount of load is taken off the Storage Server. In order for this model to make sense, the sum of server bandwidths must be greater than the bandwidth of the Storage Server.

The Dispatcher is responsible for the allocation of the servers. Client requests arrive to this component which redirects them to the fittest server. In case of each request, the Dispatcher decides whether the current set of allocated servers can host the newly arrived request. In order to be able to make this decision, the Dispatcher tracks the current state of each server, more precisely the available bandwidth and free storage space. Moreover, the Dispatcher queries the Storage Server about the size of the video that is being requested. This is important since a server hosting the request must have a free or releasable cache size, no less than the video size. A part of the cache is considered releasable if the video that it stores is no longer requested by any other clients. If none of the servers can handle the new request, the Dispatcher asks for a new server from the Resource Manager and redirects the request to the newly obtained server. No client request is rejected as long as there are available servers returned by the Resource Manager. Otherwise, if there are servers that still have enough storage and bandwidth resources, then the fittest is selected to serve the request.

When the Dispatcher selects a server to host the request, a fitness value is calculated for each potential server. The one having the highest value will be chosen. Two fitness policies have been defined for the server allocation: load balancer and cache-aware policies. The purpose of the load balancer policy is to distribute requests such that all allocated servers have similar unused bandwidth values. In case of each new request that arrives to the Dispatcher, the server with the maximum unused bandwidth is chosen.

On the other hand, the cache-aware policy takes into account the cache contents of the servers. First the available set is filtered by the bandwidth constraint (the sum of the video's bit-rate and a server's currently used bandwidth must not exceed the available bandwidth of the server). After this step, if there are some servers that already have the video in their caches, then the one with the highest unused bandwidth will be chosen. If none of the filtered servers have the video in their caches, then similarly the one with the highest unused bandwidth will be chosen, and in this case the video needs to be fetched from the Storage Server. In order to apply this policy, the Dispatcher must keep track of the cache contents of each server. Due to this, servers notify the Dispatcher about their cache modifications.

5.2. Caching

As described in the previous subsection, each server stores the videos in its cache (which is a predefined area on its hard disk). No segmentation is used in this stage of the project; videos are fully cached.

When the Dispatcher receives a new server, it is informed of its cache size. During the system operation, the Dispatcher also tracks the cache content modifications such that a new request is redirected to the fittest server. If a new video must be streamed into a server's cache from the Storage Server, a cache section equal to the video's size is reserved for the given video.

As cache replacement policy the LFU (Least Frequently Used) policy is applied. The videos that cannot be discarded at a certain moment are those which are being streamed from the Storage Server or which are being streamed to at least one client.

5.3. Streaming

When a server receives a new video request one of three cases can hold:
- case 1: the video is not yet in the cache
- case 2: the video is already fully stored in the cache
- case 3: the video is only partially contained in the cache (the streaming of the video from the Storage Server is still in progress).

In the second case the streaming to the client can start immediately. However, in case 1 the video needs to be requested from the Storage Server and needs to be partially fetched before the streaming can begin. Case 3 can only occur when another client has previously requested the video. In cases 1 and 3 the server needs to decide when is the best time to start streaming the video to its requesters. If the streaming is postponed until the entire video is in the cache, then the end-user will have to wait a relatively long time until they can start viewing the video. Another option is to start streaming to the client as soon as the first packages are received from the Storage Server. However, in this case the client might perceive occasional interruptions during the viewing time. Therefore an optimal starting time should be found which offers an acceptable QoS. The QoS value depends on two measurable metrics: the initial delay and the total waiting (disruption) time. In order to obtain a satisfactory QoS, both values should be minimized.

According to [4], three QoS classes can be differentiated: QoS guaranteed services, Assured services and Expedited services. To satisfy requests that belong to the first class, resources are reserved in advance to offer the lowest initial delay and no waiting times. The other two classes do not make reservations in advance. Instead Assured services policy starts the streaming only when service continuity can be guaranteed (which often leads to increased initial delays). On the other hand, Expedited services policy minimizes the initial delay, but allows service discontinuity. From the three QoS classes we selected the Assured services, because the goal is the minimization of the total waiting time. However, since in case of a multimedia streaming system the initial delay also affects client satisfaction, this value should also be considered. As a consequence, a balance should be kept between the two metrics, but the waiting time should be favored. The reason is that clients are usually more willing to accept some initial delay if afterwards the viewing exhibits only small interruptions.

As mentioned before, there are two extreme moments when the video streaming can be started:
- only after the entire video is in the cache – this approach minimizes the total waiting time but severely increases the initial delay (especially in case of large multimedia files)
- as soon as the first data package is received from the Storage Server - this approach has a positive impact on the initial delay, but instead might result in annoying interruptions.

Instead of choosing one of the above two solutions, we propose some mixed policies that try to optimize the QoS parameter, defined as follows:

$$QoS = \frac{1}{0.25 * initialDelay + 0.75 * waitingDelay} \tag{1}$$

These policies differ in the way in which the best moment for starting the streaming to the client is chosen. The first, called *FixedPercentPolic*y decides to start the streaming after a fixed percentage of the data is cached in the proxy server. We performed experiments with 10% to 50% of data loaded in the cache. The next three policies take into consideration the streaming speed (more precisely the bandwidth) between the storage server and the proxy server. The bandwidth is used to predict the best moment to start the transfer to the client in order to increase the QoS. So, the second policy called *CurrentSpeedStreamPolicy* considers the currently measured bandwidth between the two servers. The next policy called *CurrentSpeedStreamPolicyWithBuffer* is using an extra buffer for smoothing the transfer. The last policy, *TrendBasedPolicy*, takes into consideration not only the current bandwidth (*current_Bw*), but also the minimum (*min_Bw*), maximum (*max_Bw*) and the trend of the bandwidth. According to this policy, if the requested video is not yet fully cached, then its streaming does not start until at least 5% of its content is cached. During this period, the streaming speed of the Storage Server is measured and predicted at each package receipts according to the formula presented below:

$$Bw = \frac{\max\_Bw + \min\_Bw + 4 * current\_Bw}{6} * \left(1 + \frac{trend}{2 * \max\_trend}\right) \tag{2}$$

The *trend* variable tries to reflect the trend of the observed bandwidth change. A positive value indicates that the bandwidth had mostly increased during the observed interval. Initially the *trend* value is zero, and it is modified at package receipts as follows: it is increased by one if the current bandwidth is greater than the last measured bandwidth and decreased otherwise. The *max_trend* is the maximum value that the *trend* could achieve. The bandwidth value (*Bw*) will hold the estimated Storage Server bandwidth for the near future. Based on this predicted value, we evaluate whether starting the streaming to the client at that moment would result in service interruptions. If so, the starting time is postponed. Otherwise the streaming can be started.

Serving the clients is done using a predefined number of threads. This number is determined for each individual server by the Benchmark Client. Experiments showed that this approach yields better results than creating a separate thread for each client. Clients are allocated to the set of threads such that the load on each thread is similar. A thread serves clients in a round-robin fashion: each client receives a number of data packages for a certain period of time, and then the next client will receive a portion of data. This process is repeated until there are no more clients to serve. Video data is sent through packages of equal size (except for the last package, which might be less). The optimal package size for each individual server is determined during the benchmarking phase.

5.4. Releasing Proxy Servers

In case of a multimedia system the number of client requests fluctuates over time. There are times when this number reaches a peak and more servers are needed than usual; but similarly there are periods when fewer servers could handle the current requests. In the second case, from an energy-efficiency point of view, it would be a good idea to find the servers which are least needed and return them to the Resource Manager. The Resource Manager could then reassign them to some application that needs more resources or put them into a lower power-consumption state.

The release of a server is performed in two steps. First, when the Dispatcher decides that the server is no longer needed, it puts it into an idle state. Servers in this state continue to serve their assigned clients, but they don't get new assignments from the Dispatcher. When they finished serving all their clients, these servers are returned to the Resource Manager. However, it might happen that after a server is put into the idle state there is an increase in the request volume and the Dispatcher realizes that more resources are needed. In this case, instead of requesting a new server from the Resource Manager, an idle server is reactivated.

We proposed and experimented four different policies for selecting the server to release. The first policy (*MinNrClientsDisposalPolicy*) chooses the one with the lowest number of clients. The second policy (*MinBandwidthDisposalPolicy*) selects the one with the lowest total bandwidth potential. The third policy (*MinBandwidthDisposalPolicy*) would return the one with the smallest cache size. Finally, the fourth policy (*AdaptiveDisposalPolicy*) is based on a utility function that takes into account both the cache size and the bandwidth of the servers. The utility is calculated as the weighted sum of these two properties of a server. Initially both weights are equal to 0.5 (both properties have the same impact on the server selection). During operation the weights are dynamically changed in accordance with the last conditions (cache or bandwidth) that required allocation of a new server. The weights are recalculated based on the algorithm shown in Appendix A.

If the last two occasions when the Dispatcher required a new server were due to the same reason (both were because of storage limitation or both were because of bandwidth constraints), then we keep those servers which could fill this particular constraint better. Therefore the corresponding weight should be increased to reflect that the utility of such servers is higher. On the other hand, if the last two occasions when the Dispatcher required a new server were due to different reasons (one was because of the storage constraint and the other because of bandwidth limitation), then this suggests that previously the weight was increased by too much. Compared to the other three policies, this one adapts better to the changes that are happening in the system.

6. Experimental results

6.1. Evaluations through benchmarking

During benchmarking and server configuration calibration, more approaches were tried out. The first approach is using one thread per client for streaming. Each client request will be served on a different thread. The results can be seen in Figure 1a.

The figure shows that for a large number of clients the server's performance (server's total bandwidth) is highly decreased. Threads are beneficial when their number is small, becaus they allow parallel execution; but for a larger number there are too many context switches and the overall performance is lower than without threads. In figure 1a it can be seen that the bandwidth is increasing linearly until we reach 4 clients (threads) and it increases much slower until approximately 10 clients. The intuitive explanation for this phenomenon is that we used a quad-core computer, which can handle 4 threads in parallel.

The failure of this approach has inspired the use of a limited number of threads. More than one client is mapped onto a thread. On a thread, the clients are served in a round-robin manner, each client being served with a number of data packages proportional with the video's bit-rate that is being requested. The results are shown in Figure 1b. The overall performance remains stable even for a larger number of clients. This behavior is due to the controlled and reduced context switches.

The optimal configuration of a server is set experimentally by trying possible combinations of thread number / data package sizes. The result of running the benchmarking process on a HP Elitebook 8560w notebook with Intel Quad Core i7 CPU, 8GB of RAM, 5200 RPM hard disk with eSATA2 connection, is shown in figure 2. Finally, for a given particular server the combination that offeres the highest bandwidth is chosen.

The average of the results grouped by data package and thread number variation can be seen in figure 3.

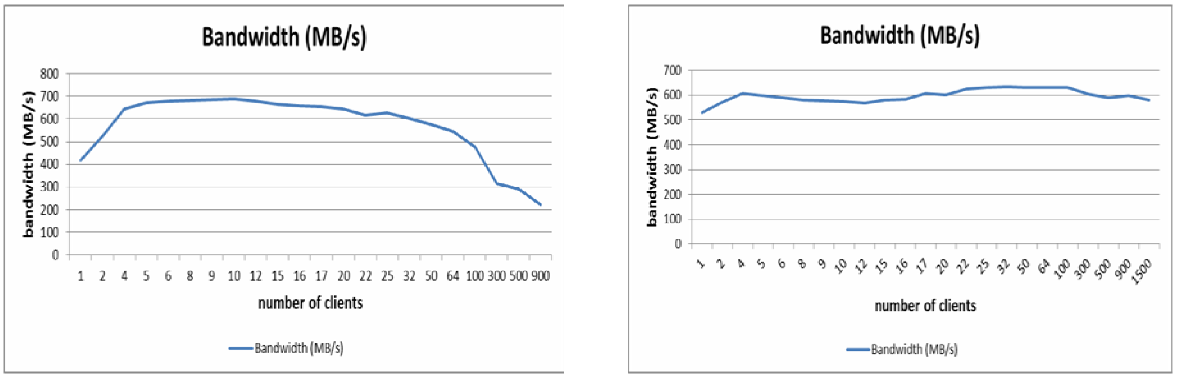

Fig. 1. (a) client per thread approach; (b) limited thread approach

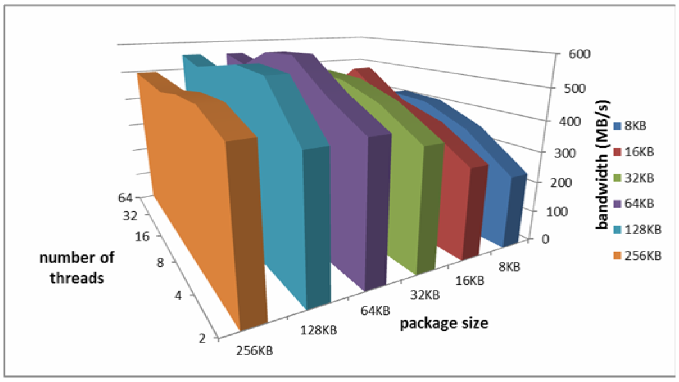

Fig. 2. Results of searching for the optimal configuration

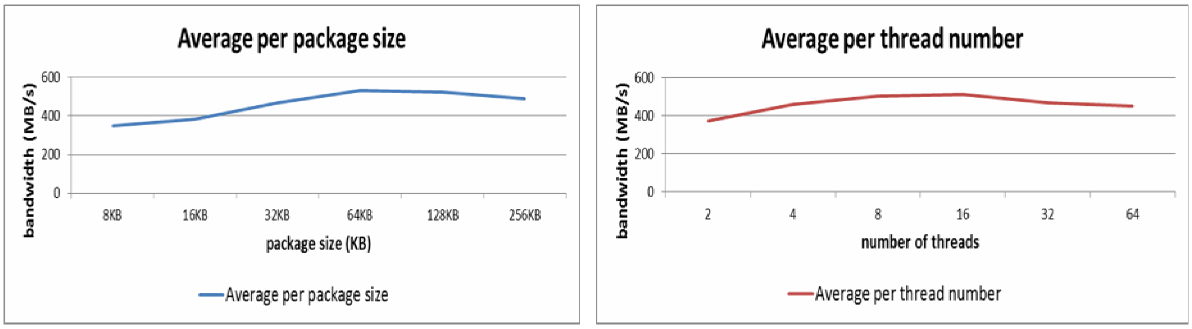

Fig. 3. (a) average bandwidth grouped by package size; (b) average bandwidth grouped by thread number

6.2. Proxy Server Management

In order to test our proxy server management policies we have performed several experiments. The stream of client requests has been generated using the Webtraff [9] synthetic web traffic generator tool. The system is tested with the generated input data, which is represented by 500 client requests for 150 multimedia files. The file sizes range from 5 MB to 150 MB and have a duration ranging between 1 and 20 minutes. The "one-timers" (videos requested only once) represent 30% of the requests. The generated data has a lightly skewed (Zipf slope of 0.3) object popularity distribution.

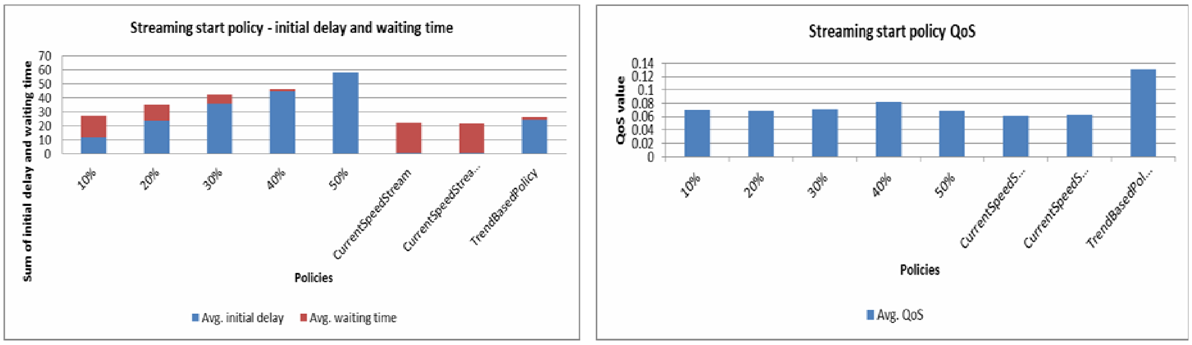

Fig. 4. (a) Streaming start policy – sum of initial delay and waiting time; (b) Streaming start policy - QoS

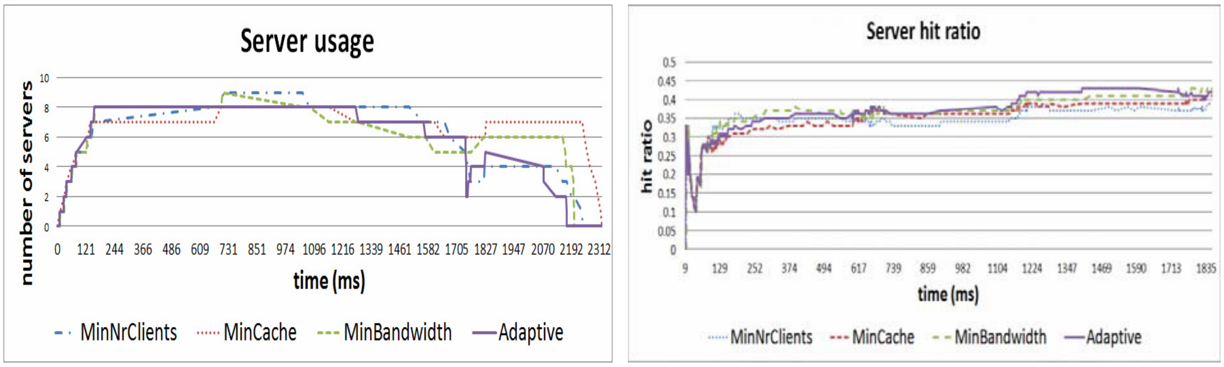

Fig. 5. (a) Server usage; (b) Server hit ratio

We experimented four policies that establish the starting moment of a new video stream: FixProcentPolicy, CurrentSpeedStreamPolicy, CurrentSpeedStreamPolicyWithBuffer and TrendBasedPolicy (described in section 5.3). These policies have direct impact on the QoS. Figure 4 illustrates the obtained results.

The FixProcentPolicy is tried out with different configurations: 10%, 20%, 30%, 40% and 50%. Figure 4a shows the sum of the initial delay and waiting time, while the figure 4b shows the QoS value of the measured parameters. Taking into consideration that the waiting time is much more important than the initial delay, the TrendBasedPolicy is proven to be the best choice (because of its adaptability).

The next experiments were meant to establish the best policy for the allocation and release of proxy servers. Figure 5 shows the results for the four tested policies: minimum number of clients, minimum cache, minimum bandwidth and the adaptive one (described in section 5.4).

As it can be seen, at the beginning the allocation is almost identical: all four policies request the same number of servers. However, later on, the number of servers differs. This tendency is influenced by the fluctuation of the client request stream and the release strategy. As it can be seen, the first one to release all resources is the AdaptiveDisposalPolicy. Measurements also showed that this policy uses resources for the shortest time. In order to study the efficiency of the four disposal policies, the cache hit ratio of the servers has been observed Figure 5b shows that the policies obtain a similar cache hit ratio during the operation of the system. The AdaptiveDisposalPolicy and the MinBandwidthDisposalPolicy obtain in turn the best values for this metric.

7. Conclusion

The paper proposes a self-tuning cloud-based infrastructure destined for multimedia streaming. The optimization strategy, used for self-tuning, takes into consideration streaming quality parameters, bandwidth maximization and power consumption minimization.

The available resources are tested through benchmarking in order to optimize the allocation of incoming streaming requests. Computing resources are allocated and released in accordance with the request fluctuations. The system can determine the optimal number of threads for a given particular computer and uses this information to maximize the bandwidth of the streaming server.

We proposed a data caching system that can increase the throughput of data which is requested more frequently. We also proposed a combined resource allocation and release strategy that uses a smaller server time. The policy that proved to be the most energy-efficient and which resulted in a good cache hit ratio is based on a learning algorithm that adapts to the client request fluctuations.

Experiments showed the feasibility and adaptability of the proposed solution. Some experimentally determined performance parameter values guided us in finding a good task allocation strategy. For instance, in the case of optimal thread number, the common approach is to generate a new thread for every new client request. But, experiments showed that a limited number of threads (which depends on the overall characteristics of a particular computer) perform better. In this case as performance parameter, we considered the server's total bandwidth. Grouping a number of clients on the same thread proved to be a better choice.

References

[1] Jon B. Weissman and Siddharth Ramakrishnan, "Using Proxies to Accelerate Cloud Applications", Workshop on Hot Topics in Cloud Computing (HotCloud'09), San Diego, CA, June 2009.

[2] C. Cobarzan, "Dynamic proxy-cache multiplication inside LANs", In Euro-Par 2005, volume 3648 of Lecture Notes in Computer Science, pages 890-900, Springer, 2005.

[3] C. Cobarzan and L. Boszormenyi, "Further development of a dynamic distributed video proxy-cache system", In Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, pages 349-357, IEEE Computer Society, 2007.

[4] Dario Bruneo, Giuseppe Iellamo, Giuseppe Minutoli, Antonio Puliafito, "GridVideo: A Practical Example of Nonscientific Application on the Grid", IEEE Trans. Knowl. Data Eng. 21(5): 666-680 (2009)

[5] Cristina Mihaila, Claudiu Cobarzan, "Evolutionary approach for multimedia caching", In 19th International Workshop on Database and Expert Systems Applications (DEXA 2008), 1-5 September, 2008, Turin, Italy, IEEE Computer Society, pp. 531-536, 2008, ISBN 978-0-7695-3299-8

[6] C. Cobarzan, "Mirror segmentation: A New Segmentation Strategy for Video Objects inside Video Caches", In 7th RoEduNet International Conference 2008. Networking for Research and Education, 28-30 August, 2008, Cluj-Napoca, Romania, Editura U.T. Press, pp. 101-105, 2008

[7] Kun-Lung Wu, Philip S. Yu, and Joel L. Wolf, "Segment-Based Proxy Caching of Multimedia Streams", Proceedings of The Tenth International World Wide Web Conference, May 1-5, 2001, Hong Kong

[8] S. Gruber, J. Rexford, and A. Basso, "Protocol considerations for a prefix-caching proxy for multimedia streams", Computer Network, 33(1-6):657-668, June 2000.

[9] N. Markatchev and C. Williamson, "Webtraff: a GUI for Web Proxy Cache Workload Modeling and Analysis", In MASCOT'02:Processdings of the 10th IEEE International Symposium on Meling, Analysis and Simulation ofComputer and Telecommunications Systems, IEEE Computer Society, 2002

Appendix A. Pseudo-code for calculating weights

```
LAST_REASON = the reason why a new server was needed
the last occasion (cache or bandwidth constraint)
LAST_BUT_ONE_REASON = the reason why a new server
was needed at the last but one occasion (cache or bandwidth
constraint)
if (LAST_REASON == CACHE) then {
  if (LAST_BUT_ONE_REASON == CACHE) then {
     previousCacheCoeff = cacheCoeff;
     cacheCoeff *= 3/2;
  }  else {
     temp = cacheCoeff;
     cacheCoeff = (previousCacheCoeff + cacheCoeff) / 2.0;
     previousCacheCoeff = temp;
  }
  if (cacheCoeff > 1.0)   {  cacheCoeff = 1.0; }
bandwidthCoeff = 1.0 - cacheCoeff;

} else {
  if (LAST_BUT_ONE_REASON == BANDWIDTH) {
     previosBandwidthCoeff = bandwidthCoeff;
     bandwidthCoeff *= 3.0 / 2;
  }  else  {
     temp = bandwidthCoeff;
     bandwidthCoeff = (previosBandwidthCoeff +
bandwidthCoeff) / 2.0;
     previosBandwidthCoeff = temp;
  }
  if (bandwidthCoeff > 1.0) {
     bandwidthCoeff = 1.0;
  }
  cacheCoeff = 1.0 - bandwidthCoeff;
}
```