Ondřej Papík

Communication in mobile edge-cloud environment

Department of Distributed and Dependable Systems

Supervisor of the master thesis: doc. RNDr. Hnětynka Petr, Ph.D.
Study programme: Informatika
Study branch: Softwarové systémy

Prague 2018
I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ........ date .......... signature of the author
Title: Communication in mobile edge-cloud environment

Author: Ondřej Papík

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Hnětynka Petr, Ph.D., Department of Distributed and Dependable Systems

Abstract: Edge-cloud brings the computation power as close to the clients as possible. This reduces latencies and overall computation time in the cloud. Thanks to the mobile nature of clients we must be able to migrate tasks among different servers. The goal of this thesis is to examine possible problems in communication and propose the architecture of framework. Our framework uses gRPC and is written as module to it. It is platform independent, uses reliable communication and focuses on easy usage. We provide implementation of this framework with some example uses.

Keywords: edge-cloud gRPC mobile devices
I want to express my gratitude to those who supported me while writing this theses.

Especially I want to thank my supervisor who helped me with many advices and planning of this thesis.

Another thanks is to my family for giving me great environment and for their support.
# Contents

1 Introduction 3

2 Communication technologies 5
   2.1 Background and analysis ................................. 5
   2.2 Running example ........................................ 7
   2.3 RPC ....................................................... 8
   2.4 gRPC ....................................................... 9
      2.4.1 Protocol Buffers .................................... 11
      2.4.2 Channel ............................................... 11
   2.5 Load balancing ........................................... 12

3 Edge-cloud framework 15
   3.1 Requirements ............................................. 15
      3.1.1 Network communication .............................. 15
   3.2 Architecture ............................................. 16
      3.2.1 Central and edge server ............................ 16
      3.2.2 Client ............................................... 17
      3.2.3 Adaptation server .................................. 21
      3.2.4 Time computation .................................. 23
   3.3 Metamodel ................................................. 26
      3.3.1 Application requirements ............................ 29
      3.3.2 Network ............................................... 29
   3.4 Communication data ...................................... 30
      3.4.1 Adaptation server - client .......................... 30
      3.4.2 Computation server - adaptation server .......... 31
      3.4.3 Computation server - client ........................ 32
   3.5 Data sharing ............................................. 32
   3.6 Recovery ................................................. 34

4 Evaluation and related work 36
   4.1 Visualization ............................................. 36
   4.2 Face recognition ........................................ 37
      4.2.1 Real-time ........................................... 38
      4.2.2 Disk .................................................. 40
   4.3 Related work ............................................. 41

5 Conclusion 42

Bibliography 44
1. Introduction

The mobile devices are getting more and more used. We are processing a lot of data there. But some complex tasks cannot to be computed at the mobile devices. This has many reasons. The mobile devices commonly do not have enough efficient hardware. Therefore the computation can be very slow. Even so slow that it could be unusable to its purpose. Also not all the mobile devices have a lot of spare memory. So it is necessary to keep the applications in the mobile devices as small as possible. Another thing is that the mobile devices are powered by a battery and it could be drained really quickly when doing complex computations.

These are reasons why the cloud computing is so much popular nowadays. It allows us to compute very complex tasks without buying an expensive hardware. Unfortunately the data may travel a long distances to reach the servers which leads to a greater latency. For particular applications it is necessary to have as low latency as possible. These are especially applications which have to perform real-time data processing. For example autonomous cars can use the benefits of the edge-cloud. In this case the big latency could cause injuries of people.

The big problems come especially with the IoT (Internet of Things). The number of mobile devices is still growing. Any of the devices can produce a lot of data which have to travel through the network. We need to make sure that the network is not overloaded near the servers. Also the data have to travel with appropriate speed.

To decrease the latency we are moving the computation power on the edge of the cloud, i.e. as close to the clients as possible. If the clients are not moving and we have an absolutely stable network, we could statically assign a server for each of them. We assume that the clients can be mobile and therefore we have to migrate the tasks between the servers to keep the computation as close to the clients as possible. We are doing it to decrease the latency but the close clients does not have to have the lowest latency possible. Therefore in this thesis we will look also at other factors influencing the latency and we will propose a way to migrate the tasks to decrease it maximally.

The latency is not only reason why to use the edge-cloud computing. It can be also used to increase a security. In fact there is not increased the security but decreased the vulnerability. The data are traveling much shorter distance therefore there is not so many places where a hacker could do any damage.

Although we are decreasing the latency by using the edge-cloud, we are also getting a lot of complications. For instance, the tasks have to follow up each other correctly. The tasks can migrate among the servers unpredictably. There can be failures in the network or some server can shutdown.
The main goal of this thesis is to analyze these complications and to design a framework capable of seamless communication in the edge-cloud. So all edge-cloud specific problems will be hidden in our framework and the usage will be as transparent as possible. So the programmer who uses our framework, will be able very easily to program his application without taking care of any edge-cloud details. Only reason why the programmer would have to know anything about the edge-cloud is optimizing his code to get even better latency.

Our framework is meant to be used in many applications. Therefore the framework will be universal and not designed to a specification of a single application.

The main reason to create a framework is to allow easy usage of the edge-cloud. The easy usage have priority even over the latency. If the usage of the framework would be too complicated then there would be no reason to use it. The programmer could just implement the edge-cloud communication himself/herself. Of course we have to try to lower the latency maximally anyway.

We will implement an application using our framework. This way it will be proved that our framework is working.

In the second chapter of this thesis we will look at the edge-cloud problems and what technologies we are using. In the third chapter we will describe an architecture of our framework. In the fourth chapter we will describe an exemplary usage of our framework and we will compare it to other solutions.
2. Communication technologies

2.1 Background and analysis

People are using the cloud computing very often nowadays. They all know how it works and how to use it. When we change the classic cloud computing to the edge-cloud computing then we want to look it the same. It is shown in figure [2.1]. The users just send data to the cloud and expecting responses. They do not care where exactly they are connected or where the data travel. Nobody wants to take care of too much details. The common users just want to know how fast it will be and what they have to do to get it working.

The edge-cloud in more details can be seen in figure [2.2]. We have multiple servers and each user is connected to most appropriate one. Also the servers may need to communicate with each other. This is not absolutely necessary but it allows us to share data in databases for example.

Now we have clients connected to the servers. The servers are doing their computation for the clients. If the servers count would be stable and the clients would not move, we could statically assign the server for every client. But any server may restart for any reason or just the network between the client and the server may fail. Not mentioning that the clients are mobile therefore the statically assigned server may not be the best after some time. We need to dynamically connect the servers and the clients and change the connections according to the actual situation.

One way is to implement the decisions at client side. That would mean that any client would have to need to know about the state of the whole network to decide correctly. It would rapidly increase the communication with other components in the network for the clients. Also to get some reasonable result we need to make some complex logic therefore the client could get really big. For mobile devices we want to keep the clients as thin as possible and be careful to not drain the battery too quickly.

For this reasons it is better to have as much as possible of this logic at some server. It is possible to have this logic at the same servers which are doing the computation for the clients or we can have separate servers for this tasks. We have chosen to keep it separate. The computation of the connections would slow the actual work at the server. Moreover we may to need just a few of the servers for the connections computation but a lot of servers for the actual work given by the clients. We will be calling these servers for connections computation as adaptation servers and the servers for the actual work as computation servers. So the components of our framework are communicating as shown in figure [2.3]. All clients and computation servers are communicating with adaptation servers. The
Figure 2.1: Simple cloud computing

Figure 2.2: Simple edge-cloud computing
computation servers have to communicate with adaptation servers thereby the adaptation servers could properly do load balancing and to know which servers are active.

The communication can be implemented many ways. For example we can use REST API, Sockets or RPC. We could use reliable or unreliable communication. For some applications could be better one way over another. However we have to choose only one. Because we need to have our framework easily usable, we have chosen the reliable communication with RPC. The reliable communication is much easier to work with. Unfortunately we cannot guarantee the latency anyway. The RPC is chosen because it is common way of communication which very well fits to the requirements of the cloud computing.

2.2 Running example

Through this thesis we will work with a face recognition example. Even so our framework is supposed to be universal, we might need to make decisions which not may be perfect for all use cases. In these cases we are choosing the variant which suits the best to the face recognition example. Of course we have also to look if there is some more universal solution which fits us.

The face recognition is used to identify faces in the images. It works in two phases. The first one just detects any faces at the image. The second phase recognize the specific people among the faces. The recognized faces have to be
trained at first. We provide some images with faces of certain people and label them. From these labelled images we train a recognizer and store it to a database.

The detection could be done at the phone. Unfortunately for the recognition we would need a copy of the database at every phone. The distribution of such application would be really complicated. Also the application would be quite big and the phones have memory limitations. So it is better to do the computation at the server instead of the phone. Another problem would be that the computation at the phone could be slow.

Our example have two ways of usage. The first one is to take a photo by the camera of the phone. This photo is then analyzed to find the faces. The second one uses the camera preview and immediately draws the locations of the faces at it. The first usage does not need any extra speed therefore we could use common cloud computing to achieve our goal. The second one on other hand needs to be as fast as possible. As we move with the phone, the preview is continuously changing. So the location of the face at the preview could be totally different from the recognized one if the response from the server came late.

2.3 RPC

We need a way to communicate with a cloud server. Typically the server implements some functionality and the client wants to use that functionality to get results. RPC (Remote Procedure Call) is a way of calling remote function as it was local [1]. The call of a single function is shown in figure 2.4. When we look at the classic RPC in more detail it works as follows.

1. Calling stub method as it was local. The stub is generated code containing remote functions.

2. The stub packs the parameters of the function into the message. It is called marshalling.

3. The message is sent to the server.

4. Server receives the message and passes it to the skeleton. The skeleton is also generated code as stub and it has similar functionality.

5. The skeleton unpacks the parameters from message. It is called unmarshalling.

6. The server function is called with the appropriate parameters.

7. The result of the call is packed by the skeleton to the message.

8. The message is sent back to the client.
9. The stub unpacks the result from the message.

10. Finally the result is returned from the originally called function.

Common usage is that the client and the server run at different machines but it is not rule. They can run just in different processes at the same machine. The main reason to do this would be for testing.

There exists many implementation of RPC. Some of them are language specific. Java’s Java Remote Method Invocation [2] or Python’s RPyc [3] are examples of such implementations. But there exist also universal RPC frameworks. Probably the most famous is CORBA [4].

Because the client and the server can run in different environments and sometimes even use different programming languages we need a way to define an interface in language and platform independent way. Very widespread way of defining the interface is usage of the interface description language (IDL) [5].

2.4  gRPC

*gRPC is a modern open source high performance RPC framework that can run in any environment.* [6]

It uses http/2 [7] protocol. This protocol works over TCP/IP. The main impact of this approach for us is a reliability of the communication.

Because gRPC is open source framework, it is possible to adjust it to specific needs of any developer. Even so there is such a benefit, not many users have to edit the source code of the gRPC to achieve their goals. There are already the main communication tools implemented.

The framework is platform independent. Thereby if it is properly extended with new features, it can still run anywhere. The framework is also implemented
in many programming languages. At this moment these are C++, Java, Python, Go, Ruby, C#, Node.js, Android Java, Objective-C, PHP and Dart. The important part is that the client and the server can be implemented in different languages. Unfortunately the framework has differences in implementations among languages. Thereby further in this theses some implementation details can be specific for client Android Java and server Python.

gRPC has four types of service methods:

- **Unary RPC.** The client sends a single request and receives a single response. This is the most similar to normal function call.

- **Server streaming RPC.** The client sends a single request and obtains a stream for reading a sequence of messages back.

- **Client streaming RPC.** The client sends a sequence of requests. After all requests are sent a single response is received.

- **Bidirectional streaming RPC.** Both the client and the server send a sequence of messages. The streams are absolutely independent therefore the messages can be read and written in any order on each side. Therefore the server can send a response after each request or it can wait until it receives all requests and then send all responses for example.

The unary call has to always return a response. Sometimes we might want to return nothing. This type of function would just change the state of the server in some way. Normally we would implement a void function which returns nothing. This is not possible in the gRPC. But we can return a message of an Empty type. This message is already defined in the gRPC and contains absolutely nothing. Even so this type of message already exists, it is not good practice to use it. It is better to define own message which has an empty content. The reason for this approach is the extendibility. The message defined in the gRPC cannot be extended. The message defined by us can be extended very easily. Even so the user thinks that he/she will not need to expand the function, he/she cannot think about all possible circumstances. Therefore it is the best to keep the possibility to adjust the function in order it could return some actual data.

The classic RPC is synchronous. gRPC also allow us to use asynchronous version of communication. In fact the synchronous communication is more restricted than the asynchronous. It is possible to use it only in combination with unary RPC. All streaming RPCs have to be asynchronous. This allows the user to speed up processing of requests but he/she also has to properly handle congestion control.
2.4.1 Protocol Buffers

The interface of communication is defined by Protocol Buffers [8]. Generally Protocol buffers are used as language-neutral, platform-neutral mechanism for serializing structured data. But it is also possible use it to define services and for each service supported methods.

Currently Protocol buffers of version 3 is available. This version is fully compatible with gRPC. Therefore in this theses the version 3 is described and used.

The basic element in Protocol buffers is a message. Message can contain basic types, other messages or maps. It consists of fields. Each field contains type, name and unique integer identification. Thanks to this concept each Protocol buffer can be extended if some rules are kept up. The main rule is to keep for the same field the same identification and type. If an old Protocol buffer is received by new client the missing fields are filled by default defined values. If a new Protocol buffer is received by old client the unknown fields are ignored.

The data are serialized to a binary format. This reduces necessary memory requirements. On other hand the serialized data are not human-readable. Mainly for this reason it can be better to use different formats such as XML or json in some cases. Also the Protocol buffers’s data are not self explaining. Thereby for serialization and deserialization it is necessary to have the definition of the concrete Protocol buffer.

2.4.2 Channel

gRPC has many components that allows us to change behavior of the communication in almost every thinkable way. These components provide us support for example for load balancing, authentication or health checking. Even so these features are very useful, common user needs to know only about three components. These are the channel, the stub, the skeleton and the server. The skeleton in the gRPC is called the servicer. The skeleton with the server are server-side components and the channel with the stub are client-side components.

The server-side of the communication is very simple. The user provides implementation of his RPC functions to the skeleton. The next step is starting the server component. The user provides to the server an implemented skeleton and other necessary data such as port at which the server will be running. These are all required steps for starting a server.

At the client-side the most important part is the channel. It provides a connection to the server and handles all RPC calls. The stub is used only for an easier usage of the channel. An instance of the stub is created with the given channel. The user then calls the RPC functions of the stub as these were implemented locally. Anyway it is possible to not use the stub. In this case the user
would have to manually start the connection, provide all parameters for starting the calls and then send the messages through these calls.

By setting the parameters of the channel we adjust the connection to the server. For example by setting the channel we can specify if to use authentication. The settings are applied to all RPC calls therefore it is necessary to set it only once.

The connection to the server is established only after it is needed. Therefore if the user created the channel and did not make any RPC call, the connection would not be created.

Even so the gRPC supports many programming languages, not all of them has absolutely similar functionality. For example in Java it is possible to shutdown the channel and wait until it is terminated but this is not possible to do in Python.

### 2.5 Load balancing

The load balancing is used to distribute the workload across different nodes. In this meaning the node can be a computer, a CPU core or a disk drive for an example. The reason for distribution of workload is to minimize the computation time and to avoid the single resource exhaustion. In our framework we have multiple servers and the clients are distributing the workload among the servers. The use of the edge-cloud computing can be considered as using the load balancing. The reasons can differ a bit but the techniques are remaining very similar.

The load balancing can be handled both on the client or on the server side. One approach is to use the proxy server. In this case all clients are connected to the proxy server. The proxy server then distributes the requests from the clients among the servers. The proxy server can implement some complex logic for the distribution. Thanks to the one central component which knows everything about the workload size, we could very well optimize its distribution. On other hand a single central component can very easily become a point of failure of the whole framework. It is also not very usable for too many requests. All the requests have to go through a component other than the client and the server. This increases the latency. One of the main reason why to use the edge-cloud computing is to decrease the latency. Therefore mainly for this reason is not this approach suitable to us. Even the gRPC dropped this approach for the mentioned reasons.

Another approach to the load balancing is to have all distribution implemented at the client side. The client has to gain the list of all possible servers. The list could be statically configured in the client or somehow received from the servers dynamically. The static configuration has a drawback that the framework cannot easily scale. We could use the random distribution or the Round-robin algorithm for example. These approaches are working without knowing the state of the

12
The client could also communicate with servers to gain additional data for optimized distribution of the workload. Thereby for good results would the client have to communicate a lot with its surroundings. Anyway the client would have to be really complex and big but we want to keep it small and portable.

Final approach is to have extra server for the load balancing. This extra server can communicate with the backend servers. Therefore it has information about the loading and the health of the servers. There can be implemented very complex logic because it will not slow down the actual computing between the client and the server. The client and the computation server are communicating directly. This extra server is sending a list of server addresses to the client. When something changes then the client receives a new list of the server addresses. The client can choose from the list any of the addresses. It is using some simple algorithms like Round-robin.

The gRPC allows us to choose from two approaches of the load balancing. These are all logic implemented in the client and the logic implemented in the extra server. The client chooses which one to use by the configuration of itself. The client is always having a list of servers. It is creating the subchannel to each of them. Then the workload is balanced between them.

There are more possibilities what will be balanced. We can balance the connections. The client chooses one server and then it makes connection to it. This connection does not change then. This is the simplest approach but it also cannot
react to the changes. Another approach is to balance per call. In this case we choose every call the destination server. This is used by the gRPC.

The important thing to mention is that the call in the gRPC does not mean sending a request. The call means sending the request only for the unary call. For the streaming calls is the call sending possibly multiple requests. Our clients are mobile and therefore it can be necessary to switch servers during the call to keep the latency low. For this reason we cannot use the load balancing of the gRPC. The gRPC load balancing would be useful only if the streaming calls are short. Unfortunately our framework is universal therefore we cannot guarantee it.

![Figure 2.7: Load balancing with the extra server](image-url)
3. Edge-cloud framework

3.1 Requirements

The most important requirement on our framework is easy usage. If it would be too complicated to use it then the user could implement the edge cloud communication itself. We also do not want to use it only in a single application. So the framework has to be universal. Another requirement is platform and language independence.

The natural way to achieve our requirements is to extend already existing framework that fits our needs. This framework is gRPC. The gRPC is a platform independent framework. So if we do not add anything platform specific then one of our primary requirements is already satisfied. The gRPC is implemented in eleven languages so the user has really big chance that he/she can pick his favorite programming language.

The gRPC is an open-source framework. This allows us to easily implement any extension we want to. We want to be as transparent as possible so the user can implement his application without knowing it will run in the edge cloud environment. For us this means that the implementation of our framework has to be hidden inside of the already existing gRPC components. This cannot be achieved absolutely because there is necessary some extra input for some optimizations of our framework and its correct behavior. These extra parts are only a small part of the whole framework. So if someone is familiar with the gRPC framework then he/she is also able to automatically use our framework without learning a lot of about edge cloud.

3.1.1 Network communication

The communication in the gRPC is reliable. The reliability is achieved by using the TCP. This perfectly fits to our requirement. Using the reliable communication is much easier for the user than using the unreliable version. Even so it easier to use it also has some disadvantages. The main disadvantage is that we cannot guarantee the latency in any way. This is caused by the fact that we cannot to throw away any messages or their parts. In our requirements has the easy usage bigger priority than the speed. For this reason we are fine to keep the reliable communication.

We could also implement the unreliable version of our framework thanks to the open-source nature of the gRPC. It is not done because the benefits of it are not so big considering the formed complications. The users of the gRPC are already asking for the unreliable approach. They want to use it mainly for game
playing. The authors of the gRPC are not against it so we will probably see it soon. Our framework will have then possibilities to be extended by knew features that could dramatically decrease the latency.

There also exists the congestion problem. For the user it would be nice if he/she would not have to handle it. Unfortunately this cannot be universally achieved without throwing away some messages. As we said that would lead to more complications. So it is on the user to send only the data that can be handled by the servers or the clients respectively.

3.2 Architecture

We have three main running components in our framework. These are client, server and adaptation server. Anyone who uses the classic cloud computing, has to work with the client and server components. These components in edge cloud computing should look very similar to the ones in the common cloud computing.

The schema of the architecture can be seen at figure 3.1. The blue parts are implemented as the part of this theses. The orange ones contains the logic for switching between servers. This logic can be specific for the network that our framework runs on. Finally the green parts are application specific and these are implemented by the user of our framework. If the user used only the common cloud computing, he/she would have to implement the same parts in almost the same way.

Client creates requests for some computation. These requests are sent to the server. The server does the cloud computation and returns a result to the client. The client is also connected to the adaptation server. The main purpose of the adaptation server is to decide to which server should be the client connected. For correct decisions have to exist a connection between server and adaptation server. Through this connection the adaptation server gets the information about states of the servers.

3.2.1 Central and edge server

The servers that does the computation can be divided into two types. These types are central and edge servers. The central servers are the ones used by the common cloud computing. The edge servers are the ones used by the edge cloud computing. Our framework is targeting the edge cloud even so we have to have also the central servers.

The final application that will use our framework can have multiple functions. Some of the functions would have to work quickly or there could be planned a lot of calls of these functions. In these cases we want to use the benefits of the edge
There could also exist some functions which would not need to use the edge cloud. We could have a single central server that would have more power such as CPU or GPU for executing some complex tasks. It is better to use the single central server if these tasks are not so common. It is possible to have the necessary power at the edge servers but it would be expensive because there will be more edge cloud servers than the central servers.

There are another reasons to use the central servers. If we do not need the speed gained from the edge cloud, we can have just the central servers. It would lead to much simpler network. Also in some cases it can be better to have the data at one place.

For an example we have an application which is doing a lot of computation and needs a lot of speed. This computation should be done at the edge servers. This application also have some extra logging. Because we want to have all of the logs accessible at one place we should use the central server. For logs there is no need for quick response that would have to be shown to the user. So we do not care for the performance so much.

### 3.2.2 Client

The complete architecture of our framework can be seen at figure [3.2]. The biggest part of the common client is an application part. It contains UI and data generation for the computation server. This part is completely dependent on a specification of the application.
Figure 3.2: Detail architecture of the edge-cloud framework
Another part specific to a particular application is the stub. One approach to implementation of the client for the edge cloud is a generation of a specific stub. This specific stub would take care of all the edge computing stuff. Mainly it means switching between the edge cloud servers. This would be totally transparent to the user. Only difference between an edge and a classic cloud computing would be usage of the different stub. On other hand this approach would be not really natural in gRPC environment. Somehow we would have to inform our stub about an edge cloud specific data. We could create a specific channel with these data or insert it straight to the stub. In these cases the user of our framework would have to know a lot of about an inner functioning of our framework. The absolute minimum would be a creation of the specific stub instead of the classic gRPC stub. For this reasons we use an unchanged stub completely generated by gRPC.

The channel represents the connection to the server. It has interface containing the necessary basic functionality for connecting to the server but there exists also some extensions with extra features. These extensions allow for example to terminate the connection to the server. We designed an extension of the channel which allows us to work in edge cloud environment. This extension is also implemented in Java for Android. We are keeping the basic interface so the common gRPC components are able to work with it correctly. This also includes the gRPC stub that we do not want to change.

The edge cloud channel is a wrapper around three components. These are the switcher, the informator and the actual channel. The actual channel represents a connection to the server that is currently in use. When a function of the edge cloud channel is called, it is only passed to the actual channel which does all the work. The important thing to keep in mind is that the actual channel can be exchanged for a different one in any time. This happens when we want to connect to another server. For this reason all functions of the edge cloud channel has to be synchronized. So when making changes to this part of the channel, we have to be extra careful to not create a deadlock.

The decision which server is currently the best for the client is a very complex task. It can depend on many factors and it can use many metrics. In our implementation we are deciding only based on a distance. Closer the client is to the server the better. The informator gathers all necessary data for decisions and provides them to the switcher. These data can be anything. Besides location it could contain the expected amount of data that would be sent to the server for example. The switching between servers has some overhead. So it would be useful to not switch between servers if we do not have enough data for the new server. Not all application could use such functionality because we do not have to know the amount of the data that would the user generate. For this reason is the informator a component that has to be known to the user of our framework.
He has to choose the informator that suits the best to his needs.

Final edge cloud channel subcomponent is the switcher. The switcher communicates with the adaptation server and changes the actual channels. The switcher runs in a separate thread and it is not visible outside of the edge cloud channel. So the user of our framework does not have to know anything about it.

At figure 3.2 we can see one more component and it is called ClientCall. The ClientCall represents the single call of the function to the server. The common user of gRPC does not have to know anything about it. It is used internally by the stub. Our framework also isolates the user from this component to keep transparency. For this reason we have to keep the interface and the main functionality of the original one when we are creating some extension of it. Each of the unary calls creates its own ClientCall. It corresponds to the common RPC. The ClientCall is also created for a streaming call. It is created for each call but not for each request. For the unary calls is created the common gRPC ClientCall. There is not necessary any special handling. We have just a single request and a single response. So there is no time for switching between servers or mixing the responses together.

The complication comes when using the streaming calls. The edge cloud channel creates a gRPC channel pointing to the actual computing server. That channel then creates the ClientCall which handles all requests and responses. We could just use the gRPC ClientCall if there is no switching between servers. But we are expecting the server switching so we have a special edge cloud ClientCall which contains the actual call that can be switched during the computation.

The edge cloud channel and ClientCall are both used as wrappers to the actual channels and ClientCalls. The ClientCall have also one more functionality. It is to keep the order of the responses. The communication we use is reliable. So we have to keep the order of responses the same as they go from the servers. This is completely handled by the gRPC when there is no switching between servers.

The streaming call can be used for two reasons which can be combined. The first reason is to keep the state of the communication so we do not have to initialize the communication over and over again. The second reason is to gain speed. The streaming communication is asynchronous so we do not have to wait for the response and we can keep sending the requests so the server and the network are fully used. The easiest implementation of the first reason is to wait on all responses after each request. In this case we would have already all messages correctly sorted. The complication comes with the second reason where the requests are continuously sent to the server and the unknown number of the responses will be returned. The situation can be described in the following steps:

1. Some requests are sent to one server.
2. The client switches from one server to another one.
3. Some requests are sent to the second server.

4. The client receives some responses from the second server before the connection the first one is finished.

We could just deliver the responses if no more responses are sent from the first server. This would just mean that the connection was not properly closed yet. But each request can lead to zero or to infinite number of responses. Even so the single response after each request is the most common case, we cannot guarantee it. For this reason we cannot determine that all responses were delivered until the communication with the first server ends. We need to keep the responses from the second server in the buffer until all responses from the first server are delivered so we could keep the order. There could also be a situation where we receive a request from the third server but the communication with the first one is still active. So instead of having one buffer, we have to have a buffer for each server.

The user of our framework has to keep in mind the problematic switching between servers. The direct consequence is possible slowdown of the computation when we have to wait on some results although we already have the newer ones. This also affects the implementation of the target application at server side. When the client stops sending the requests then the server has to correctly end his computation. It cannot stuck or throw an exception. The first case would lead to deadlock and the second case means that something went wrong and we should end the computation. We cannot automatically recover from a mistake in a server computation because that would lead to loss of the data which is not possible in reliable communication. So the user is notified that something went wrong and is totally on him to choose how to respond. He could stop processing any further requests or he/she could just restart the channel and continue in work.

3.2.3 Adaptation server

The adaptation server decides where should clients and servers be connected. It has three main components. These are the servicer, the model and the decider.

The model stores all the data about the state of our framework. This means for example how are the servers and the clients connected. The servicer takes care of all communication with the computation servers, the clients and all other adaptation servers. The gathered data are stored in the model.

The decider contains all logic necessary for correct decision about connections. The servicer asks directly the decider how should be the state of our framework changed. But it provides no data to the decider. All the input data for the decider have to be stored in the model. Only exception to this rule are environment specific data. This can be for example the topology of the network. Our example
implementation of environment provides only location of the servers and the adaptation servers.

The adaptation server do not initialize any communication with other components and it also do not initialize any computation. Only exception to this rule is communication between the adaptation servers where they are exchanging data about themselves. In this situation one adaptation server initialize the communication and starts to send the data about itself. The second one is then sending the data about itself as response. So the adaptation server has no way to send some data on its own. Every sent data are responses to the client or to the server.

The communication with the client is done in the following steps in the specific order.

1. The adaptation server receives the data from the client and updates its model.

2. The decider selects the most appropriate adaptation server for the given client. If this adaptation server is different from the current one then the address of it is sent to the client and the communication is terminated.

3. The decider selects the most appropriate computation server for the given client. If this computation server is different from the currently used one then the address of it is sent to the client.

4. The decider chooses the functions for which it needs to discover the time of the computation. The request for time computation is then sent to the client. The model is updated with the response.

5. The decider chooses how often should the client send its data to the adaptation server. If the value is different from the old one then it is sent to the client.

Only action that is done everytime is the update of the model. If the decider decides that the current state is the best possible then no data are sent to the client. So in case the count of all nodes in the system is stable and the clients are not moving too much then the traffic between the clients and the adaptation servers would be minimal.

The discovery of the time necessary to compute some function can take a long time. During this time we can receive updated data from the client because the communication can be asynchronous. In this case we update the model but we also wait until the time is discovered. The client is kept connected to the same adaptation server and the same computation server in the meantime.

The client is periodically sending the data about itself to adaptation server. How often the data should be sent is decided by the adaptation server although
it is absolutely the client variable. The period could be a static number but in that case the client could sent the data too often or not sufficiently. In the first case the client and the adaptation server would be unnecessarily busy. In the second one there could be better server for the client and it would not know. For this reason is the period a dynamic variable. Only the adaptation servers have knowledge about the state of the network so it is the only component that can choose the correct period of the time.

The example of need of setting the period dynamically is shown at figure 3.3. We have two servers S1 and S2. They separate the place on two locations where they can serve to the client the best. The client is moving from the point A to point B. When the client is close to point A then the period can be long. When it comes close to the border of the location of the two servers then the period has to be very short. After the client crosses the border we want to quickly change the servers. But we do not know if the client crosses the border. It could change direction back and so it would be the best to keep the connection the server S1. Even so it keeps the direction we cannot determine when will be the border crossed. The client can change its speed any time. This is only simple example with one client. When we have multiple clients, we might need to reduce the period to keep more actual data.

3.2.4 Time computation

The main benefit of the edge cloud computing is the gain of speed. From the look of the client the computation is faster. The overall speed is dependent on too many factors. It could be the physical distance, the count of the nodes on the path or the capabilities of the server. One server can be faster even so the parameters
of the second one look better. It can be caused by some missed variable or just
genesis of some unpredictable condition. The overall speed also changes in time.
It is caused for example by instability of the network and the variable count of
the clients.

It is not possible to absolutely guarantee the time necessary for computation.
But with the knowledge of communication times we are able to adjust the con-
nections to improve it. For this reason our framework allows to compute the time
necessary for completion of the tasks.

The user of our framework already had to define the interface for a client-server
communication. So we already know what type of data will be sent. There also
has to exist an implementation of the server computation. Without providing
any additional information we know the type of data and what to do with it.
Only other thing we need is the exact data. The request could contain just array
of bytes for example. But we do not know the format of that array. It could be
an image or a content of some file for example. But knowing the format is not
enough. There is also important the size of the data. The bigger the data are,
the longer time is necessary to send it and process it generally. So for correct
computation of the time it is necessary to have the data similar to the commonly
used ones.

The computation of the overall time is not enough. The computation of the
task will take similar time for equally configured servers. Mostly we are able
only to shorten the time of the travel because the time of the computation is
completely depended on the user implementation. So we need to compute the
time of the travel and the time for the computation. This information allow us
to update the state of our framework adequately. Even so these information are
important only to the adaptation server, we have to adjust the adaptation server,
the client and also the computation server.

The computation of the time is done in these steps:

1. The adaptation server sends a request to the client containing for which
   function it needs the time computation.

2. The client packs the data provided by the user and send it to the server.
   At the same time it also remembers the actual time.

3. The server remembers the actual time and then sends the received data to
   the appropriate function.

4. After return from the computed function, the server calculates the time
   from actual time and the remembered one.

5. The result of the function and the computation time are sent back to the
   client.
6. The client computes the overall time from the actual time and the remembered one.

7. Both the time received from the server and the computed one are sent back to the adaptation server.

The important part to notice is that the result of the function is sent back to the client even so it is not used. It is done to compute the travel time correctly. The returned data will be commonly small and will not make so big difference but in some cases these data could be enormous. For the most appropriate result we have to simulate the common computation how best we can.

The server can implement one function for the time computation or it can have a generated function for each of the original ones. We do not want to make any name changes to the functions. Any such change could lead to a collision of names. Especially if the user extends the server. If we choose to generate the time computation functions, we would have to have the separate instance of the server to avoid the collisions. This new instance of the server would have to have absolutely the same settings as the original one. Both servers would also had to be connected so the new instance could call the functions of the old one. If there is no such connection, there would have to be twice implemented the server. This approach is too complicated for the user. So there is implemented only one function for the time computation. It is called "computeTime". The user of our framework cannot use this function name for his services.

The parameters for this one function have to contain the description of the function to call and its parameters. The description of the function consists of the name and the type. The type is one of unary, server-streaming, client-streaming or bidirectional-streaming. The complication comes with the type of the parameters to the function. It can be absolutely anything. Fortunately gRPC has a special type called Any which can contain any data.

The user’s server has to be extended by our implementation of the time computation function. This function just unpacks the parameters and the name of the destination function. Then it calls the destination function and computes the time. When the destination function has the unary type or the server-streaming type, we call it just once. For client-streaming and bidirectional-streaming there can be multiple sets of parameters. In that case we have multiple calls of the destination function. The initialization of the server computation might be done only the first time the function is called because these types of functions are keeping the state. Such initialization can take some time but it is only done when the function is called on the server for the first time. We need to remove the impact of the initialization on the computed time if the initialization takes more time. It is done by multiple calls of the function and computation of the average time for one call.
The multiple calls of the server function cannot be done automatically. Each of the calls puts a pressure on the computation server. So the computation server could exhaust all of its resources on time computation and the computation with the actual data could be slowed down. The user even does not have to have the initialization part or it can be so quick that it can be ignored. Also the size of the request has to be increased properly to simulate the multiple calls. So the client has to already send all the requests and the server cannot duplicate them. Even so the server has to call some function multiple times with different parameters, it is the client that has defined the number of the requests. So the server computes the whole time and this time is send back to the client. The client then computes the average time. The user has to choose the proper number of the requests so the computed time would be accurate but the network and all other components are not overwhelmed.

The entire process of time computation is shown at figure 3.4. The users input is a list of parameters for functions with correct types. This list is then automatically packed into the list of parameters of Any type. This all what is happening to the request at client side. Next at server side the parameters are unpacked to the original type. When the parameters are unpacked, the function is called with each set of the parameters. We get a list of results. these results are packed to the list of Any type. Finally it is sent back to the client. Notice that for unary calls is N and M equal to one.

3.3 Metamodel

A model is description of something. In our case it is a state of nodes and a connection of them. A metamodel describes the model. The basic usage of the metamodel is to describe what components exist in our framework. It is useful mainly for architectural reasons.

There exist many ways to describe the metamodel. One of the popular ways is to use EMF (Eclipse Modeling Framework). The EMF uses UML diagrams to describe the metamodel. From this metamodel can be generated a code for the creation and editing of the models. This is called model@runtime. The EMF is also able to serialize or deserialize the model to or from a string representation. This feature is necessary when we need to exchange the model or its parts between nodes.

Another option to describe the metamodel is to use the Protocol Buffers. The Protocol Buffers have many advantages for us. We do not have to take care about platform specific details because they are already platform independent. We are also free to choose any programming language. Generally the usage of the Protocol Buffers for the creation of the metamodel gives no new restrictions.
Figure 3.4: Workflow of the time computation
because we are already using it in the gRPC. The serialization and deserialization of the data is also supported by it. Only difference is that the data are serialized to a binary format instead of the text one.

The benefit of the EMF over the Protocol Buffers is allowing of the back references. For example consider we have components A and B. In the EMF A can point to B and B can point to A. In the Protocol Buffers we can point in only one direction. However this disadvantage can be easily removed. We can to give IDs to the components of the model. Then it is no problem to manually search for the component with the specified ID. However it is good to keep it in mind therefore the lookup would not be slow for the big models.

The component that uses the model directly, is the adaptation server. It needs to know many data about other components. It also needs a way to store these data and to exchange them with other adaptation servers.

Our metamodel is containing two main parts the application requirements and the network part as shown at figure 3.5. The gray parts are not implemented and are specific to the environment that our framework will be running in.
3.3.1 Application requirements

Our framework can be used by more than just one application. Additionally each application will typically consists of multiple services and calls. Each of the services can have different requirements. One could have to be quick another could have to use special hardware for example. So we need a way for the user to easily describe his application requirements.

The description of the application is done through the yaml file. This file has to be called application_requirements.yaml and be placed next to the adaptation server. Any time the adaptation server is started, this description file is automatically loaded and stored in the model. The file consists of the list of the applications. Each application has a name, requirements and services. The name has to be fully qualified name as defined by gRPC. The requirements can be omitted if we do not need any. Each service also has a fully qualified name and the requirements. The requirements for the service have the same syntax as the requirements for the the application. These requirements can contain anything relevant for settings of the framework. For example it could be hardware specification of servers, performance or information if we want to use central or edge servers. If some service does not have specified some requirement then the requirement of the application is used. If both the service and application do not have some requirement then it is omitted when the adaptation server makes decisions.

3.3.2 Network

Each adaptation server can be connected to multiple edge servers, central servers and the clients. Each of these components has to be connected exactly to one adaptation server. The adaptation server is responsible to keep updated the state only of the components it is directly connected to. The adaptation servers are connected each to all others. This we they are exchanging their parts of the model with each other. So every adaptation server is keeping its local copy of the model of the whole network.

The edge servers and central servers have no differences. We need to separate them only for correct decisions about connections. Each of the servers has an address which is used as an ID. It also have a location and other specific data such as CPU and GPU usage. These data are dependent on the provided data of the cloud. Another information stored in the model about the servers is the count of active clients. Many clients can be connected to the server but not all of them have to do any computation.

The client has the address that is also used as the ID and the location of it. It also contains the computed time about functions that are available to
the client. If the client does not support the time computation for some of the functions then these are not listed in the model. For each function we need to store its name, time of the whole computation including the time of the data travel, only computation time on the server and the timestamp when these data were computed. The time computation can take some time therefore we do not want to do it too often. Also there could be problems in the network or the server can became too busy therefore the computed time can differ from reality and we need to recompute it.

Our metamodel consists only from the essentials parts. To gain the best result from our framework we would have to expand our metamodel about all the data we are able to get from cloud servers and the environment. One of such important improvements would be to store the network usage.

3.4 Communication data

The common cloud computing has only one server and the clients that need to communicate with the server. Our edge cloud computing has more components. All these components need to communicate with each other. There is necessary to specify the format of the data that will be sent. Especially we have to focus on which data are mandatory and which can be extended to achieve better quality of our framework.

3.4.1 Adaptation server - client

The communication between the adaptation server and the client is always initialized by the client. There is a single bidirectional RPC call. The client is sending a client information and receives a client action.

The ClientInfo messages can contain two different types of data. These two types of data cannot be mixed in one message. The one type is information about the client. This can be considered as a request and the action that should be done by the client is the response. These information contain the location of the client and a name of the service it is using. The location field is mandatory only in the current implementation of the adaptation server. There could be implemented a better way of getting the location of the client at the adaptation server or the decision logic can be implemented in a way that it would not need to know the location of the client. These data can be extended in any way thereby the adaptation server has all necessary data for decisions. After an extension of the data, the client has to have implemented the EdgeCloudInformator class which provides the data. The name of the used service has to be always present. The user application could contain multiple services with different requirements.
The adaptation server is then able to choose the correct computation server by knowing the current service.

The other type of data send by the client are the time computation results. This can be considered as the response to the adaptation server because the adaptation server choose what functions needs the time computation and when it has to be done. The time computation result consists from the time of the whole computation including the time in the network and the time of the computation without the time in the network. We are storing these results in map where the key is the name of the function and the value is the result of the computation for that function. The result contains also the timestamp of the computation but it is added by the adaptation server when it stores the results to its model.

We have four actions that can be received by the client from the adaptation server. After the data about the client are sent to the adaptation server, it can send back some actions. If there is nothing to change, it will send no action. If there has to be changed multiple things then multiple actions are sent.

First received action is the address of the more appropriate adaptation server to connect. Then there is the address of the most appropriate computation server. Next is the time how often should the client send the information about itself. The last action is request for time computation of the functions.

### 3.4.2 Computation server - adaptation server

The communication between the adaptation server and the computation server is always initialized by the computation server. The computation server is sending information about itself and receives the addresses of all adaptation servers.

The information about the computation server consists of its address, flag if it is central server and the actual data. The address and the flag about central server are sent only the first time it connects to the new adaptation server. The actual data can be extended in any way thereby the adaptation server would have all the necessary data it needs for correct decisions. Currently it contains the number of active clients connected to the computation server. There could be many clients connected to the server but only a few of them can do some actual work. So this information is good to know how busy the servers are. The drawback is that this information has to be provided by the user. So this approach gives us important data but removes part of the transparency of our framework.

The computation server receives the ordered list addresses of the adaptation servers. This list is ordered from the most appropriate one to the worst one. this list is then used for connection to other adaptation server if the current one has some failure or the current one is not the most appropriate one.
3.4.3 Computation server - client

The computation server and the client communicates also in the common cloud computing. This type of communication is completely implemented by the user. Our framework inserts only support for time computation of the functions. The client sends the request containing the description of the function and its parameters. As a response it receives the result of the function and the time necessary for computation. The functions are implemented by the user of our framework therefore we do not know what will be the type of the parameters of the functions. Even we cannot put any restriction to that type. Fortunately the gRPC provides us the Any type which can contain any message type. It is totally created at runtime.

Android has to keep its applications small. So the libraries used by applications should be also as small as possible. For this reason Android uses an adjusted version of the gRPC called gRPC lite. This version does not contain all features. One of the missing features is the Any type. Any type consists of two parts. These are data descriptor and the data. The data are just byte array serialized by protocol buffers. The messages also contains the description when it is generated by the full version of gRPC. This description is then used to fill the descriptor of the Any type. This descriptor is just a string containing the name of the message. Because the description of the messages is missing in the gRPC lite, the user has to manually provide the names of the messages used as parameters of the server functions. With these information we are able to implement the Any type functionality ourself.

3.5 Data sharing

When we use a single cloud server then we have all data at one place and everything is keeping its state. In the edge cloud we have to share data between the servers and also to keep the state of the computation.

The unary RPC call does not have any state. But the state is kept by all the streaming RPC calls. The special case is the server-streaming RPC call. The client makes a single call but it receives a stream of responses. This type of call is not very proper for the edge cloud environment. The server could infinitely send the data but for client it could be better to connect to the different server. For this reason the server-streaming RPC call should be used only if it is guaranteed it will end quickly or use it on central servers.

The client-streaming and the bidirectional streaming RPC calls are both keeping the state of computation and can be interrupted by the client. The workflow is as follows.
1. The client is sending requests to the server.

2. The client receives notification that it should change the server.

3. The client ends the communication with actual server.

4. The client connects to the new server.

After the client connects to the new server we are losing the state of the computation. The typical streaming call have two phases, the initialization and sending the requests. The initialization phase is used when we have the same data that would be repeatedly sent but these can be stored in the state of communication. Typical example is to send some ID in initialization message and then to load some data from database dependent on that ID. The access to the database can be slow and it is better to keep the data in the memory rather than to load them after each request. In the second phase we are sending the request and expecting to get the results of the cloud functions. This second phase also can store its state. If it does we would have to transfer this state to the new server. Unfortunately this state is absolutely specific to the application. That means that the user would have to edit implementation of his functions to allow receiving the state. The client then would have to wait until it receives the state and send it to the new one. The switching between the servers would be slow for this reason and we could not ensure the latency in any way. Everything is also complicated by the fact that the streaming communication is asynchronous. So we decided not to keep the state of the computation between the requests. If the user needs to keep the state in some part of the computation he/she has to send a single request for that part. This ensures that everything will be computed correctly because we cannot interrupt the computation in the middle of the request.

The initialization phase is something that cannot be omitted. The user could send the initialization data in each request but it would lead to unnecessarily large request and thereby greater latency. So the initializing message can be defined at the client by the user for each of the functions. This special message is sent as first immediately after connecting to the new server.

We are already able to keep the state of the communication. Another thing is also to keep the context of the computation. By context we mean any data that are available only to the current server. Typically it is the content of some database. Unfortunately this is absolutely specific the concrete application and thereby it cannot be implemented in our framework. When user designs his application he/she needs to keep it in mind. But the distribution of the data is already implemented in many frameworks. For example the mongoDB\[10\] has a functionality called sharding.
3.6 Recovery

We cannot depend on stability of the network. Any component can be terminated anytime or just the connection between the components can be broken. Such events cannot shut down our framework. So most of our components are able to recover from failure of another ones. We do not have any central component which we could ask what to do. This complicates the situation because every component has to handle recovery itself and cannot to rely on any other. On other hand the unavailability of one component will not mean absolute failure of the framework.

Each client is anytime connected to exactly one adaptation server and one computation server. The termination of the client does not have to be specially handled. The adaptation server just removes the client from its model and the computation server stops to receive the request and thereby stops the computation. This is not anyway different from the regular shutdown of the client from the look of the servers.

The termination of servers has to be handled from the look of the client. The termination of the adaptation server is not a big problem. The client just cannot receive addresses of the servers to which it would be the best to connect. This means that the client is stuck with just one computation server. In this case the latency will be not optimal but the client is still available to do its computation.

The client handles the termination of the computation server dependent on the time of the failure. If there are no calls to the server then nothing will happen. The situation where some calls are connected to the server has to be handled by the user. gRPC uses the reliable communication and so we. So when the server is terminated in the middle of the computation then the client cannot receive the response with the result. The standard gRPC behavior in this situation is to throw an exception. We want to be as transparent as possible therefore we keep this behavior. This way is the user notified that some of his data are lost. It is totally on the user if he/she chooses to restart the channel and continue with computation or he/she needs to handle these situations a specific way. It is important to notice that the adaptation server is also informed about the computation server termination. So it will not offer this server anymore and the clients will receive an address of a new server.

The computation servers serve to the multiple clients and each of them is connected exactly to one adaptation server. The computation server periodically receives from the adaptation server the ordered list of the addresses of the adaptation servers. The list is ordered from the most appropriate adaptation server to the worst. This way has the server the list of all the adaptation servers. When a connection to the adaptation server fails then the server tries to connect to others in the list until it succeeds. If no of the adaptation servers is available then the
server tries the connections from the beginning of the list again. When a server is not connected to any adaptation server then no new clients can connect to it. But it can still work with already connected clients therefore we do not want to shut it down.

The adaptation server is connected to some computation servers and to all other adaptation servers. The failure of the computation server just means to remove it from the model. The adaptation server removes the terminated adaptation server from the model. When an adaptation server restarts then it is responsible for connections to all other adaptation servers.
4. Evaluation and related work

4.1 Visualization

We need to have a way to test our framework. The testing of our framework
mainly means that the components are correctly connected to each other. There
is a problem that we do not have any central component which would know about
the state of the whole system. Moreover any component can shutdown anytime.

We made a simple python program which gathers the data from the com-
ponents in the system and shows the visual representation of the system. All
adaptation servers have to run on the same computer as the visualization. This
is not a problem because it has only a testing purpose. We commonly just want
to check that any metric we use for the decisions for connections is working fine.

The running visualization can be seen in figure 4.2. There are two online adap-
tation servers, two online computation servers and a single client. One adaptation
and one computation server are offline. The online adaptation servers are marked
with yellow color, the online computation servers are marked with green color and
the client is marked with blue color.

The adaptation servers have the necessary information for the visualization.
We do not want to make any changes to the adaptation servers just because of
the testing. For this reason is the visualization acting as the proxy server. The
visualization starts a proxy server for each adaptation server. It is important to
notice that not all communication has to use the proxy server. The adaptation
servers can communicate directly because we do not get any additional data from
their communication.

The visualization needs to know which adaptation servers are online and which
are offline. From computing servers it needs to know which are online and which
are offline and additionally to which adaptation server it is connected. The last
thing is to know the location of the clients and the adaptation and computing
servers it is connected to.

The state of the adaptation servers can be obtained directly from the gRPC. We will create a channel to each of the adaptation servers. This channel is then used to resend the messages between the client and the adaptation server. gRPC provides us the state of the connection and also notify us when it changes. The notification does not happen immediately so the visualization can show a bad state for a few seconds.

Each computing server is connected to some adaptation server. If it is not connected then it is shutdown or the network failed. In both cases is the computation server unavailable so we can mark it as offline. Otherwise we show it as online with correct adaptation server, it is connected to.

The location and the connections of the client does not have to be totally accurate. In our demo implementation is the location determined by the client and then it is sent to the adaptation server. In real world it does not have to be the best solution so the visualization would have to be extended to allow get the location different way. The address of the best adaptation server for the client has to go through the visualization so this is an information we also have. From the communication of the adaptation server with the client we have to get the address of the computing server to which the client should connect. The computing server and the client then communicates directly. So the visualization does not show where is the client connected but where it should be connected. If there is no bug at the client then these two connections are the same.

4.2 Face recognition

We have implemented two exemplary applications which are using our framework. Both of them are for face recognition. The application sends some photos to the server and the server sends back the locations of the faces in the photos also with the names of the people. The computing server uses openCV for the face recognition. The server part is the same for both applications. Only difference is in the start of the server because for one of the applications is the server marked as central.

One of the applications is labelled as ”real-time”. This application uses the camera of the phone and sends the photos. Additionally it can also sends the frames of the preview so we can immediately see the faces of the people as we are moving with the phone. The second application is labelled ”disk” and it sends the photos located in the phone. This application does not need any extra speed so it can use the central server.

The client side of the applications have also some similarities. These are the menu and the settings screens. The menu has only two actions. These will
send us to the settings screen or to the main screen for face recognition which is dependent on the application.

The settings page allows us to choose the host and the port of the adaptation server and the id which is used by computation server to identify the client in the database. The real-time application allows us also to set the parameters for the camera.

The important note is that these implementations are just used as examples how our framework can be used. Some of the constructions are not recommended to use by Android.

4.2.1 Real-time

Android does not allow to access the network from the main thread. So the creation of the photos and the sending to the server has to be totally separated. For the ”real-time” application we have created an Android service which handles all work with our framework.

The GUI consists of two screens working screens. The first screen shows us the preview of the camera. There we can switch between preview and photo analysis. After we have chosen what to analyze then we can click the button to start. The second screen is used only if we have chosen the photo analysis. At this screen is shown the taken photo and after short period of time the found faces are marked. The analysis of the photo can take some time because the photos are commonly big.

For the preview we use the bidirectional streaming because we need to send the frames repeatedly. The photo analysis uses a simple unary call. This is switched in the Android service. After the service is started, we create a new thread. We load all necessary data from the settings. There is immediately initialized the communication with the adaptation server. So when the first images for face recognitions are sent to the service, we can send them to the computation server without any delays. When the client connects to the adaptation server for the first time, it has to get the address of the optimal adaptation and computation server. Without these information we cannot connect to the computation server and do any work. It is also important to keep in mind that these data can change any time. So make a connections and before we have data to send to the computation server, the client moves and we have to change the connection to the different computation server.

At figure 4.2 we can see the screenshot of our application. In the top right corner wee can see the time how long it took to send the frame, analyze it and receive the result. Then there is the marked face with a label who is at the image. There is also the number which indicates how well it matched the face. The lower number means better match.
Figure 4.2: Recognized face
If there is any image to analyze before we got the results from the previous one, we throw it away. This way the server does not get overloaded. The thrown away images would be just frames from the camera preview anyway. When we use this functionality, we want to have the most actual data shown and the workflow to be continuous so this is not problem for us.

The service is destroyed after it is not used anymore. At this time it closes the connection to the adaptation server.

Using the Android service is one of the ways to implement the network communication at Android phone. For the streaming types of communication it is probably the best one.

4.2.2 Disk

The second application has only one working screen. At this screen we can choose the images to analyze and then start the recognition. We did not implemented our own file selection. For this reason it is dependent on other Android applications if it is possible to choose multiple images at once. It is really dependent on applications and not on the phone. So when we have multiple applications which allow us to pick files then we might be able to get this functionality from one but not from others.

The chosen images are shown in the list. After we click the button to start the recognition, we send these images to the computation server and we are waiting on results. When the faces in the image are recognized then it is shown at the image in the list.

We are using the AsyncTask class to implement the network communication. It is the simplest way to start a task in different thread and to get notifications from it. Unfortunately it is not recommended to use it for long running operations. This is reason why it should not be used in this use case. We are using it just as an example.

The AsyncTask is not really appropriate even for the unary calls. We can make a single unary call but in the background we have a connection to the adaptation server. The repeated creation and destruction of this connection would remove the benefits of the edge-cloud. Of course we could have this connection in a separate thread and then make the calls to the adaptation server from the AsyncTask. But it would be complicated and our framework should be easy to use. So this approach is not recommended.
4.3 Related work

There exist a few frameworks for developing applications in the edge-cloud. First we look at the ESF [11]. This framework is platform and programming language independent. The IoT devices are connecting to a gateway instead of connecting to the cloud directly. This framework is not focusing on the mobile devices. This framework is good when working with IoT but could have problems with dynamic migrating of the tasks.

Another example of the framework for the edge-cloud is OpenStack [12]. OpenStack is platform for cloud computing. Actually it is getting a support for the edge-cloud. OpenStack is widely spread and that is reason why it is so interesting. In the past OpenStack supported only Python. Nowadays it supports more programming languages so we can say it is programming language independent. Because it is going to work on its own platform, it can get better performance. On other hand we are limited at the platform.

Overall there exist many frameworks which would be usable in the edge-cloud. A lot of them are getting a really good performance. Unfortunately when the framework has high performance, it also has complicated usage or other limitations like it is platform or language specific. The most of the frameworks are not universal in some way.

Even so the edge-cloud is becoming more known nowadays and in the future it can easily become necessity for the IoT, we do not have many frameworks capable of work in the edge-cloud environment and focusing on the mobile devices. So this framework can help anybody who wants to easily get the benefits of the edge-cloud and he/she does not want to implement everything himself/herself. This framework also allows to use reliable communication which is not typical in systems which require a low latency. On other hand it has easier usage therefore generally the users who do not need to lower the latency critically, might still want to use this framework.
5. Conclusion

The goal of this thesis was to analyze problems caused by the edge-cloud computing and to propose a suitable solution. We managed to achieve our goal. We analyzed many problems caused by the edge-cloud. These problems were solved in an implementation of our framework. The usage of the framework is nearly the same as the usage of the gRPC. So it is really simple and anybody who knows the gRPC can immediately use our framework without any extra studying. Some extra knowledge is only necessary when we are wanting to get better latency and other benefits of the edge-cloud.

Our framework is stable and robust. The whole framework is distributed and we have no single point of failure. So when one of the components fails we can reconnect the components to another one. Only problem is that we are using the reliable communication therefore the user get notified that something is wrong and he/she has to react correctly to this situation. This makes a bit complication to the framework but user is not secretly losing his/her data.

Our framework is not using any platform or language specific constructs. Even so our framework is currently implemented in Python and Java, we can implemented it in any language supported by gRPC. This is related to the fact that our framework is really universal. Any application using the edge-cloud can work with our framework. In particular applications which need to guarantee the latency we might to look after another framework or to adjust this one. The whole framework was design especially with focus on easy usage. We have also implemented two applications using our framework. These applications are using the edge-cloud in several ways. Therefore we proved that our framework is enough universal to be used in multiple applications.

Our framework can be improved in several ways. The most important extension to our framework would be the possibility of unreliability communication. But the reliability communication is still useful in inner parts of our framework. The user should have the possibility to choose between these two ways. Not all applications have the same requirements. So we need to keep this framework as universal as possible. In the forums we can already see the requests from the users of the gRPC for the unreliable communication. So the gRPC will have this support very likely.

Actually the decisions how should be the components connected are very simple. For a real deployment it is necessary to implement a really complex logic. We also would need to get enough information about the network. So this part may not be possible to be done absolutely universally. We have implemented a way to compute the latency between the clients and the computation servers. Using the data from this computation we might be able to get a decent results.
The most important part of it is that we can dynamically change the connections according to the state of the network. We cannot think about all possible problems. Sometimes there can happen a unpredictable failure in the network. But our framework has possibilities to react on such unpredictable problems.
Bibliography


List of Figures

2.1 Simple cloud computing ........................................... 6
2.2 Simple edge-cloud computing ................................. 6
2.3 Detail communication in edge-cloud ................. 7
2.4 Schema of a single RPC call ................................. 9
2.5 Load balancing using the proxy server ............. 13
2.6 The client aware load balancing ....................... 13
2.7 Load balancing with the extra server ............... 14

3.1 Base architecture of the edge-cloud framework ....... 17
3.2 Detail architecture of the edge-cloud framework ...... 18
3.3 Example of the period setting ............................... 23
3.4 Workflow of the time computation ....................... 27
3.5 The meta model of our framework ....................... 28

4.1 Visualization ......................................................... 36
4.2 Recognized face ..................................................... 39
Attachments

CD containing following files:

- AndroidApk - A folder containing the built Android applications.
- AndroidDoc - A folder containing the documentation for Android implementation.
- AndroidSrc - A folder containing the source files for Android implementation
- PythonDoc - A folder containing the documentation for Python implementation.
- PythonSrc - A folder containing the source files for Python implementation
- thesis.pdf - This thesis.