A Centralised data aggregation layer across different consuming platforms

Aliaksandr Ivanou

A thesis presented for the degree of
Master of Science

UPPSALA
UNIVERSITET

Department of Information Technology
Uppsala University
Sweden
23.05.2015

Acknowledgements

I want to thank my girlfriend Melissa Melgar for helping and correcting my grammar mistakes. I think she did a great job and I am very appreciative of it. I also want to thank Swedish Institute for providing me with scholarship so I can study in Sweden.

Contents

1 Introduction

1.1 Background

The thesis project is done in the company *Accedo Broadband AB. Accedo is the market leading enabler of TV application solutions. Accedo provides applications, tools and services to media companies, consumer electronics and TV operators globally, to help them deliver the next-generation TV experience. Accedo's cloud-based platform solutions enable customers to cost-efficiently roll out and manage application offerings and stores for multiple devices and markets* [1].

The main business of the company is developing smart user applications for customer APIs. Unfortunately, every customer API was developed in a unique way and has their own communication protocol. Consequently, it is difficult to write smart user applications that use many customer APIs because a developer would have to write the communication layer for every customer API.

The problem can be defined as follows: There are a lot applications written for different operating systems and they require frequent communication with many customer APIs. What architecture is the most appropriate for communication between applications and external services represented by a customer APIs?

One of the solutions, is a simple direct communication between client and services. The client will send requests to every service and wait for responses. This solution has several drawbacks: a developer would have to maintain many communication layers written in different languages. When the external API changes, the developer would be forced to rewrite every communication layer. This increases the cost of development and the development time. Another problem is security, some services might have a private API that should not be used directly.

A different solution will be to introduce the middle layer between the client and external services. The client will communicate with middle layer using predefined protocol, middle layer will then gather data from external services and send it to the client. This approach has several benefits: every application will be written with standardised communication API, defined by the middle layer; if external API changes, only middle layer should be rewritten; the response time can be decreased by caching the data in the

middle layer.

Another solution can be the usage of CDN (Content Delivery Networks). The application will send the requests directly to the external services, but the request will be intercepted and handled by the CDN Edge Servers that are located between Application and External Services. This solution requires dynamic usage of HTTP Cache-Control headers. The advantage of using CDN is that the developer need not maintain the middleware service. Alternatively, the results will be cached somewhere on the way between Application and External services producing a faster response to the final user. On the other hand, the drawbacks are that it is difficult to configure CDN to process dynamic content and the developer would need to write the communication layer for every external service.

This project tries to break the connection between applications and external services by providing the middle layer. The applications communicates with the middleware using a predefined format that is the same for every platform. The middleware will manage the requests, translate them to the format appropriate for the external services and forward them to the appropriate servers. It will also manage the responses from servers and send them back to the applications.

1.2 Project Objectives

The project is in the context of networking, caching and data aggregation. The goal of the project is to investigate different solutions for implementing a caching layer used in a backend middleware, design and develop prototypes for each solution, run set of tests and select the most optimal solution. The selected solution is then further developed and integrated into an existing middleware provided by the company, and an analysis is then made in a real life scenario.

1.3 Thesis Outline

During the project the middleware server developed by company will be examined. The application will be studied and analysed in development environment. The set of problems and future improvements will be retrieved and discussed. The appropriate solutions for selected problems will be presented in this paper.

The Theory section will give the information about the modern web caching technologies and how they can be used on practice. Also, some parts of HTTP protocol will be introduced.

The Architecture overview will present the company solution and will identify the drawbacks. The web caching solution for some problems described in architecture overview will be introduced.

The HVG chapter will present the second part of the thesis: Hierarchical VMO Generator.

The Test chapter will compare the performance of Redis cache and Web cache solutions and evaluate the HVG algorithm in comparison with existing solution.

2 Theory

2.1 Content Delivery Networks

Nowadays the Internet is available from almost any place in the world. It allows people from different countries to communicate with each other and exchange information. The content providers and applications should serve the user's requests from all around the world with equally high speed in order to provide good user experience. In order to solve this task, companies should deploy their servers all around the world in multiple data centers. For world-class companies like Google and Netflix this task is solvable[13], but for middle-sized companies there should be another solution because it is very expensive and sometimes complicated to deploy servers in multiple data centers. Fortunately, the content delivery networks solve this problem[10].

The Content Delivery Network(CDN) is a distributed system consisted of many servers deployed all around the world in different data centers. They act as a local content holders. The routes to the content providers and applications are configured to lay through the CDN's servers. When the client wants to access the server through the Internet, the request will be processed by the local CDN server; if it will find the data locally, it will deliver the response to the client without making the request to the content provider that can be located in different country. The CDNs can store both static and dynamic content. One of the first global solutions was presented by Akamai company[10]. The graphical architecture is depicted on figure 2.1.

The CDN consists of several parts: routing, load balancing, and web caching[11]. The web caching part will be reviewed because it is relevant to the project.

2.2 Web Caching

Web caching is the technique that allows to store temporary content that is requested from the Internet. For example, HTML pages, JSON, XML or CSS files are part of temporary content. It alleviates the server's work by reducing the amount of requests to it and reduces bandwidth usage[5]. A web cache system serves as a communication point between client and sevrer. The client's requests and server's responses are routed through it.

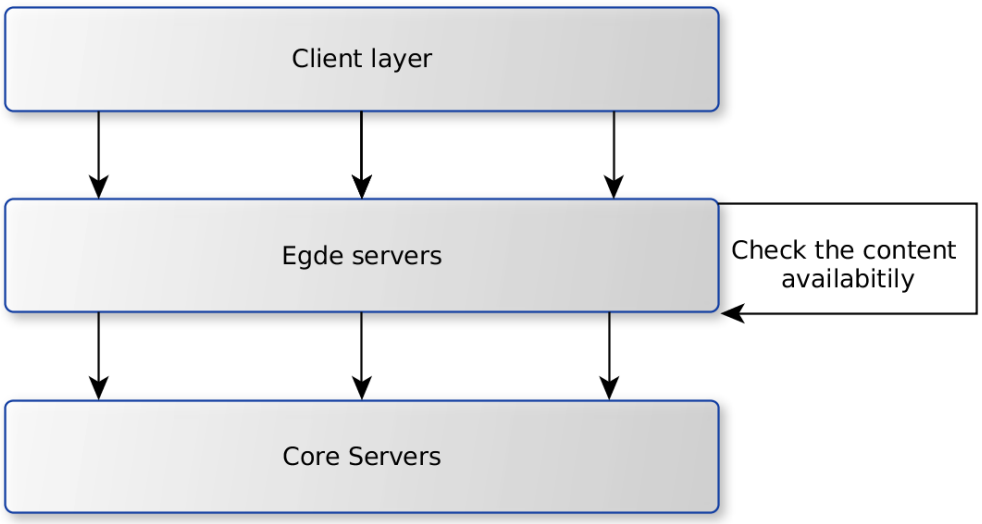

Figure 2.1: Content Delivery Network Overview

The web cache stores responses from the server and returns them without hitting the underlying server. It also can manipulate the request/response headers.

The benefits of web caching:

- Reduces the server workload; Using web caching the requests will go through the cache and will touch server only if the data does not present in caches or the data is stale. That will reduce the amount of requests made to server.
- Web caching improves user experience; The data will be delivered faster to the end users.
- Reduces bandwidth

2.2.1 Content types

The content stored by web caches can be static or dynamic.

The static content is a set of resources that stay the same no matter what was the user input. Static objects are identified by the unique path and can be cached for a long time by the web caches.

The dynamic content is generated at run-time, based on the user input[4]. The dynamic requests are almost always processed by servers. They are the main consumers of the web server's resources. The dynamic requests are time-dependent meaning that at different times the same request can produce different output; as a result, they cannot be cached for long periods of time by the web caches. Moreover, for security reasons, usually dynamic objects that contain client's personal data and preferences are configured in order to not be cached by the external web caches and CDNs.

However, almost all dynamic content is static for a short period of time. It changes only when the internal resources are altered. The internal resources can be represented as a databases. This gives a possibility to configure web cache for storing dynamic objects.

2.2.2 Http Cache Control Headers

Web cache is controlled by the http cache control headers[12]. The control mechanisms can be specified on both request and response sides. There are several http headers that are relevant for the project:

- Cache-Control
- Vary header
- Etag, If-None-Match
- Last-Modified, If-Modified-Since
- Expires (for http 1.0)
- Widely used extension headers, like X-Cache.

There are two caching techniques: Time-based caching and Data-based caching.

The time-based caching is represented by the next http headers: *Last-Modified* and *If-Modified-Since*, *Cache-Control* with *s-maxage* parameter and *Expires* header. The client sends the request with specified *If-Modified-Since* header, where he indicates as a parameter the time when the content

was modified. The server will then check the content type which was requested and send the corresponding response. If the static content was requested, the server will check if the resource was changed since the date specified by the client in *If-Modified-Since* header, and will send the new content with the code 200 and updated *Last-Modified* header if it was modified, otherwise it will reply with the code 304 (resource not modified) without content and with old *Last-Modified* header. The server does not usually set the Last-Modified header to the dynamic resources because it is hard to know where exactly they were modified. During the response the server can specify the *Cache-Control* header with s-maxage parameter. In this case, if the content is public, it can be cached by the web cache servers for s-maxage time specified in seconds. If the next request will occur in the next *s-maxage* seconds, the content will be served from the web cache. This technique is used for caching both dynamic and static content. The difference between static and dynamic content is that the s-maxage parameter for the dynamic content is set dynamically, while it is set statically for the static content.

The data based caching is represented by the *Etag* and *If-None-Match* headers. It is supported since in HTTP/1.1. On the first request, the server will compute the hash of the content and send it to the client in the Etag header. The client will remember the hash value in the *If-None-Match* header. When the next request occurs, the server will recompute the hash of the content and will compare it with the value specified in the *If-None-Match* parameter. If the values are the same, the server will reply with the 304 code, without content and with the old Etag header; Otherwise it will change the Etag header to the new value and send the normal response.

The web cache can be implemented on:

- Client side, by using browser caches
- Proxy servers, by introducing the middle caching server between client and server.
- Server side, by implementing cache programmatically

2.2.3 An overview of proxy server types

A proxy server is a server that is deployed between client and server. It acts as a middle point in communication between clients and servers.

The proxy server redirects requests to servers and responses from servers to clients. It can improve the performance of the servers by storing the copies of frequently used resources. When a client makes a request to the server through the proxy server, the proxy server serves as a web cache, it will try to find data locally and will return the resource back to the client with success.

There are two main types of proxy servers: forward and reversed proxy[3]. A forward proxy is one of the most common types of proxy servers. The client is aware about the proxy server and can configure requests through it. A reversed proxy is deployed by server administrators in the internal network. The client contacts the desired server, but the request is routed through the reversed proxy server. In this case, the client may not know about the underlying proxy. The reversed proxy server was selected for the project because it perfectly suits the architecture, the client should not know about the existence of the middle point. (the web caching is mostly done by using reversed proxy servers).

2.3 Http Session Management

HTTP protocol is stateless by its nature, meaning that there is no possibility to distinguish one request from another. Http requests usually open new connection to the desired server every time. Nowadays, servers can specify *Keep-Alive* header in the response in order to give browsers a hint that this connection can be used again for the new request. Unfortunately, without transferring user-specific information it is impossible to distinguish users.

The session management is implemented through header fields *Set-Cookie* and *Cookie* headers. When a server wants to distinguish one user from another it sets the unique identifiers for the user. This identifier is transferred in the header field Set-Cookie. The browser will parse this field and remember the unique identifier. For every new request, the browser will send this identifier its header field Cookie. Of course, the server can send additional information in Set-Cookie header that is unique for user. Unfortunately, it is very dangerous to transfer private information this way. The server can store a dictionary of user ids and corresponding private information in memory and retrieve this information every time when the browser specifies

Cookie header.

3 Middleware Architecture

3.1 Server Architecture

The company architecture contains the following components: Client Application, Middleware server, Middleware Cache, Metadata Server, and Content Servers. The brief architecture overview is presented on figure 3.1.

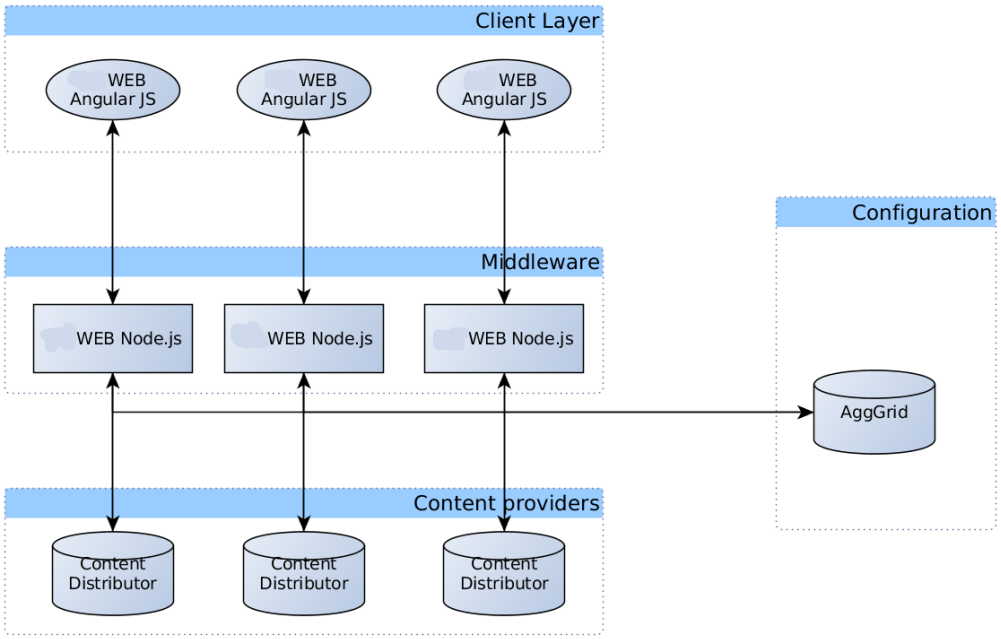

Figure 3.1: Global architecture overview

The VIA WEB Angular.js represents is a web application developed using Javascript, HTML, CSS framework and Model View Controller pattern. In the system it is called client application. It communicates with middleware server through REST services based on HTTP protocol. The client has several components: Controllers, Managers, Services.

The contollers validate the user data, invoke corresponding managers, and render data to the view objects represented by HTML pages. The managers are implemented using Facade pattern[6]. They hold and manage services, construct View Model Objects from service responses, and send them back to controllers.

The services are communication layer between the client application and the middleware server. They communicate via REST services based on HTTP protocol. This brings flexibility to the architecture and makes components loosely coupled.

The VIA WEB Node.js is a middleware server. It serves as a data aggregator and security point. It translates requests from the client application to the format understandable by the metadata server or content distributors, gathers data from content distributors, builds data model objects(the definition is given in the next section) and sends the as JSON entities to the client application.

Content servers (content distributors) are customer servers. They are the main sources of information that need to be presented by the client application. They can be represented, for example, by the movie entities or music distributors.

The metadata server is the company-developed server that stores and processes auxiliary and configurational information. The requests can be:

- Data aggregation - required for analytical purposes
- User action logs
- User settings - specific user settings, (i.e watch history for movie entities)
- Client and middleware configuration

The interaction between components can be described as follows: the client application initiates the process by sending requests to the middleware server. The middleware server then redirects requests to the underlying servers (content distributors or metadata server), builds data model objects, and sends them to the client application. The client request can be one of two types, configurational or data demand. If the request is configurational, the middleware server redirects it to the metadata server; otherwise, it redirects the request to the Content server. The middleware supports local cache and caches every response that is not associated with the user session or unique and private user data. The message diagram between architecture components is depicted on figure 3.2.

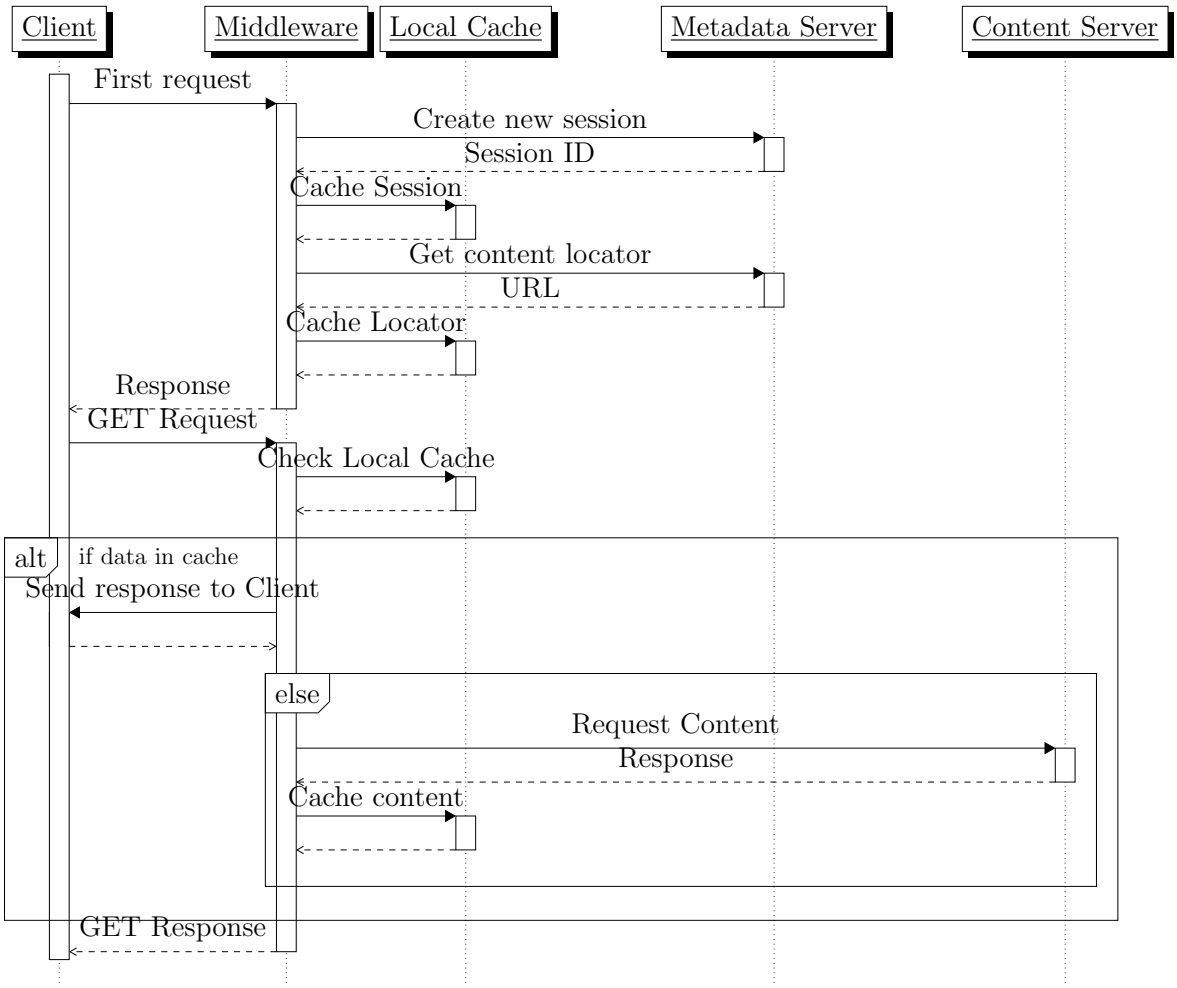

Figure 3.2: Sequence Diagram of message exchanging

3.2 Model objects

In order to understand the existing architecture several definitions should be introduced: Data model object, View Model Object and Application model object.

The data from the content servers differ from one to another; as a result, the structure for representing content server objects should be generated dynamically. The asset from the content distributor that have a dynamic structure is called *Data Model Object*(DMO). For example, the DMO can represent information about videos, music, or any other entity. The DMO is generated by the DMO builders from JSON by the middleware server.

The middleware server sends DMOs to the client application. The client application gathers several DMOs and constructs *View Model Object*(VMO). This object is then presented to the users as a view asset. As a result, the VMO can be defined as a set of data model objects. A VMO

is constructed for each HTML page.

Each content distributor stores the finite set of assets. This finite set of assets is called *Application Model Object*(AMO). Therefore, the application model object is the array of View Model Objects. The AMO represents the information and objects that can be fetched from the single content server.

3.3 Middleware server architecture

The middleware server is developed using server side Javascript language and asynchronous server Node.js [9]. The server is developed using Model View Controller (MVC) pattern and communicates with other components through REST services. As a result, the server components are loosely coupled with each other which gives great flexibility in changing and replacing components and simplifies testing.

The middleware contains the following components: Controllers, Managers, Services, Configuration and Data Model Object builders. The interraction between components is presented on figure 3.3.

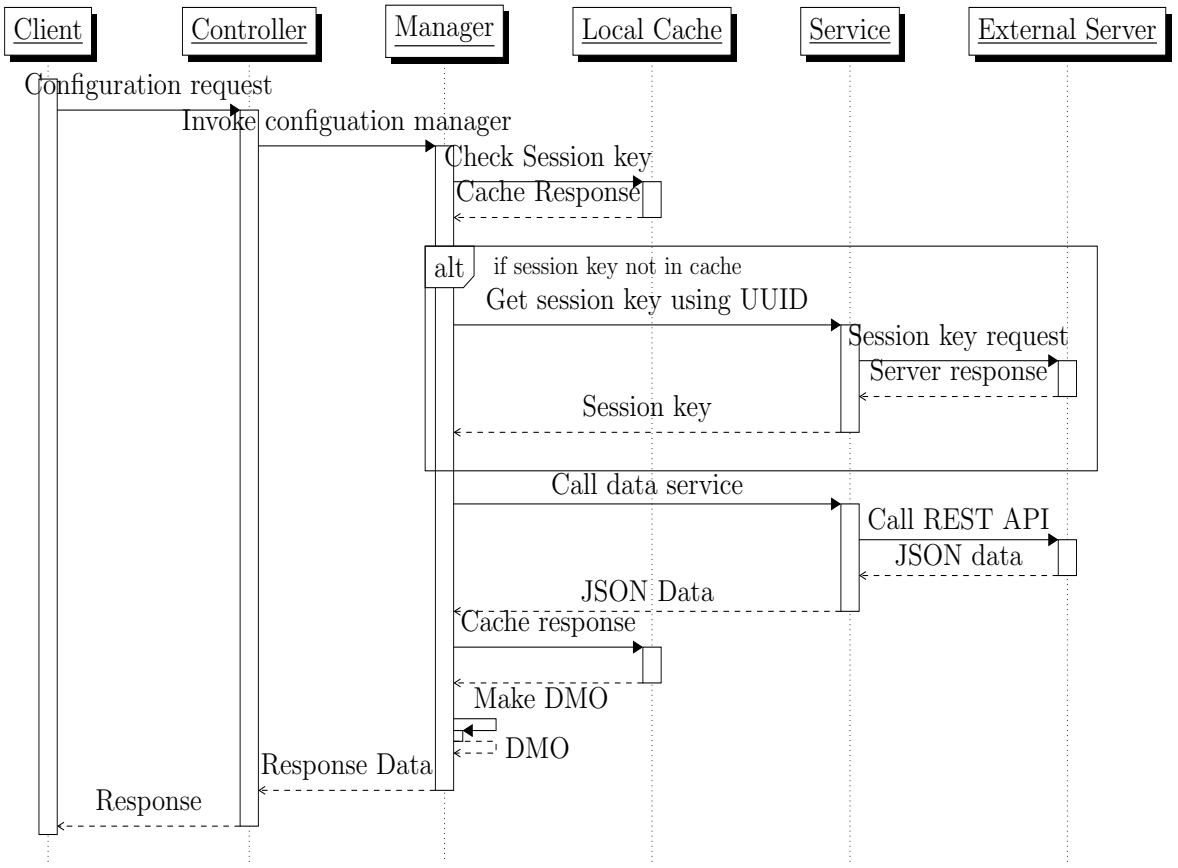

Figure 3.3: Sequence Diagram of Middleware Server Request Process

The controllers accept requests from clients. They validate user data and invoke corresponding managers.

The middleware server managers are similar to the client application manager components: they implement facade pattern, aggregate multiple services, and redirect requests to them. They also gather the data from services and build immutable Data Model Objects (DMOs). These DMOs are sent back to the client as responses. The workflow of managers is depicted on figure 3.4.

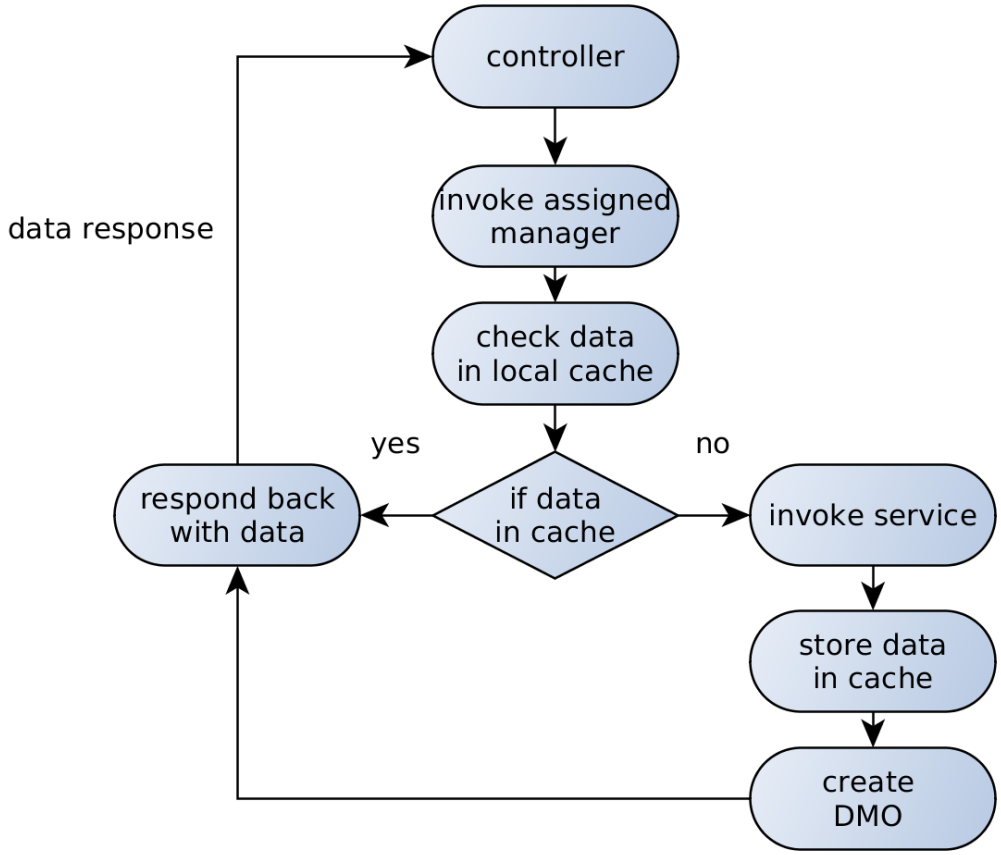

Figure 3.4: Manager workflow

Services communicate with metadata and content servers. They aggregate the cache layer and react according to the following rules: they check if the data presents in the local cache. If service observes cache hit, it will check the object TTL(Time To Live; i.e the time -usually in seconds- that represents how long the object can be considered fresh without hitting database) and send the corresponding object back to the manager. On the other hand, if a cache miss occurs, it will send the GET request through the REST protocol to the metadata server or content server, store the response locally for predefined period of time, and send it back to the manager. The

workflow of services is presented on figure 3.5.

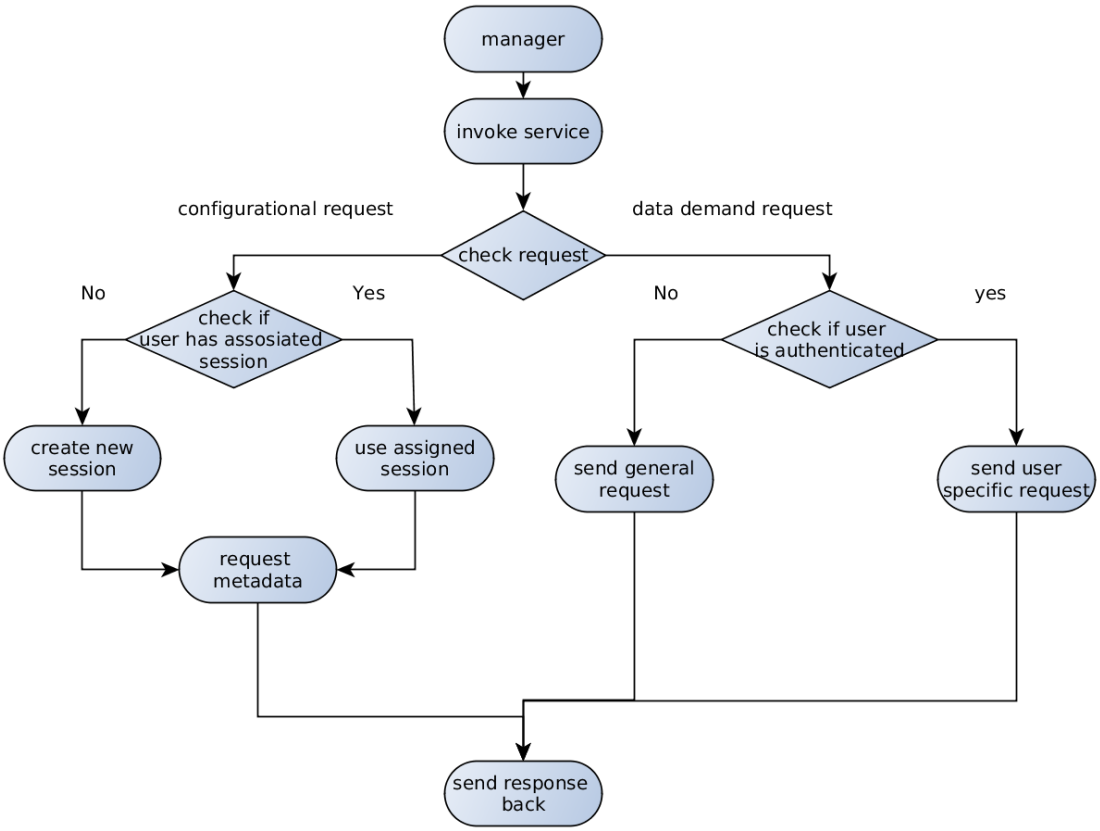

Figure 3.5: Service workflow

The client application can make two types of requests: configurational and data demand requests. The configurational request can have the following purposes:

- Provide configuration parameters for both middleware server and client application
- Write user activy
- Check the health of metadata server
- Get Content Server url
- User settings and preferences
- Analytics

The data demand request is a request to the content servers. Since content servers are customer servers, the response doesn't have predefined structure and can have any structure specified by customer service.

The control flow can be described as follows: the controller receives the request from client application, validates data in the request, and redirects it to the assigned manager. The middleware server manager invokes corresponding service (i.e. communications layers). When the client application makes the data demand request, the manager invokes content services that redirect request to the assigned content server. The response is then stored in the cache, translated into Data Model Object(DMO), and transferred back to the client in the JSON format. The DMO builders are in charge of translating JSON data into persistent Data Model Objects. For every DMO there is an assigned manager and service. As an example, if the content server responds with video object, there will be VideoManager and VideoService components.

The client layer and middleware layer are loosely coupled and communicate with each other through REST services. This gives the application great flexibility.

3.4 Session management

In order to process communication between middleware server and inner servers (metadata server and content distributors), a security layer was implemented (i.e. session management).

The session consists of two parts:

- The client session
- The middleware session

The purpose of sessions is to distinct users and provide corresponding analytics to the customers.

The client session is represented by the unique session identifier (UID) and the browser's ID (the string that identifies browser version in the internet). These parameters are generated by the middleware server and transferred to the client through the Set-Cookie header. The browser remembers the data and sends it back with every request in Cookie header.

The session is generated per client when he makes the initial request. It contains the metadata session key and user session, if the user is authenticated. The metadata session is obtained by making the request to the metadata server. The middleware server sends the application key parameter, which is specified in the configuration file and browser ID. The metadata server validates the application key and generates a new session for the middleware server. The middleware server assigns this session to the client and stores it in local memory. Each time the client will communicate with the middleware server it will send the client's session data. The server will find the metadata session associated with the client and retrieve the metadata session. Using this session the middleware can make configurational requests to the metadata session. The sequence diagram of session management is presented on figure 3.6. The application key is given by the system administrator to each middleware server. The browser ID can be any string and does not have validation rules.

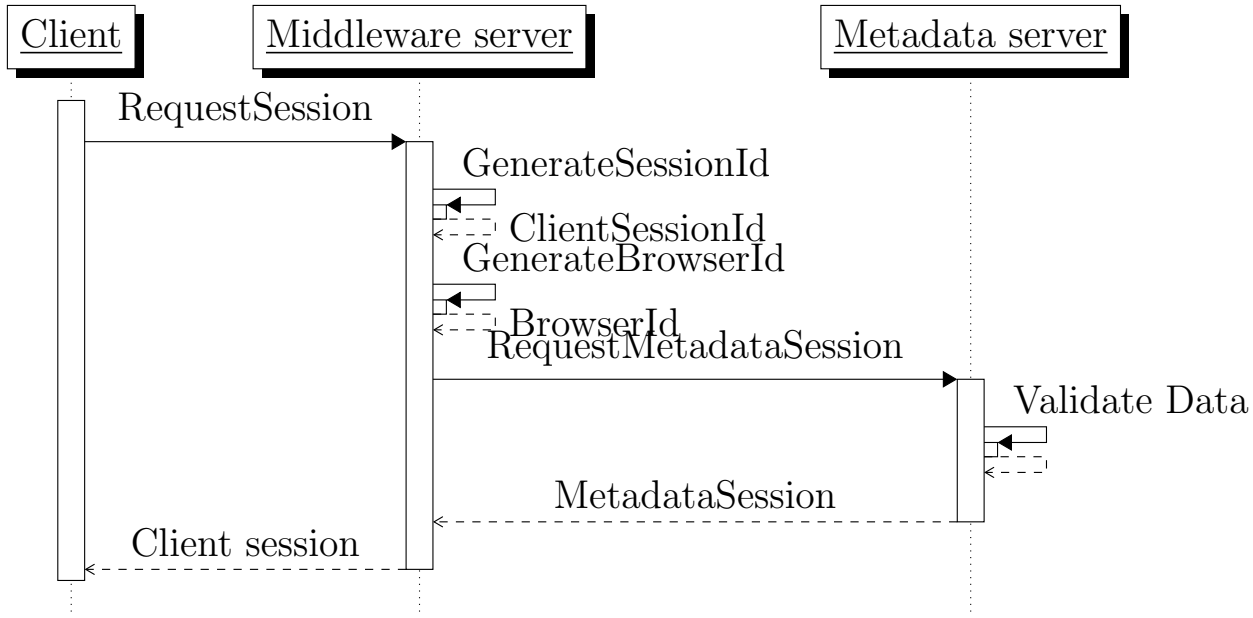

Figure 3.6: Sequence Diagram of Middleware Session generation

3.5 Drawbacks of the current architecture

After careful examination, two categories of problems were defined: Client application drawbacks and Middleware side drawbacks.

In order to present the page, the client needs to generate a View Model

Object(VMO). The VMO contains several Data Model Objects (DMOs). The client makes a request for every DMO, aggregates the response, generates VMO from DMOs, and renders it to the HTML view. The drawback is that the client has to make several HTTP requests in order to generate single VMO. It would be better for the client to make a request for VMO instead of DMO. This approach has some advantages: the client will make less HTTP requests which will increase the performance by reducing the latency and it will simplify the client logic considering the client will not be required to generate VMOs from DMOs.

Another problem with the current client implementation is that it is not generic. If a new content server is introduced, a lot of code will have to be changed on the client side in order to implement the new logic. To solve this, the client can maintain the caching layer that will cache VMOs from the responses.

Additional complications occur when the client implements MVC pattern, which produces duplication with the middleware server. This approach increases the complexity of the system, since the developers would have to support both the client and middleware MVC applications. To rectify this, we can simplify the client application and assign two tasks to it: caching and rendering VMOs.

On the middleware side, the DMOs are not generic. The purpose of the middleware server is to serve as a transparent layer, but without dynamic DMO generation a lot of code has to be changed when the new content server is introduced.

The middleware cache can be replaced by the Content Delivery Network(CDN). The middleware caches only information that is common for every user. This work can be done by the CDN edge servers. These will decrease the middleware complexity and decrease the cost of maintaining middleware server.

4 Model and Methodology

The next two chapters describe the methodology and proposed solutions to improve the performance of the current company middleware server. In particular, the model consists of three parts: replacing the Redis cache server with web cache solution, reducing the amount of requests for generating View Model Objects, and removing dublication of the MVC pattern. This section describes the solutions for replacing Redis cache and the next section describes the algorithm for reducing the requests to the middleware server and removing duplication.

4.1 Reversed proxy cache

The figure 4.1 shows the amount of requests and time that browser needs in order to generate a single VMO and render a page. The browser needs to make more than twenty AJAX requests for a single page. The middleware was deployed on the local machine meaning there is no latency between the browser and middleware server. As can be seen, these are not optimistic numbers. The amount of requests are too high and computation has to be done on the both client and server sides. Several questions arise:

- Is Redis a good caching layer for this project?
- Can Redis be replaced by something else? Maybe it would be better to use the configurational cache (represented by the web caches) and control it through the http cache control headers.

Let's consider system architecture using reversed proxy server instead of Redis cache. The client application will make http requests through the reversed proxy server. The proxy server will decide whether the content is stale or not and will act as a web cache. If the content is stale (non-cacheable) the proxy server will redirect the request to the middleware server. Otherwise, it will reply to the client application without touching the middleware server. The benefit of this solution is that instead of contacting server and then caching the object, the cache is contacted first and then the server. In theory, it should give the performance and flexibility. The Redis cache could be replaced by the Content Delivery Network solution in future.

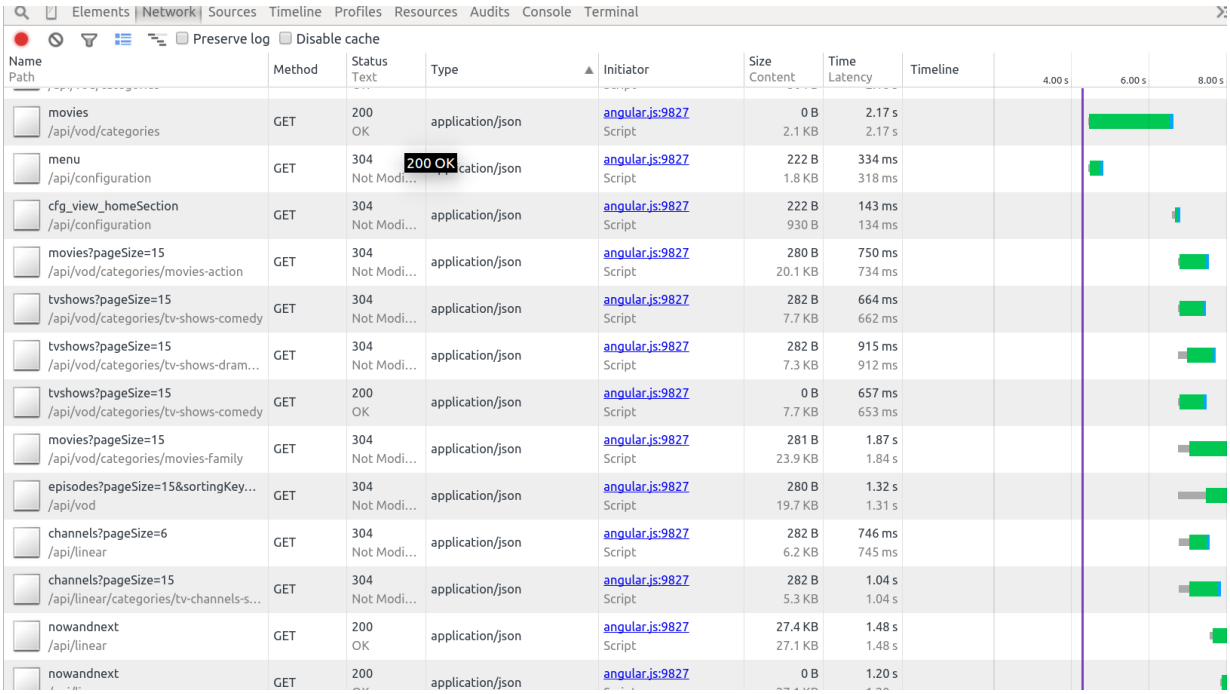

Figure 4.1: Example of page generation by browser

4.2 Web cache selection

The web cache for the project should satisfy several parameters:

- Be Configurable- The client application sends requests both for DMOs and for writing user actions. The web cache should not cache the requests for writing user actions. The sessions still should be supported in order to write user actions.
- Be Responsive, Easily Maintainable- The web cache should be deployed easily and provide statistics and internal logging.
- High performance- The web cache should be transparent and must not increase the request time and latency if the cache object was not found locally.

There are many web cache solutions available in both commercial and open source. For this project only open source solutions were considered. The initial candidates were: Squid, Varnish, Apache server with proxy module, Nginx server as a cachable reversed proxy, and Apache Traffic Server.

The Squid server is a forward proxy server, but it can be configured as a reversed proxy server. Squid is preferably used for storing static content.

The Varnish was developed as a reversed proxy server from the beginning. It is fast, reliable, and lightweight. It uses Varnish Configuration

Language (VCL) for configuring and describing the data workflow in cache. The VCL is translated to the C code and compiled to a shared object which is then dynamically linked into the server process. It is a powerful tool that helps to set up Varnish as a dynamic reverse proxy server.

The Apache traffic server was developed by the Yahoo group and eventually moved to the Apache Incubator [8]. According to Yahoo Inc., Apache traffic server can handle more than 400TB of internet traffic per day and works as a forward as well as reversed proxy server. It has a growing community and is continously improved upon. The configuration is simple and consists of changing several files. All these, make Apache traffic server a good candidate.

Other solutions (Nginx and Apache server) were not originally developed to be proxy servers, but have additional modules that one can install and configure. They are not well-configured and work worse than the solutions described above [8].

A thorough comparison and performance evaluation of Varnish and Apache Traffic Server can be found in [2].

4.3 Web cache configuration

Before performing execution and comparative study, reversed proxy servers should be properly configured. They should aggregate and store requests that contain public data and skip analytical requests and requests with private user information. For example, payements.

Proxy servers should also work with http sessions. Usually, when the session is specified (the set-cookie http header included in the server response), proxy servers are transparent, meaning they are skipping these requests and not storing them in memory. As was described in previous chapters, the metadata server uses http session for analytical purposes. This means that even anonymous users will have a unique session.

In order to solve this problem, the Varnish was configured to replace Cookie header with X-Cookie header. This gives the possibility for Varnish to store the requests and still have the analytical requests available. The metadata server was modified in order to treat X-Cookie header as a Cookie header.

4.3.1 Varnish configuration

The varnish logic is configured using Varnish Configurational Language(VCL). The varnish logic works as a deterministic state machine consisting of several states:

- vcl_init – the initial state
- vcl_recv – the initial point for the request to varnish
- vcl_pass – in this state the request is passed to the backend server
- vcl_hash – the varnish computes the hash of the request in this state
- vcl_hit – occurs when the hit is observed on varnish server
- vcl_miss – occurs when varnish has stale data or has no data at all
- vcl_fetch – when the document is acquired from the backend server
- vcl_deliver – when the cache object is about to be delivered to the client
- vcl_error – occurs when the error is observed

The Varnish configuration is presented in Appendix A.

The Varnish server starts from the command line with the appropriate arguments. The Varnish uses the following arguments:

- -a : the backend server address in format ADDRESS:PORT
- -b : the address in format ADDRESS:PORT
- -f : the path to the VCL file
- -F : if specified, the varnish will be started in the foreground
- -n : the varnish working directory

4.3.2 Apache Traffic server configuration

The Apache Traffic server configuration consists of several files:

- cache.config – the caching behaviour is specified here
- remap.config – is used for overall configuration
- cluster.config – is used for configuring Apache Traffic server in distributed mode

The Apache Traffic server uses the command line to start. The obligatory parameter is –httpport which specifies the listen port. The Apache Traffic server consists of three parts:

- traffic_cop – supervises process for traffic_server
- traffic_manager – monitors the traffic_server and gathers the logs
- traffic_server – responsible for starting the web cache server

The detailed configuration can be found [7]

5 Hierarchical VMO Generator

5.1 View model objects

The company's solution was developed in order to provide flexibility and customization to the end user. The middleware server retrieves all configurations from the metadata server. The configuration can be user specific information as well as global information. For example, the location of content distributors. The middleware translates the responses from the content distributors to Data Model Objects. They are the immutable objects that are transferred to the client's application. On the other hand, the client application operates with View Model Objects. They are used to render HTML pages. Single HTML page can contain single View Model Object. The VMO is constructed from multiple data model objects and some additional information. For example, the location of the image.

One important drawback can be observed: application duplication. The middleware server and the client application have the same pattern of execution: they both operate with data and build new data patterns from existing ones. The difference is that the middleware server is doing it with content distributor responses and the client aplication is processing middleware responses. Because of this situation, the client application should make several HTTP requests for retrieving necessary DMO objects from the middleware server. The example of the single page generation is on picture 5.1. As can be seen, in order to generate a single main page, the client application has to make about ten Ajax requests. What if there was a possibility to generate the VMO objects on the middleware side? Would there be a possibility to get rid of the duplication? What should be done for it? This section will describe the solution for removing duplication of MVC pattern and reducing the amount of requests to the middleware server.

5.1.1 VMO properties

Typical DMO and VMO objects are presented in Appendix E. As can be noticed, a VMO object is a JSON dictionary build from DMO objects. The VMO supports following properties:

Generic; The DMO objects are specific and unique for every content distributor. They are built according to the specific rules in the middleware

server. The VMOs are generated from DMOs according to another set of rules specified in the client application. As a result in order to move VMO generation to the server side some descriptive language should be introduced that lets developers to describe the VMO structure.

Hierarchical; DMOs are basically data from content distributors. That means that the DMOs can be depend on other DMOs. For example, the figure [add DMO dep. figure] shows that in order to build a movie list DMO, first the DMO that represents movie categories should be generated, then for each movie category the set of movie objects should be fetched; Only on the third step can the list of movies be build. These means that the VMOs have a hierarchical structure. The typical VMO is presented in figure 5.1.

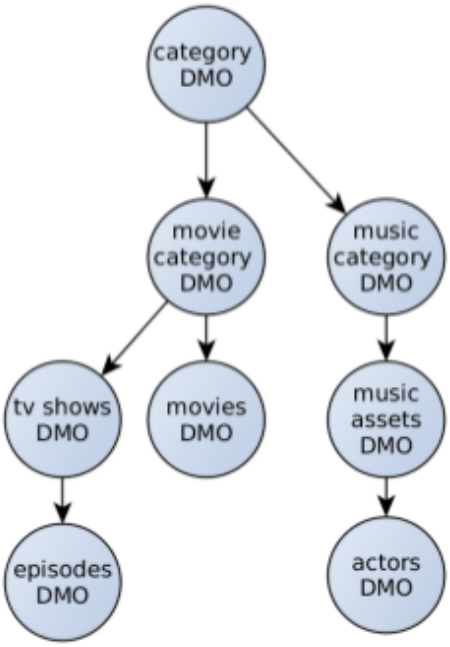

Figure 5.1: VMO from the set of dependent DMOs

5.2 Suggested Middleware server architecture

In the current architecture, the single instance of the middleware server is deployed for a single content distributor. This is done due to the specifics of content distributors and data that they are providing. The following question arises: Is there a possibility to move the individual logic that requires to processes data to the client application while keeping the middleware as generic as possible? The theoretical benefits of this approach are: there will need to be a single middleware server or cluster deployed. This will simplify

the deploying system; instead of supporting middleware for every content distributor, there will be just one. The new architecture is presented on figure 5.2. As can be seen, the configuration server (Appgrid) is now treated like a content distributor. The benefit is that we do not need to specify separate logic for it.

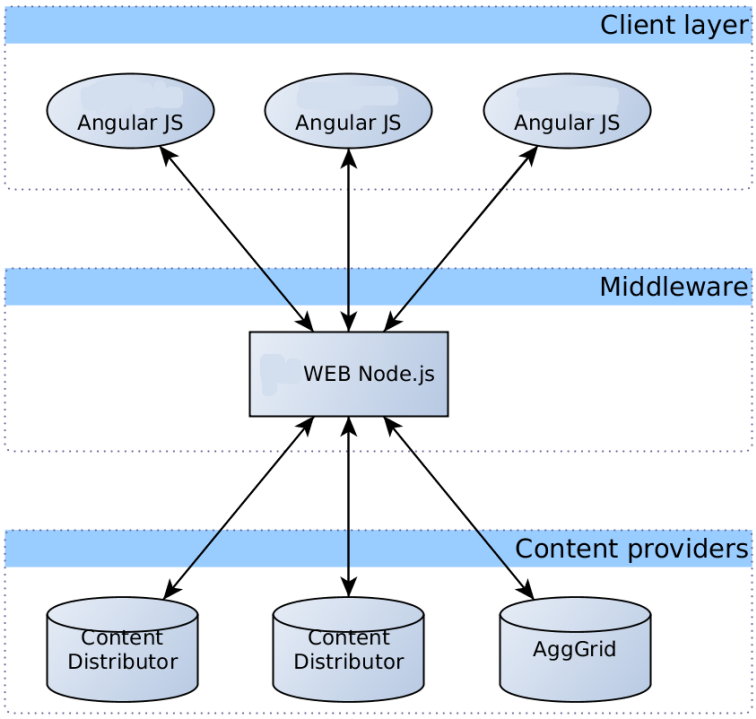

Figure 5.2: Manager workflow

Let's consider the architecture from another prespective. The middleware server receives queries for the VMO generation. It supports several content distributors (databases) and makes queries to the content distributors for retrieving data model objects (tables). The middleware server resembles a relational database system. The comparison of database management system and middleware server is presented on figure 1

5.3 Approach description

Let's consider relational database management systems and applications that are using them. Usually, there is a single instance or cluster of databases installed and deployed on servers, depending on the amount

Database	Middleware server VMO generation
serves multiple clients	should serve multiple client applications
supports creation of new databases	should support insertion of new content servers
supports creation and alternation of tables	should support creation and alternation of new endpoints
supports dynamic queries to the desired databases and tables	should support queries to the content distributors and their endpoints

Table 1: Comparison of Database System and Middleware Server properties

of requests it should serve. Nobody makes a new database instances for a single client, because it is neither efficient nor effective. In order to get data from the database the queries are executed. The SQL queries are basically the rich descriptive language that is interpreted by the database and executed for specified tables. The data is provided in two forms: array of table rows and array of view rows. Tables are the single source of information that resembles DMO. Views are the mixed information from tables that looks like VMO.

In order to solve the problem of duplication and reduce the amount of requests, the hierarchical VMO generator(HVG) was developed. The HVG is basically the descriptive language that developers are writing in a JSON format. The JSON is then translated into the GET HTTP or POST HTTP request and sent to the middleware server. The middleware server parses the request, builds the asyclic graph, and fetches the corresponding data model objects. The data model objects are then combined together and sent as a response to the client application. The client application executes specific logic on the data received from server and builds HTML pages.

5.4 Path in VMOs

This section introduces the definition of path in the View Model Object. The main format for data transferring is JSON. The JSON format consists of two main objects: array and dictionary. An example of dictionary:

Listing 1: The example of JSON dictionary

```
{key1:value1,key2:value2,key3:{key1:value1}}
```

An example of array:

Listing 2: The example of JSON array

```
[value1,value2,value3:{key1:[value1,value2]}]
```

Very complex entities can be built using combinations of these two objects. Let's introduce the definition of path: The path is the route to the specific object or a set of objects in a JSON entity. The examples of path are presented below:

Listing 3: The examples of objects in different routes

```
Object: {root:{child:value}}
Path: root.child
Value: value

Object: {root:[{key:key1, value: value1},{key:key2,value:value2}]}
Path: root.key
Value: [value1,value2]
```

As can be seen, the path identifies a single object if it lays though a set of dictionaries. However, if the array is found on the route, all elements will be traversed and a set of objects will be extracted.

5.5 Hierarchical VMO Generator overview

In order to solve the problem of multiple requests and middleware server duplication, the descriptive hierarchical VMO generator is implemented. It consists of three modules: HVG fetcher, HVG query parser, and HVG query builder. In order to build and process VMOs, two data structures are used: content provider dictionary and query dictionary.

The content provider dictionary has a tree-based structure and its scheme is presented on 5.3. Rectangles indicate the single instance of object appends, while parallelograms indicate the dictionary of objects. The content provider dictionary contains a dictionary of content distributors. When the query is passed to the middleware server, the locations of necessary content distributors are retrieved from this dictionary. It is used by the HVG fetcher module.

The key is a unique identifier that is used by query objects (as described below). Each content provider consists of two parts: location and a dictionary of resources. The location is a domain with corresponding scheme.

Input values:	*{id:[value1,value2,value3]}*
Endpoint:	*http://testdomain.ext/{id}*
Result:	*[http://testdomain.ext/value1*
	http://testdomain.ext/value2
	http://testdomain.ext/value3]
Endpoint:	*http://testdomain.ext/{id*}*
Result:	*http://testdomain.ext/value1,value2,value3*

Table 2: URL Template examples

The resource has one field: endpoint. The endpoint is a path in URI [RFC to URI] with several embedded templates. The template can be one of two types: {*param_id*} and {*param_id**}. The first type indicates that only one value can replace the template. If an array of values is given, it will produce the array of resolved endpoints. On the other hand, the second type produces a single endpoint, even if an array of values is given. The examples of templates are presented in table **??**.

The example of content provider map is given in Appendix C.

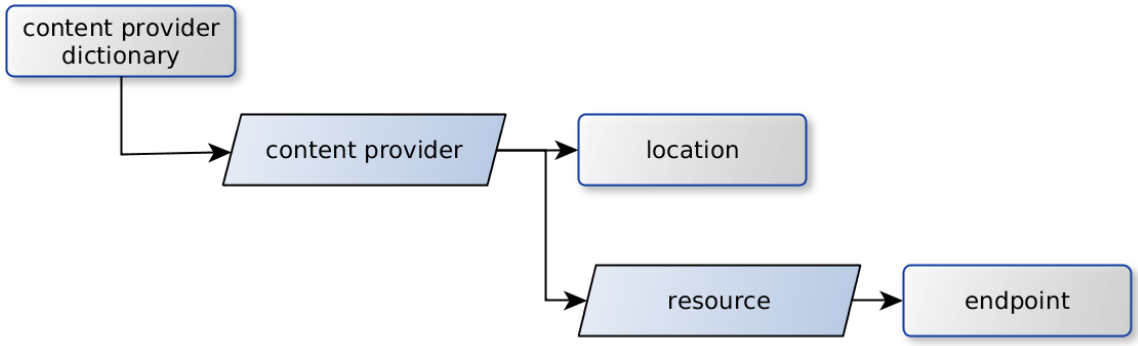

Figure 5.3: Content Provider dictionary structure

The query represents the request from the client application to the middleware server. The tree-based structure is presented on figure 5.4. The single query consists of several query objects.

The query is built on the client application using HVG query builder. When the query is finished, it is translated into a corresponding JSON format and sent to the middleware server.

Each query object may have up to three parameters: *content_distribu-*

tor, *path_parameters*, *query_parameters*. The obligatory parameter, or *content_distributor*, corresponds to the key in the content provider dictionary desrcibed above. Through it, the set of resources is retrieved from the content provider dictionary. The *path_parameters* and *query_parameters* are similar to each other, as a result only one of them will be described. The *query_parameters* is a dictionary of parameters, the key corresponds to the template parameter in resource object described above. Each path parameter has a type field. It can have one of two values: *constant* or *dependency*. If the value is *constant*, only one additional field should be specified: *values*. This field serves as a container of values that should replace the corresponding template in the endpoint.

On the other hand, if the value of *path_parameter* is *dependency*: the query object depends on another query object, the list of categories should be requested first. That means that in order to request a query object, the parent object should be requested first and then the specified dependency variables for the template parameters should be retrieved. In this case, additional parameters should be specified: *parent*, *parent_parameter_type*, *path* and *property*. The *parent* specifies the query object identifier of the parent object. The *path* is a JSON path to the object as described above. The *property* is a key in the dictionary retrieved by using *path* parameter. The *parent_parameter_type* can be one of two values: *key* or *constraint*. If the value is *key* then the dependency variables will be extracted using path: *path+property*.

If the value of *parent_parameter_type* is *constraint* then the objects in *path* that obey constraints will be retrieved. In this case, the additional parameter should be specified in corresponding path parameter: *constraints*. The *constraints* is an array of constraints. Each constraint has two parameters: key and value. The key specifies the name of the property that the object should have and the value specifies the value of the property. The summary of parameters is presented in the Table 3.

Hierarchical Query Parser(HQP) is a module that translates client requests into JSON format. The available methods are presented in table **??**.

Table 3: Description of Query Object's parameters

Parameter	Description
content_distributor	the location of resource dictionary in provider
path_parameters	dictionary of variables that will be retrieved in runtime and replace the corresponding template parameters in endpoint for building a request URL
type	the type of path_parameter, can be constant or dependency
parent	the parent ID of query object. Note: specified only when type=dependecny
parent_parameter_type	the parent parameter type, can be key or constraint. Note: specified only when type=dependency
path	the path for the objects in parent response. Note: specified only when type=dependency
property	the name of property that contains dependency value. Note: specified only when type=dependency
constraints	the array of constraint. Note: specified only when type=dependency and path_parameter_-type=constraint

Table 4: Description of HQP module

Method Name	Parameters	Description
select	object ID, content distributor ID	creates an empty query object with specified content distributor ID
setConstantParameter	parameter ID, values	appends the path parameter which has a type *constant* to the query object's path_parameters dictionary
setParentParameter	parameter ID, parent query object, path , property	appends the parent parameter to the query object
setQueryParameter	key, value	appends query parameter to the query objects query_parameters dictionary
addConstraints	path parameter ID, array of constraints	appends the constraint array to the query object's path patrameter
build		translates query objects into JSON format

The example of HQP usage is given in Appendix D.

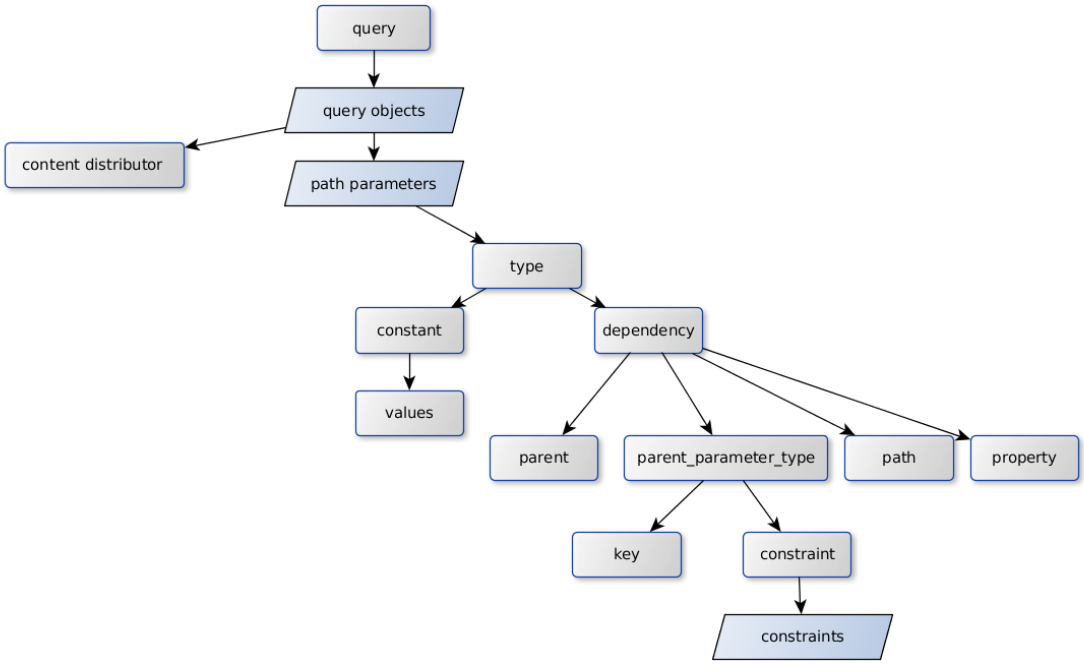

Figure 5.4: Query structure

The HQL supports two level caching. The first level caches the data model objects. It lays after parsing the client request strictly before requesting data from the content distributor. The second level cache lies on top of the execution process and is touched before executing the client request.

5.6 HQL workflow

The developer writes the request using HQL parser methods. The request is translated into a JSON format that contains a dictionary of query objects. The JSON is sent as a POST HTTP request to the middleware server. The middleware server builds the asyclic graph (if there were dependency cycles it will generate error). The direct asyclic graph is then traversed using breadth first search algorithm, on every node (query ob-

ject) the data is fetched from the appropriate content distributor (if the parent data has already been fetched) and is written into the output array. The output array is then sent as a response to the client POST request.

The schematic workflow of HQL fetcher and HQL query parser is presented on fiqure 5.5.

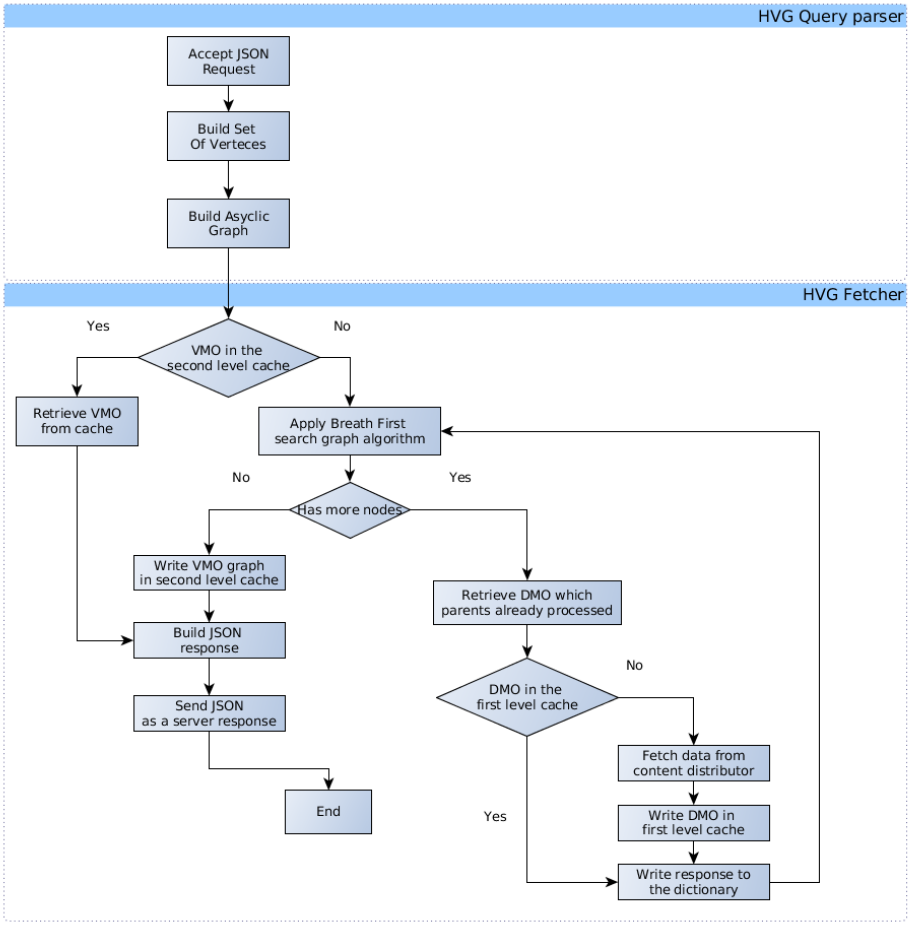

Figure 5.5: HVG workflow

5.7 Alternatives to HVG

The middleware server can be used in order to build one of three object types: Data model object, View model object or Application model object. The initial implementation was responsible for building DMOs. The current implementation builds the VMOs.

The DMO and VMO were described above. The Application Model Object(AMO) contains a set of VMOs that are specific for a content distributor.

The alternative to the dynamic VMO generation on the middleware side can be a generation of AMOs on the middleware server. However, it is not a good idea because the size of object provided by content distributors can vary a lot and some of them can be very heavy. If the full AMOs would be transferred to the client application, a big amount of memory would have to be utilized by both client application and middleware server. In theory, it will dramatically decrease the user experience, but it could be checked as a continuation to this project.

6 Testing

6.1 Testing tools

The Jmeter Testing tool was used during the testing part of the project.

JMeter is a tool that helps measure the performance of applications that are using HTTP protocol. The Jmeter configuration consists of several test plans that each represent the set of actions made by clients. This approach gives necessary flexibility for this project. Several test plans that emulate requests to the main VMO were developed. For the current solution the tests plan consists of nine GET requests through HTTP protocol that return data model objects. On the other hand, for the solution with HVG, the test plan sends single POST request with JSON data that describes relations between DMOs. The Jmeter configuration is presented in Appendix B .1. JMeter also builds the plot and the table of the request that was made with the corresponding time parameters.

6.2 Performance Comparison of Redis and Web caches

During the project experiments, several configurations were conducted. The list of configurations is presented below:

- Middleware server with Redis cache
- Middleware server with Varnish web cache
- Middleware server with Apache Traffic server cache
- Middleware server with HQL and second-level cache
- Middleware server with HQL, second-level cache and data compression
- Middleware server with HQL, second-level cache and first level cache
- Middleware server with HQL, second-level cache, first level cache and data compression

As can be seen, the tests are divided into two parts: configurations for testing internal cache replacement and configurations for testing VMO

creation. The first part analyzes and evaluates performance of redis cache and web cache servers. The second part evaluates HVG performance.

Each experiment consists of several parts and can be described as an algorithm below:

- Configure Jmeter test plan
- Run Jmeter Test plan on predefined set of parallel request (which emulate users)
- Evaluate Summary Report

For each test, the next configuration was used: Jmeter emulates one hundred users that are making continuous requests for fetching the view model object of the main page.

6.3 Performance evaluation of Redis Cache and Web cache solutions

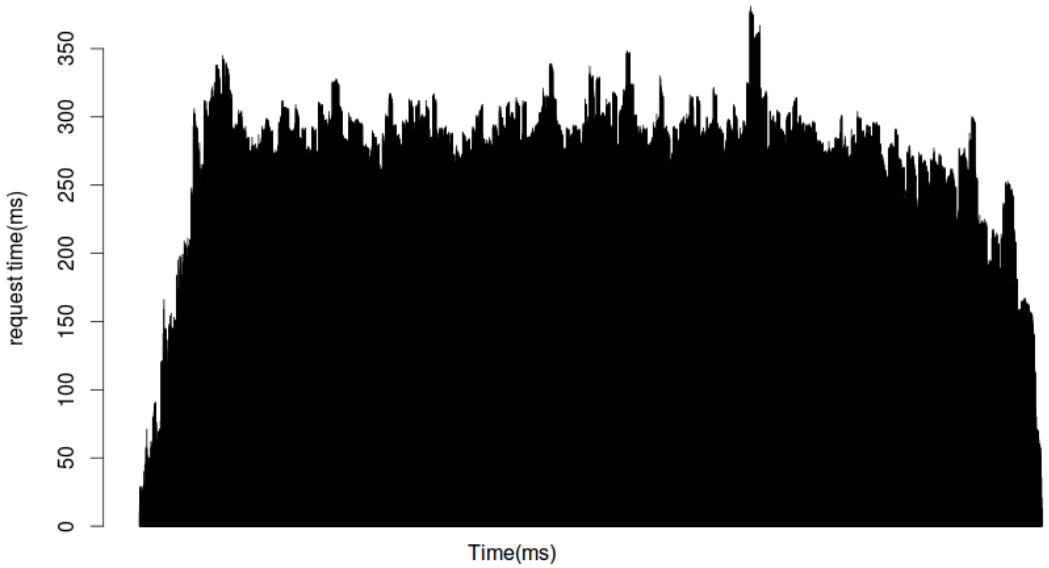

Figure 6.1: Response time of middleware server wihth Redis cache for generating VMO through parallel requests

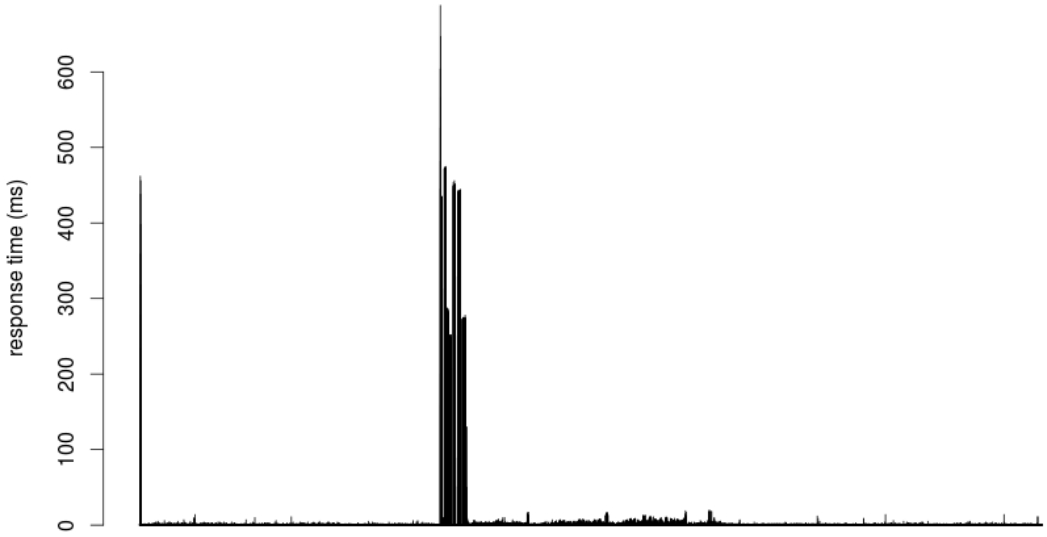

Figure 6.2: Response time of Varnish web cache for generating hierarchical VMO

This type of test emulates the user requests for generating view model object for the main page. In order to build a VMO, several requests should be done for gathering data model objects. The request for building a single page consists of nine REST subrequests through HTTP protocol for gathering DMOs. The corresponding Jmeter test plan is presented on figure 6.3.

Figure 6.1 illustrates server response time for generating VMO that has no dependencies between objects. As can be seen, the maximum response time is about 300 ms. Figure 6.4 shows response time from middleware server for generating hierarchical VMO that tree structure is shown on 6.5.

The VMO has tree structure with depth equal to three. The initial node is always called *root*. It has no data and is introduced for convenience to "glue" other nodes in a tree. As can be noticed, it is the longest path (Root → Episodes → TvShows → TvEpisodesMetadata) and a has depth of four; As a result, three sequential requests should be made for generating data (the root node is excluded because it has no data). As shown, the performance decreases about three times. It is quite predictable because it is necessary to make three sequential requests. Figure 6.2 shows the VMO

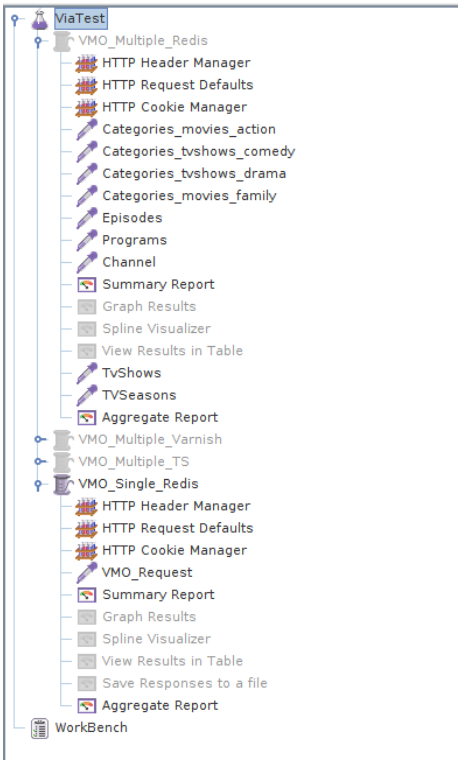

Figure 6.3: The Jmeter test plan for current solution and proposed solution with HVG

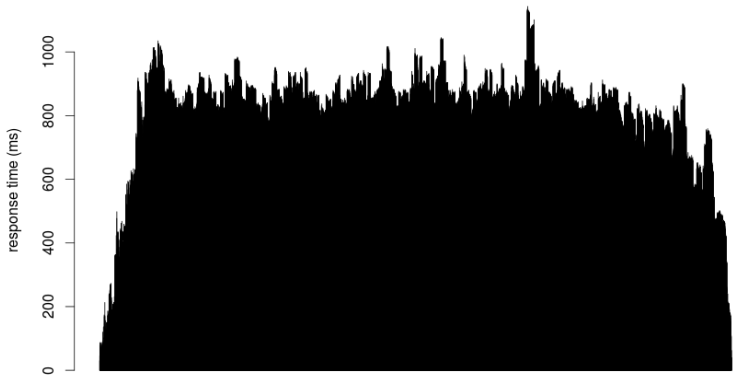

Figure 6.4: Response time of middleware server with Redis cache for generating hierarchical VMO through sequential requests

generation through Varnish web cache. The response time decreases dramatically. The reason for this is that it takes constant time to get data from the Varnish web cache. The request for getting DMO rarely touches the middleware server. It is noticable from the figure, when the request goes through varnish web cache to the middleware server the response time

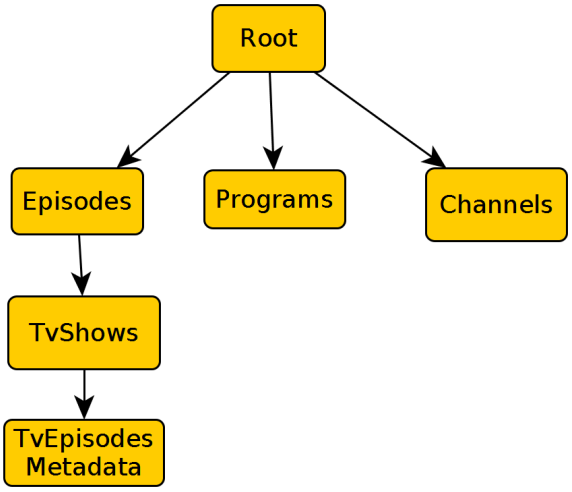

Figure 6.5: View Model Object that is used during testing

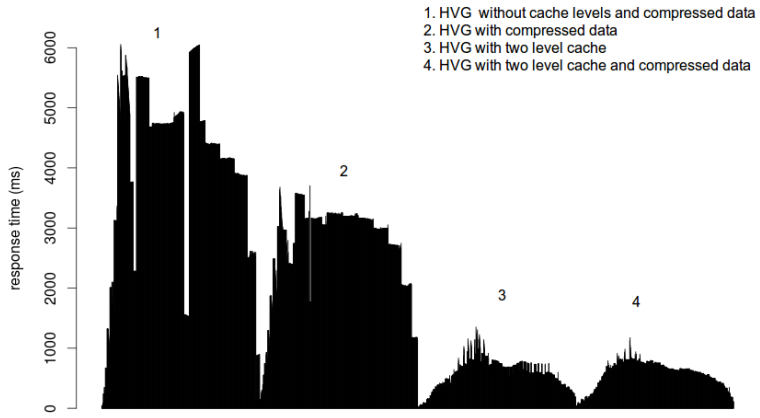

Figure 6.6: Response time of middleware server for generating hierarchical VMO through HVG

increases. Figure 6.7 shows the response time for making requests through Apache Traffic server configured as a web cache. The results more predictable than the results of varnish web cache. The maximum time does not reach 100 ms.

6.4 Performance evaluation of HVG with different configurations

Figure 6.6 shows the response time for a hierarchical VMO generator with different configurations. As can be seen, there are four hills, each

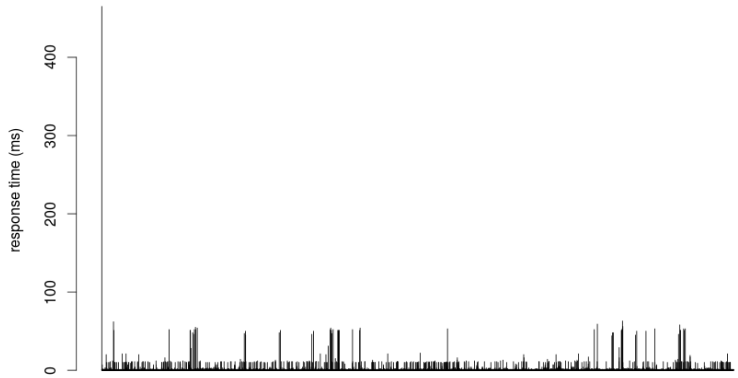

Figure 6.7: Response time of Apache Traffic Server for generating hierarchical VMO

hill represents the corresponding experiment: HVG with first level cache (caching data model objects), HVG with first level cache and data compression, HVG with first and second level caches (caching both data model objects and view model objects) and HVG with first and second level cache and data compression. The first hill appears to be the slowest one, responses come in around 6000ms in the worst case. The behaviour is quite unpredictable, because the vmo requests consume 100% of CPU and the amount of data that is transferred between client and middleware server is quite big.

The second hill is almost twice as fast as the first one. The reason for that being the reduced amount of data transferred between client and server. The compression reduces the amount of data by almost five times. As can be seen, the compression does not produce overhead and slows responses for a negligible time. In addition, the hill looks more predictable and has less valleys.

The third hill shows the response time from middleware server with two level caches. As can be seen, the server performs three times faster than hill number two and almost six times faster than the first hill. This happens because the middleware server rarely executes requests for building VMOs. It does checks if VMO is already computed every time before executing actual request.

The last hill represents the response time with second and first level caches and data compression. It can be noticed that the hill is more stable

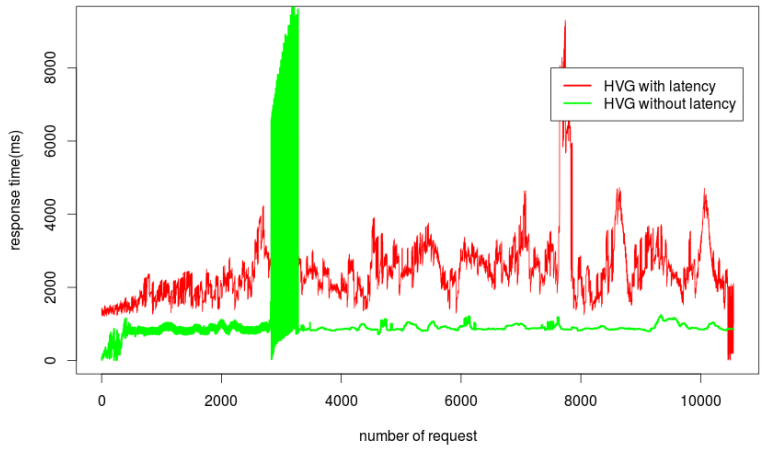

Figure 6.8: Comparison of response time with and without latency for optimal HVG configuration

than the third one, but the execution time is roughly the same.

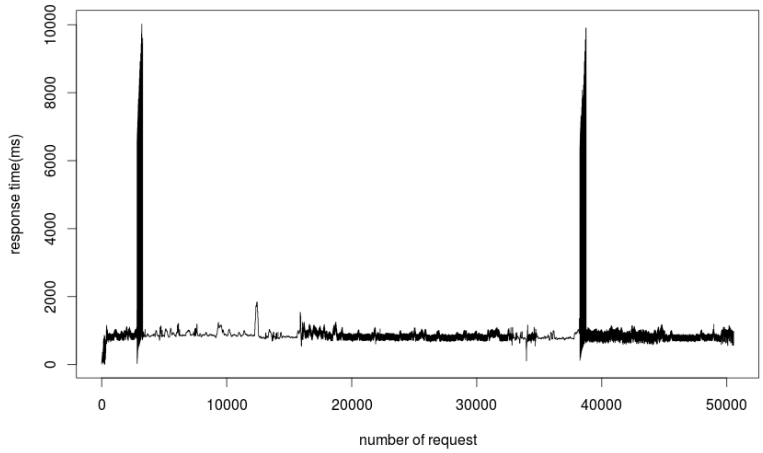

Figure 6.9: Load test of optimal HVG configuration

The brief result is that the optimal configuration for HVG is two level cache and data compression.

Figure 6.9 demonstrates the load test for optimal HVG configuration. During the test around 50000 (fifty thousand) requests were made in one hundred parallel threads. There are two peaks that can be observed that have response time around 10.000 msec. During these requests, the middleware server requested data from the underlying content distributor. There it can be seen that the mean response time is around 1500 msec.

The tests executed up until this point were conducted without intro-

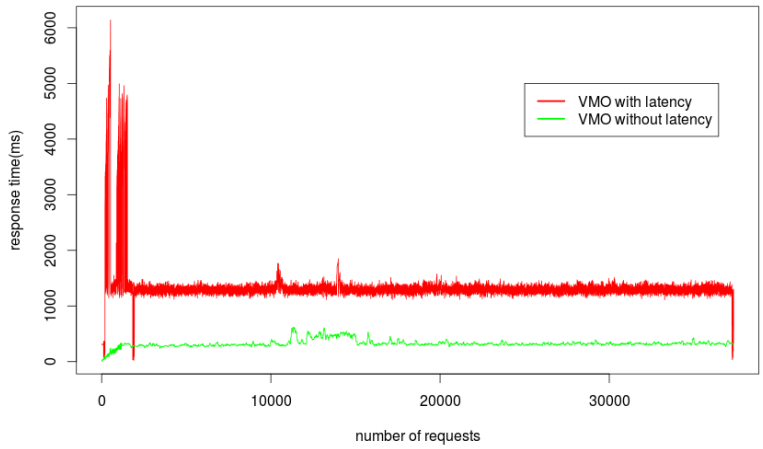

Figure 6.10: Comparison of response time with and without latency for VMO generation without dependencies using multiple requests

ducing latency. The next obvious step is to make an artificial latency and observe how the server response time depends on it. The artificial latency is about 500 msec between client and server. Figure 6.8 shows the comparison between responses with artificial latency and without. As can be seen there is one big peak on the green plot, is appeared because several events happened at the same time: the data in the redis cache became stale and the redis server could not write data fast enough due to lack of resources. The red plot also has one peak, the reason for this is the stale data in redis cache and corresponding request to the content provider.

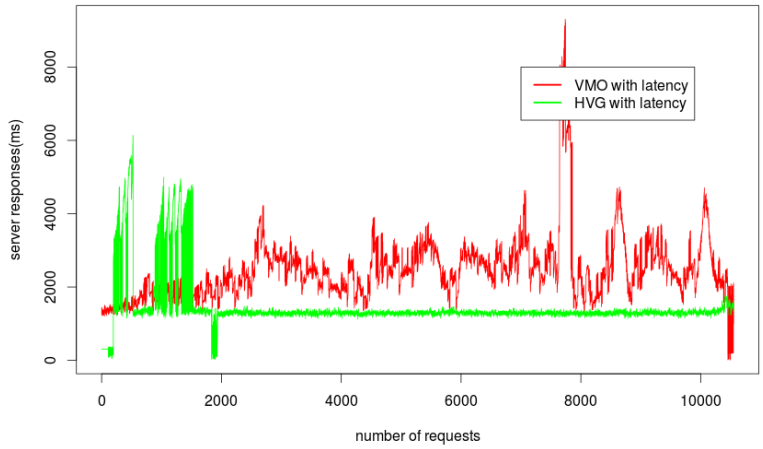

Figure 6.11: Comparison of response time between parallel VMO generation and HVG with optimal configuration

The next figure 6.10 shows the comparison between responses with and without latency for generating VMO through multiple requests. There it can be observed that the server made several responses to the content distributor; As a result, the request time is about 6000 msec. In some cases the response time for requests with latency is less than the response time without latency. It is a small persentage of errors during the experiments. When these errors occur, the server immediately sends an empty response.

Figure 6.11 shows the comparison between HVG with optimal configuration and VMO generated through multiple requests. As can be seen during the HVG test, several requests were made to the content distributor. The red plot also shows that the VMO through multiple requests also did several requests because of the stale content. Both plots had a small error during the experiment; as a result, server responses with approximately 10 ms can be observed. The HVG with optimal configuration shows great performance compared to the VMO through multiple requests. In addition, it is more stable and reliable.

7 Discussion and Conclusion

Several conclusions can be made using the results of previous chapter:

- The Hierarchical VMO Generator(HVG) works slower than company's solution without first level cache
- The stability of HVG depends on the amount of transferred data: the less data the more stable responses
- Without latency (that was introduced artificially) the HVG with optimal configuration works roughly with the same speed as current company solution where VMOs are generated on the client side
- With latency the HVG solution works faster

The [2] gives the detailed evaluation of Apache Traffic server and Varnish server; As a result, the comparison between them is omitted in this project.

The HVG solution consumes more CPU and memory because it stores the graph of data model objects plus the fetched data from content distributors. It uses computationally intensive breadth first search algorithm on directed graph. On the other hand, the client side VMO generation sends multiple simple requests to the middleware server. They are light requests, that consume small amount of CPU power and memory and is used just for storing the final dta model object.

Fortunately, the algorithm for generating HVG is executed rarely because most of the time the data is retrieved in the VMO cache. The drawback of this approach is that the data is stored two times: one as a part of Data Model Object, another as a part of View Model Object. The Data Model Object cache could be turned off, unfortunately these will give additional problems: if the VMO will be stored for the time equals of maximum TTL among all DMOs, some DMOs will become stale and the server response will be irrelevant. Taking this in consideration, the maximum TTL for VMO is computed as:

$M = \min_{1 \leq i \leq N} D_i, \{i = 1 : N\}$, N - amount of DMOs that need to be fetched in request.

Theoretically, the solution with HVG does not work well when the set of DMOs have TTL that dramatically differs from one another. In this case, the overhead produced by VMO computation will be greater than the latency for retrieving data model objects, but in general it requires additional study.

7.1 Further studies

The Application Model Object(AMO) should be studied and corresponding conclusions should be produced. For small VMOs, perhaps it would be more optimal to transfer compressed AMO.

The Web cache solution requires additional study in terms of selecting the appropriate algorithm for calculating TTL (time to live) of VMO/DMO objects. Currently the TTL is set to the default time that was not selected through careful study. The maximum time for VMO equals the TTL of DMO that has minumum value. This is one of the proposed algorithms, however this question is not studied enough and requires futher work.

References

[1] Accedo Broadband AB., *Accedo broadband ab.*

[2] Shahab Bakhtiyari, *Performance evaluation of the apache traffic server and varnish reverse proxies*, (2012).

[3] Katia Way Admiralty Rey Marina Barish, Greg Obraczka, *World wide web caching: Trends and techniques.*

[4] Matt Copeland, George Mcclain, *Web caching with dynamic content.*

[5] J. Rabinovich M. Gadde, S. Chase, *Web caching and content distribution: A view from the interior*, Computer Communications **24** (2001), 222–231.

[6] Richard Johnson Ralph E. Vlissides John Gamma, Erich Helm, *Design patterns: elements of reusable object-oriented software*, (1995).

[7] Apache inc., *Apache traffic server documentation.*

[8] Yahoo inc., *Yahoo's cloud team open sources traffic server*, November 2009.

[9] Node js inc., *Node.js asynchronous server side framework documentation.*

[10] Ramesh K. Sun Jennifer Nygren, Erik Sitaraman, *The akamai network: a platform for high-performance internet applications*, SIGOPS Oper. Syst. Rev. **44** (2010), 2–19.

[11] Rajkumar Pathan, Al-mukaddim Khan Buyya, *A taxonomy and survey of content delivery networks*, Grid Computing and Distributed Systems GRIDS Laboratory University of Melbourne Parkville Australia **148** (2006), 1–44.

[12] J. Reschke R. Fielding, M. Nottingham, *Rfc 7234: Hypertext transfer protocol (http/1.1): Caching*, 2014.

[13] F. Hao M. Varvello V. K. Adhikari, Y. Guo, *Unreeling netflix: Understanding and improving multi-cdn movie delivery.*

Appendices

Appendix A

Listing 4: Varnish Configuration

```
vcl 4.0;

backend default {
    .host = "127.0.0.1";
    .port = "9000";
}

sub vcl_recv {
#       set req.http.X-Cookie=req.http.Cookie;
#       unset req.http.Cookie;
        if(req.url ~ "^/api/vod/" || req.url ~ "^/api/asset/" ||
           req.url ~ "^/api/health/" ||
                req.url ~ "^/api/linear/" || req.url ~ "^/asset" ||
                    req.url ~ "^/partials/" ||
                req.url ~ "^/extensions/" || req.url ~ "^/
                   bower_components/" || req.url ~ "^/scripts/" ||
                   req.url ~ "^/images/"){
                set req.http.X-Cookie=req.http.Cookie;
                unset req.http.Cookie;
                return(hash);
        }
        if(req.url ~ "^/api/configuration"){
                set req.http.X-Cookie=req.http.Cookie;
                unset req.http.Cookie;
                return(hash);
        }
        return(pass);
}

sub vcl_hash {
        hash_data(req.url);
        if(req.http.host){
                hash_data(req.http.host);
        }
        return(lookup);
}

sub vcl_hit {
        return(deliver);
}

sub vcl_miss {
        return(fetch);
}
```

```
sub vcl_pass {
        return(fetch);
}

sub vcl_backend_response {
    # Happens after we have read the response headers from the
       backend.
    #
    # Here you clean the response headers, removing silly Set-
       Cookie headers
    # and other mistakes your backend does.
        if(bereq.url ~ "^/api/configuration"){
                #unset bereq.http.Cookie;
                set beresp.ttl = 24h;
        }
    return(deliver);
}

sub vcl_deliver {
    # Happens when we have all the pieces we need, and are about to
        send the
    # response to the client.
    #
    # You can do accounting or modifying the final object here.
        return(deliver);
}
```

Appendix B

Appendix C

Listing 5: Content Provider Dictionary

```
{
  'accedo.ovp': {
      'url': 'https://ovp-staging.cloud.accedo.tv',
      'categories.movies': {
                'endpoint': '/category/{id}/movie'
              },
      'categories.tvshows': {
                'endpoint': '/category/{id}/tvshow'
              },
      'episodes': {
                'endpoint': '/episode'
              },
      'tvshow': {
                'endpoint': '/tvshow/{id*}'
              },
      'tvseason': {
                'endpoint': '/tvseason/{id*}'
              }
    },
  'accedo.appgrid': {
```

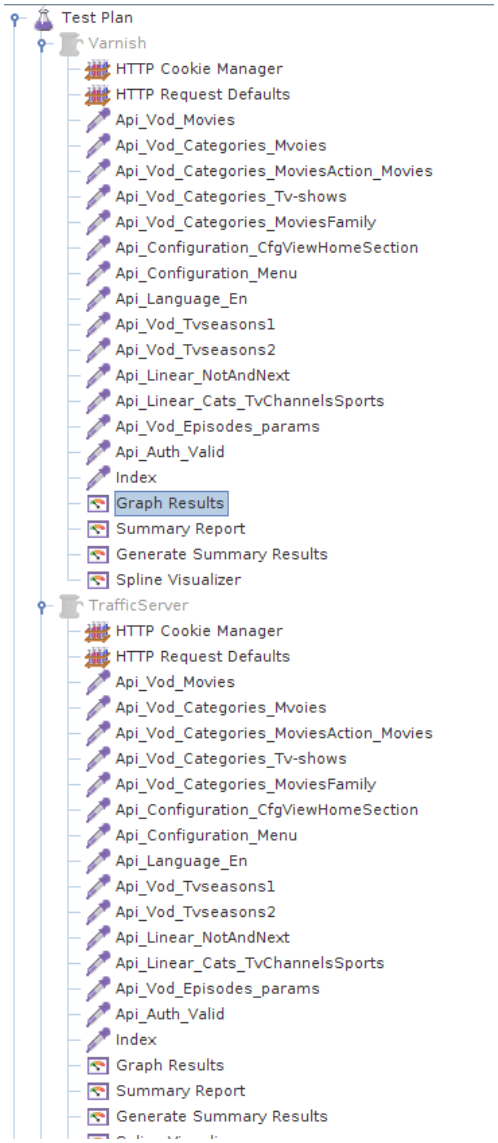

Figure .1: Jmeter Configuration

```
        'url': 'https://appgrid-api.cloud.accedo.tv',
        'metadata': {
                    'endpoint': '/metadata'
                }
    }
}
```

Appendix D

Listing 6: HQL usage

```
        var parser = new HQLParser();
        var requestJson = parser
                    .select('category_movies', 'accedo.ovp.categories
                        .movies')
                    .setConstantParameter('id', ['movies-action'])
                    .setQueryParameter('pageSize', 2)
                    .select('category_tvshows', 'accedo.ovp.
```

```
                categories.tvshows')
            .setConstantParameter('id', ['tv_shows_comedy'])
            .setQueryParameter('pageSize', 2)
            .select('episodes', 'accedo.ovp.episodes')
            .setQueryParameter('pageSize', 2)
            .select('tvshow', 'accedo.ovp.tvshow')
            .setParentParameter('id', 'episodes', 'entries.
                metadata', 'value')
            .addConstraints('id', [{key: "name", value: "
                VOD$tvShowId"}])
            .select('tvseason', 'accedo.ovp.tvseason')
            .setParentParameter('id', 'episodes', 'entries.
                metadata', 'value')
            .addConstraints('id', [{key: "name", value: "
                VOD$tvSeasonId"}])
            .build();
```

Appendix E

Listing 7: Example of VMO/multiple DMOs that were used for testing

```
      {
"movies_movies-action":{
   "content_distributor":"accedo.ovp.categories.movies",
   "path_parameters":{
      "id":{
         "type":"constant",
         "values":[
            "movies-action"
         ],
         "parent_parameter_type":"key"
      }
   },
   "query_parameters":{
      "pageSize":{
         "value":15
      }
   }
},
"tvshow    s_tv-shows-comedy":{
   "content_distributor":"accedo.ovp.categories.tvshows",
   "path_parameters":{
      "id":{
         "type":"constant",
         "values":[
            "tv-shows-comedy"
         ],
         "parent_parameter_type":"key"
      }
   },
   "query_parameters":{
      "pageSize":{
```

```
            "value":15
         }
      }
   },
   "tvshows_    tv-shows-drama-american":{
      "content_distributor":"accedo.ovp.categories.tvshows",
      "path_parameters":{
         "id":{
            "type":"constant",
            "values":[
               "tv-shows-drama-american"
            ],
            "parent_parameter_type":"key"
         }
      },
      "query_parameters":{
         "pageSize":{
            "value":1            5
         }
      }
   },
   "movies_movies-family":{
      "content_distributor":"accedo.ovp.categories.movies",
      "path_parameters":{
         "id":{
            "type":"constant",
            "values":[
               "movies-family"
            ],
            "parent_parameter_type":"key"
         }
      },
      "query_parameters":{
         "pageSize":{
            "value":15
         }
      }
   },
   "ep    isodes":{
      "content_distributor":"accedo.ovp.episodes",
      "query_parameters":{
         "pageSize":{
            "value":15
         }
      }
   },
   "tvshow":{
      "content_distributor":"accedo.ovp.tvshow",
      "path_parameters":{
         "id":{
            "type":"dependency",
```

```
            "parent":"episodes",
            "path":"entries.m    etadata",
            "property":"value",
            "parent_parameter_type":"constraint",
            "constraints":[
               {
                  "key":"name",
                  "value":"VOD$tvShowId"
               }
            ]
         }
      }
   },
   "tvseason":{
      "content_distributor":"accedo.ovp.tvseason",
      "path_parameters":{
         "id":{
            "type":"dependency",
            "parent":"    episodes",
            "path":"entries.metadata",
            "property":"value",
            "parent_parameter_type":"constraint",
            "constraints":[
               {
                  "key":"name",
                  "value":"VOD$tvSeasonId"
               }
            ]
         }
      }
   },
   "channels":{
      "content_distributor":"accedo.ovp.channels",
      "path_parameters":{
         "id":{
            "    type":"constant",
            "values":[
               "tv-channels-sports"
            ],
            "parent_parameter_type":"key"
         }
      },
      "query_parameters":{
         "pageSize":{
            "value":15
         }
      }
   }
}
```