A First Design Approach to a Multimedia SDK Based on a Hybrid P2P Architecture

Cosmina Ivan*, Radu Cojocari

Department of Computer Science, Technical University of Cluj - Napoca, Romania

Copyright © 2015 by authors, all rights reserved. Authors agree that this article remains permanently open access under the terms of the Creative Commons Attribution License 4.0 International License

Abstract The Internet multimedia streaming increased proportional to the number of streaming users and from 2005 *peer-to-peer media streaming* received a substantial amount of research attention and was applied for both live and on-demand video streaming. This technique succeeded to provide a large number of multimedia streams while consuming less bandwidth than in the case of a client-server architecture. Multimedia streaming is a complex subject, it widens over various computer science fields as the networking area, multimedia compression area and the security area. Due to the increasing need of multimedia streaming applications and the need for continuous communication with harsh constraints such as real-time communication, low bandwidth and content security, the need for a flexible and extensible tool is justified, and the main purpose of such a tool is to facilitate the development of applications such as Goobar [9], IConf [10], Ekiga [11] or Skype [12]. The responsibilities of such SDK are to capture efficiently multimedia information from a web camera and/or a microphone and send them to its peer. The proposed SDK was built on the .NET Framework 4.5 based on a *hybrid peer-to-peer* architecture. The SDK can be integrated on multiple .NET platforms such .NET Framework 4.5, Silverlight, and Windows Phone 8, and due to its flexibility it can be used by desktop clients, web clients and mobile clients. From a communication perspective, the SDK starts several independent services which capture incoming data, and uses dynamic proxy objects to send data to its peers, services which assure the necessary degree of parallelism needed to have a responsive application with real-time communication.

Keywords Peer to Peer, SDK, Multimedia Streaming

1. Introduction

In the past decade the appetite for bandwidth in the Internet has grown due to numerous sources of multimedia. Nowadays, multimedia streaming has become a need, thus a

huge demand of multimedia processing application exists, from online video and audio playback to online video calling. The need to communicate over the Internet in different ways is in a continuous growth. This along with the advances in multimedia capturing created a bottleneck for various solutions based on client-server multimedia streaming. The peer-to-peer media streaming concept is now an appealing architectural approach, as he reduced the impact on the bandwidth .Due to advances in media compression technologies and accelerating user demand, *video streaming over the Internet* has quickly risen to become a mainstream application over the past decade [1] .An overview of the history of the Internet shows its main milestones in the past decade of research and development. During the 1990s and early 2000s, research attention was focused on client-server video streaming, and new streaming protocols such as Real-Time Transport Protocol [2] were designed specifically for multimedia streaming. This protocol was used on media players installed as the clients receive multimedia streams from a server over the Internet, and this approach was the client-server multimedia streaming.

The main purpose of our project is to build a SDK -software development kit, capable of text transfer, voice and video streaming in unicast mode, based on a *hybrid peer-to-peer architecture*. As secondary goals, modularity and extensibility will be taken into consideration as well as building a working demo that consists of a client that uses the SDK.

The main goal of the project was to create a flexible and extensible architecture that can be used by desktop, web or mobile clients that run on .NET framework for video, audio and text streaming. The proposed underlying architecture of the SDK must be a hybrid peer-to-peer architecture. In many applications the server represents the single point of failure of the application. In a peer-to-peer architecture this is not the case due to the fact that the whole communication does not go through the server, but it represents a direct connection between the peers. Moreover, if the peers happen to be in the same local area network, but neither of them have access to an internet connection ,such that they cannot access the server, they still should be able to communicate, given if

they know their endpoints. The architecture of the proposed SDK eliminates this single point of failure by bypassing the server if the server is not reachable for whatever reason, and gives the user the possibility to specify the endpoint of the peer it wants to communicate. This means that the SDK uses a *failsafe server to access a database* in order to retrieve the list of possible endpoints. Aside the fact that the SDK must be extensible and flexible, it also must deliver the information in real-time, thus the chosen architecture model must take also this constraint into consideration as well.

The secondary goal of the project was to create a modular system design, i.e. each component of the system must be replaceable and/or extensible. This property of the system enforces other attributes such as flexibility, extensibility which gives the user the possibility to build custom objects on top of existing objects provided by the SDK to fit its needs and maintainability which it gives the user the possibility to replace custom components.

This paper is organized as follows. In the second section we present bibliographic research for the project, on specific subjects as the transport protocol, security concerns and development methodology and also present comparatively two similar projects. In the third section we discuss system design, containing functional and nonfunctional properties of the proposed system, identifying the most appropriate technological perspective for developing the system, and detailing some aspects from the implementation of the system components. The fourth section contains a discussion, followed in the last section by some conclusions and further developments.

2. Bibliographic Research

This project proposes a solution for live communication that exploits the advantages of the peer-to-peer topology. From a technical point of view, multimedia streaming is a challenging subject where each variable of the problem requires fine tuning. The first and most important design decision must be choosing the transport protocol. In multimedia streaming with high quality of the transmission is essential and also the integrity of the transmission must be assured, so the second decision must be about choosing the right encryption algorithm. The vast literature on cryptography provides many encryption solutions, but a naïve approach is not desired because multimedia data is not static data. The third design decision must take into consideration the patterns and practices that are necessary in order to build high quality code. In what follows the problems stated above will be discussed. Following these design decisions, we present the findings from the literature, which support the implementation process.

2.1. Networking and Transport Protocols

Nowadays choosing the right transport protocol to fit the constraints of the real time communication is a cumbersome

task, for which extensive research needs to be done in order to make the right choice.

TCP vs UDP vs others. Quite a few protocols have been standardized for streaming communication such as UDP, TCP, real-time transport protocol (RTP), and real time control protocol (RTCP). UDP and TCP are lower-layer transport protocols while RTP and RTCP are upper-layer transport protocols which are implemented on top of UDP/TCP. As mentioned in [3], UDP and TCP protocols provide functions such as multiplexing, error control and congestion control. The similarities between TCP and UDP are that they both allow stream multiplexing from different applications running on the same machine with the same IP address and both employ the checksum to detect bit errors. If a single or multiple bit errors are detected in the incoming packet the TCP/UDP layer discards the packet so that the upper layer will not receive the corrupted packet. Here, in contrast with UDP, TCP provides reliable retransmission to recover lost packets. Therefore, TCP provides reliable transmission while UDP does not. Moreover, TCP employs congestion of control to avoid sending too much traffic, which may cause network congestion.

TCP provides flow control to prevent the receiver buffer from overflowing while UDP does not have any flow control mechanism. Since TCP retransmission introduces delays, UDP is typically employed as the transport protocol for multimedia streaming but it doesn't guarantee packet delivery and the receiver needs to rely on upper layer to detect packet loss. As stated in [1] the disadvantages of using UDP are that UDP is an unreliable and non-congestion control protocol. Packet loss occurs during video streaming in UDP because of its unreliable service and UDP is in need of the error correction and retransmission mechanisms to avoid packet loss. However, the above mechanism has certain drawbacks. It is very difficult to implement efficient retransmission mechanisms and it increases overhead at the client side.

In contrast with UDP comes TCP with advantages like reliable congestion control. With TCP error recovery and error concealment mechanism are not required. TCP provides selective frame transmission and the proxy can be designed in such a way that it provides flexibility in selecting the frames to be transmitted. TCP is bandwidth adaptable in nature. Even if congestion occurs TCP utilizes the resources using that bandwidth.

In comparison with traditional protocols, new dedicated streaming protocols were designed and implemented. These protocols were standardized by the Internet Engineering Task Force as RTP/RTCP/RTSP. RTP is a transport protocols based on the UDP and is defined as a standardized packet format for delivering streams over IP and is designed for end-to-end real-time transfer of stream data. The RTP Control Protocol also based on UDP, is designed to monitor transmission statistics and quality of service and to achieve synchronization across multiple streams.

Congestion control. The TCP has a certain capacity called

transfer window. If we want to send data from Point A to Point B we load data into the transfer window and wait for an acknowledgement. Point B will send an acknowledge signal telling Point A that all those packets have been received. If we're successful, then the TCP becomes optimistic in the sense that it widens the transfer window so that it can send more data at the same time. If the transfer failed for whatever reason, then the transfer window shortens. This produces a slower traffic.

TCP makes use of sequence numbering, congestion window and retransmission timer mechanisms to achieve less congestion and reliable service. TCP sender assigns sequence number for every packet sent and expects an acknowledgement before proceeding with further data transfer. Congestion window is used to perform congestion control, which keeps track of the number of packets that can be sent by the sender without being acknowledged by the receiving side. Basically, congestion control window decides whether TCP sender is allowed to send packets at any particular instance. TCP accomplishes reliable data delivery by deploying retransmission timer mechanism which detects packet loss and retransmits them. If an acknowledgement is not received before the expiry of the retransmission timer, TCP retransmits the packet and triggers congestion control.

Alternate trigger for congestion control mechanism is duplicate acknowledgement arrival at TCP sender. TCP receiver sends a duplicate acknowledgement if the packet is received out of order. When the TCP sender receives duplicate acknowledgements beyond a certain threshold, it assumes a packet loss and fast retransmission and fast recovery mechanisms are triggered. To conclude this feature of the TCP assure reliable transmission and with it an increase in performance for static streaming. While the purpose of congestion control is to avoid congestion, packet loss is inevitable in the internet and may have significant impact on perceptual quality. Error control was a set of strategies used to ensure the smooth streaming even when there were errors in the packet delivery. In [3] the following error controls mechanisms are presented: Forward Error Correction (FEC) and Delay-constrained retransmission. The principle of FEC is to add redundant information so that the original message could be reconstructed in the presence of packet loss. Delay-constrained retransmission is usually dismissed as a method to recover lost packets in real-time video, since a retransmitted packet may miss its play-out time.

A summary of the features of the two protocols discussed so far are presented in Table 1, where we can see that a tradeoff needs to be made between performance and stream data integrity. TCP generally provides good streaming performance when the achievable TCP throughput is roughly twice the video bitrate, with only few seconds of startup delay. Based on the presented analysis, the conclusion is that although TCP has a slightly slower speed but it compensates with congestion control and error control out of the box which guarantees stream data integrity and quality.

Table 1. TCP vs UDP summary

Features	TCP	UDP
Reliability	Reliable. Uses Sequence number and Ack. Avoids Packet loss	Unreliable. No sequence number and ACK so Packet loss occurs
Error Recovery and Error Concealment	TCP is a Congestion control protocol. No need of error recovery and error concealment mechanisms	UDP is a non-congestion control protocol. FEC, Retransmission mechanisms are needed
Firewall penetration	TCP penetrates with the firewall by means of HTTP	UDP does not penetrate firewall
Selective Frame transmission	TCP provides selective frame transmission	UDP do not provide selective frame transmission

2.2. Security

Security is an important part of most applications today, especially connected systems applications. When we're building a connective system and we're transmitting information across the wire that might be a value to an adversary, then we really must plan to be attacked and we need to take precautions in our connected system architecture to figure out how we're going to prevent those attacks. When we think about security in a connected system, there's usually three basic types of protection that we need: When building a connected system we have to decide what level of protection we're going to need in each of the three different areas. When we're defining the communication services we need to think about how sensitive the information is that's being transmitted across the wire for each operation. We also need to decide what protection level each piece of information requires. We need to decide whether we want to use *Transport or Message-Based Security* on the connected system. We also need to decide what authentication protocol we need to use to figure out who the caller actually is. And finally, we need to decide how to implement the authorization logic which basically determines what we're allowing the callers to do.

Transport security vs Message-Based Security. One of the nonfunctional requirements is the security of the application. In what follows we will describe various methods by which confidentiality, data integrity and authentication can be integrated with our system. Each transport typically has a build-in security layer that was designed specifically for that transport and so it will define what kind of credentials you can send and how to configure it for different types of security mechanisms that we might want to take advantage of HTTP using SSL, TCP/Named Pipes using Kerberos, MSMQ using certificates.

We can see that every transport comes with a different implementation for security and will have different constraint around what kinds of credentials you can use to represent the client. We need to realize that transport security constraints us to point-to-point security in our connected system. Those security implementations could indeed be

completely different using different types of credentials. Therefore it becomes very difficult to secure the logical connection between peers. In this case it's hard to authenticate messages arriving to ensure that they came from expected source for example. So we end up on having to rely on point-to-point authentication and we have to implement some kind of protocol transition within the routers which can turn out to be problematic and difficult to implement correctly while maintaining the security of our system. Typically the option of transport security falls down if our architecture requires intermediaries, if not, then that may not be an issue and transport security may suffice. A lot of these security mechanism that are built into the transports like SSL have been around for a long time and they've been significantly improved and optimized. So when using transport security we'll end up with a better overall security solution and we'll also benefit from better performance.

With Message-Based security, we're essentially taking all the authentication information that exists in the transport headers and we're shoving it down into the actual SOAP message. We put that information into the SOAP header section using the WS Security Header elements. This is the main difference between Message-Based security and Transport security: we're essentially doing the same types of things only we're encoding all of that security information within the SOAP envelope. This makes it possible to use the same security implementation for our connected system over a wide variety of different transports. Thus we'll have a transport agnostic security solution but we'll still have confidentiality, integrity and authentication mechanism provided through this XML based technique. This solution offers flexibility in terms of what kind of credentials we can use within the message, but also what transports we can push those messages through as they propagate throughout our system. We can still use multiple transport across there different hops, but the security implementation is at the SOAP envelope level. Thus it doesn't really matter what transport security is used to transport those individual SOAP messages as they flow through our system. The main obvious benefit of Message-Based security is the fact that it increases the flexibility of the connected system architecture. The major downside to Message-Based security and probably the biggest one in a lot of ways, is that the performance can be significantly worse than Transport-Based security. The messages really grow in size tremendously and it takes a lot longer to process those messages on both side of the wire.

In the end it makes sense to use *message based security* when we have an architecture that warrants it. In other words, if you have an architecture where there are intermediaries or routers in between and we're going to need a transport neutrality, around your security implementation. If that isn't the case, we're probably better off with Transport-based security.

2.3. Prism Software Development Methodology

Prism represent [6] a set of guidelines and methodologies,

proposed by Microsoft, that allows us to architect our application such that they evolve and will stand the test of time, that do not break the second you change something. In its simplest form, Prism is simply a composite application framework that allows you to split down a large application into smaller and more manageable pieces. Prism relies heavily on design patterns to help promote loose coupling and separation of concerns. Some of the most common used patterns by Prism are: Dependency Injection pattern, Inversion of control pattern, Command pattern, Model-View-View-Model pattern, Model-View-Presenter. Prism was designed around architectural design concepts such as separation of concerns and loose coupling. This allows Prism to provide many benefits such as: Reuse, Extensibility, Flexibility, Team development, Quality of the code.

2.4. Similar Systems

There are numerous vendors out there that provide similar solutions. The module stacks are the same but the fine tuning differs from vendor to vendor. In scientific literature, two systems were identified with a high degree of similarity but both of them are commercial solutions. A wide used commercial solution is *Goobar's VIVO Engine SDK* [9], It offers real time communication over IP. It contains elements necessary for expanding application features like VoIP and video communication. It also provides a wide variety of audio and video codecs adjustable to bandwidth. It uses SIP signaling standards over TCP and UDP. As transport protocols it supports RTP/RTCP, SRTP, UDP and TCP. From a multimedia point of view it provides a voice quality optimizations like automatic echo cancelation, noise suppression and automatic gain control. Regarding the video engine, it supports a wide variety of codecs like H.263, H.264 AVC and H.264 SVC, LSVX and LSVX-S, video 3D effects, recording and playback capabilities and a synchronization mechanism with the audio packets. All in all, it is a robust solution but it has some important faults like lack of security and no default protocol set. It lets the user to make this important decision, potentially leading to performance issues.

Table 2. Similar solutions comparison

	Goobar [9]	iConf [11]	Our project
Architecture	Client-Server	Client-Server	Peer-to-Peer
Performance bottleneck	Server	Server	Peer
Injection of custom components	No	No	Yes
Configurable security	No	No	Yes
Multi-threaded	No	No	Yes
Extensibility	No	No	Yes
Offline Capability	No	No	Yes

Another widely used commercial solution is *iConf SDK*

[11]. It provides functionalities like video/audio/text conferencing and screen sharing capabilities. Like Goobar, iConf SDK also offers real-time communication over IP but doesn't provide such a wide variety of transport protocols such as Goobar. Another advantage for iConf SDK is that it facilitates communication between NATs and firewalls. Although this is an interesting and useful feature it may be the source of a bottleneck from a performance and communication point of view. Unlike Goobar, iConf is a modular system composed of 3 components: client, server and UI solution. From a multimedia perspective iConf SDK supports DirectX compatible video capture devices and webcams with a wide variety of compression methods. iConf SDK offers extra functionality like screen sharing, ability to record incoming video streams and file transfer.

3. System Design

3.1. Functional and Nonfunctional Requirements

For our project, the functional requirements that will be specified will refer to the client application. The end result of this project will be a client application that uses the SDK to achieve some functional requirements. There is also the possibility of proving the value of the SDK by extensive testing, but the implementation of a client application is preferred simply because, it can reveal some design issues starting right in the development phase of the client application, which can be documented and fixed. The SDK along with the test application provides the users the following functionalities:

- **User registration** -At application startup there is an option to create a new user. A new window pops-up where the user is requested personal information such as username, email, password, phone number, and other personal information. After the information inputted by the user is committed, the user can login. (the functionality is available if Internet connection exists).
- **Login** -At application startup the user can login with its username and password if an internet connection exists.
- **Contact list** -After user login, the users contact list is displayed. This list contains all the contacts of the user that are online at that time. It shows basic information of the users like their IP, their username, email, avatar picture.
- **Add user to contact list** -If an internet connection exists, the user can search and add an existing user to its contact list.
- **Session initialization** -Before any connection is established between two peers (audio and/or video call), the session need to be initialized. This means that the receiving peer needs to provide his consent so that the connection can be established. Furthermore, in this step a handshaking

communication establishes the parameters of the communication like the encryption algorithm and quality of transmission. Only after this step an audio and/or video connection can be established.

- **Text messaging** -When the user clicks a contact from its contact list, a text box opens where the user can start messaging. From the point of view of the receiver, when it receives a message, a pop-up opens with the received message and with text box where the user can respond. When a video or audio session is open, the user can still send text messages.
- **Video call** -When the user clicks a contact from the contact list, a window opens allowing the user to make a video call. When the user clicks the video call button another control opens that shows a preview from the user's webcam in the left hand side, and the video received from the called peer shown in the right hand side. In the same control there is an end call button that can be pressed at any time. From the receiving peer's perspective, when there's an incoming call, a popup appears, asking the user if it wants to accept or decline the call. If the user accepts, the connection is established and the communication starts. If it declines the incoming call, the calling application terminates the execution of the video call.
- **Audio call** -When the user clicks a contact from the contact list a control opens allowing the user to make an audio call. When the user clicks the audio call button another control opens that shows the length of the call and the end call button. If the user accepts, the connection is established and the communication starts. If it declines the incoming call, the calling application terminates the execution of the video call.
- **Offline capabilities** -At application startup, if no internet connection is detected, the application starts the fail safe mechanism. This gives the user the possibility to use the application with no internet connection but with LAN capabilities if the user is part of a LAN. Due to the fact that the SDK is built on a peer-to-peer architecture the offline capabilities are provided in a natural way. If there is no internet connection the application cannot connect to server to retrieve the contact list. There is no reason why the application should not continue just because it cannot get the contact list, because maybe the person that it tries to reach is in the same LAN as the user. In order to start a connection with another peer in the same LAN the user must know the IP of the peer that he wishes to contact. Also the peer must be online at call time. This is one of the key functionality that differentiates the SDK from other products.
- **Text spellchecker** -When writing text this feature will underline all words that are spelled incorrectly in English.

Defining the nonfunctional requirements in this early

phase of the document is essential in order to understand the possible architectures the project can have. From a functional point of view, this project can be implemented in multiple ways. The nonfunctional requirements represent the constraints that are imposed to the system. Those constraints need to be satisfied. Thus the design of our system is modeled regarding the nonfunctional requirements. It is important to understand the need of these nonfunctional requirements so that we can understand later, why a specific design was chosen.

- **Scalability** .This non-functional requirement is the factor that decides how the architecture is going to look like. It is clear that in client-server architecture the scalability is problem due to a single point of failure. The increasing number of users will increase the workload on that server, and will increase the network usage. Hence, we need to consider the peer-to-peer architecture that does not have a critical point of failure.
- **System security** .The security of the system is critical as it manipulates sensitive, personal data. On the server side, sensitive data such as user information is protected against SQL injection due to the fact that all the queries on the server side are written as *Linq expressions*. Moreover, the input parameters are parsed and verified for malicious input values, such as SQL queries. The communication channel between the peers and the server is secured due to message security mode that acts on the http binding. This security mode uses message security for mutual authentication and message protection. The security of the information that is sent by the peers also needs to be considered, thus the communication channels for text, audio and video use *NetTcpBinding*. This means that the caller must provide windows credentials for authentication and all message packets are signed and encrypted over TCP protocol. The security mode can be customized for this binding by configuring different values for the client credential type.
- **Accessibility** .The application is accessible from any location as long as internet connection or LAN connection exists. If the user has access to an internet connection he can use the whole functionality of the system. If only a LAN connection is provided the user has access only to offline capabilities.
- **Availability** .Theoretically the system is available 24/7 due to the fail safe mechanism. If for whatever reason the server is not available, or not internet connection exists, the fail safe mechanism kicks in and the offline capabilities are available allowing the user to make calls inside the LAN. Although some of the functionality of the application will not be available such as adding or removing contacts into the contact list or calling a contact from the contact list, the system make the most of the LAN connectivity allowing text messages, audio and/or

video calls in the local area network given if the user knows which IP to call.

- **Extensibility** .The system architecture is design to include hooks and mechanisms to customize the system behavior without having to make major changes to the infrastructure of the system. The infrastructure obeys the open/closed principle and Liskov substitution principle that increases the extensibility of the system.
- **Maintainability** .Due to the patterns and practices applied in the development phase the system is easy to change or modify without great expense.
- **Portability** .The SDK achieves portability by being able to run on multiple platforms such as .NET framework, Silverlight, Windows phone 7, Windows phone 8.
- **Performance** This non-functional requirement is the key factor that modeled the architecture from a client server, to a hybrid peer-to-peer architecture. This requirement is affected by the transport protocol, the security level of the encryption, the quality of the information and the internet speed. Because of the multitude of variable that affect the performance, the transport protocol cannot be modified by the users of the SDK as it may have a significant impact on the performance. Regarding the security aspect, the user can select a security mode, or no security. Note that higher security leads to a decrease in the performance. This is the reason why an extensibility hook is not provided for extending the encryption.

3.2. Technological Perspective

The technologies that were researched for this project will be classified and compared with their alternatives taking into consideration the context that they are used. The SDK is built on top of the .NET Framework 4.5. It is implemented as a portable class library that supports interoperability with, Microsoft technologies such as: Windows Presentation Foundation (WPF) ASP.NET Webforms ASP.NET MVC Winforms. Moreover, due to the portable class library capabilities, the SDK can be used for projects on other platforms like: Windows 8 Window Phone Silverlight 8 Silverlight Windows Phone 8.1 This high degree of interoperability that the portable class library offers, increases the range of potential users.

Technologies used for the centralized server component .This component is responsible for managing the endpoints and implicitly the contact that the project works with. Due to the fact that the architecture is a peer-to-peer hybrid, the application is self-contained, i.e. it does not depend on this component, but it uses it to CRUD type operations on the appropriate entities.

WCF vs Web API vs NET Remoting. The functionality of the centralized server is exposed as a web-service through Windows Communication Foundation (WCF). There are other ways by which the remote functionality can be exposed

such as .Net Remoting of Web API from Microsoft, but neither of them can offer interoperability, security and performance, all in one. Interoperability of the WCF is ensured by the way it publishes its service and data contracts. The WSDL ensures a cross-platform communication, thus adding a high degree of flexibility to this component. Due to its flexibility we can configure the security aspect of the web-service to fit our needs. If configured correctly, the WCF service can ensure confidentiality, integrity, and authorization. Confidentiality and integrity can be guaranteed at transport level or at message level, or at both, without a significant performance penalty. The authorizations in WCF provide flexibility in the way we define the authorizations. New authorization levels can be defined and configured by the developer, or the WCF service can use the windows accounts that access the service. In this case, the developer doesn't have to define the authorization roles, but it needs to configure the levels of authorization.

Table 3. WCF vs Web API vs .NET Remoting

	Advantages	Disadvantages
WCF (used by our system)	Multiple transport protocol. Configurable security. 25%50% faster than WebAPI. 25% faster than .NET Remoting. Scalable.	The client application depends on the Service and Data Contract.
Web API	Light-weight. Easy to use.	Only supports HTTP transport
.NET Remoting	-	Slower performance

Entity Framework vs LLBLGen PRO .The functionality that the web service exposes are CRUD operation on, basically, the domain model. The domain model is a separate project model, with Entity Framework to resemble the domain model of the application. There are multiple options for the ORM and Entity Framework and LLBLGen Pro were considered. Both ORM's support operations on graphs, i.e. if multiple, related entities are modified and then the root of the graph is saved, all the affected entities are going to be updated. Furthermore, both ORM's support prefetch paths. This allows us to write optimized queries that return an optimal amount of data.

Table 4. Entity Framework vs LLBLGenPro

	Advantages	Disadvantages
Entity Framework (used by our system)	Integrated with the IDE and MSSQL.	An entity graph is not saved by default.
LLBLGenPro	Entity holds context information. Offers UOW functionality.	Not open source.

The main disadvantage of LLBLGen is that it cannot be integrated in the IDE, and comes with its own configuration environment. Moreover, LLBLGen Pro is not free. The advantages of Entity Framework is that it can be fully integrated in the development environment and plus, is free. The backend database was built using Microsoft SQL Server.

Besides the high throughput and performance that it can offer, it can be easily integrated with Entity Framework, and furthermore, the development environment facilitates the integration between the two. AutoMapper is used to facilitate the transformation on entity objects to data transfer objects.

Unity is Microsoft's implementation of an Inversion of control container. It is the only one in the application that is allowed to instantiate new objects, thus facilitating extensibility, reuse and maintainability.

Table 5. Unity versus other DI Container

	Advantages	Disadvantages
Castle Windsor	. Understands Decorators. Typed factories	Query API in places
StructureMap	Works in many cases	No interception
Spring .NET	Interception. Comprehensive documentation	XML centric. No convention-based API. No custom lifetimes.
Autofac	Simple API	No interception Only partial support for custom lifetimes
MEF	Available in .NET 4 and Silverlight 4	Not a real DI container. Configuration based on static attributes. No XML support No interception
Unity (used by our system)	Interception Good documentation. Consistent API.	No convention-based API

SDK technologies .The core of the SDK are represented by the services that handle the streaming process. Currently, they are implemented with Windows Communication Foundation. Before this implementation, other technologies were considered, like .NET Remoting or Microsoft's implementation of the asynchronous sockets. WCF was preferred instead of Asynchronous sockets for the simple fact that it facilitates communication between two private networks. If the socket implementation were to be used, this situation needs to be handled, and the solution is not trivial. In the design phase, multiple approaches were considered when implementing the streaming service with WCF.

The first approach was to host the streaming services in Internet Information Services (IIS) and implement the communication between the application and the streaming services with MSMQ. After some experiments, we concluded that MSMQ impaired the streaming performance of the application. The read and write operations are costly. To eliminate this bottleneck we decided to host the streaming services directly in the application. By self-hosting the services in the same application domain, we can achieve a high throughput communication between the streaming services and the application. Moreover, the integration of the application is more maintainable, we don't have to manually configure in IIS the streaming service, the SDK does this automatically.

Another advantage of self-hosting the streaming services is that we can obtain a low coupling with the SDK by using Delegates. In .NET, delegates are powerful constructs that

offer communication between components while maintaining a low coupling between them. Furthermore, if in the future, the streaming service were to be changed with other components, the rest of the SDK will not change.

In the MSMQ implementation, we would have to implement some components that would handle the reading and writing in the queues. And all of these components needed to be thread safe. The synchronization overhead is eliminated with the current implementation. The SDK handles the self-hosting of each streaming service on another thread. This means that when some requests come in they are handled separately by each thread on each service. The only synchronization that needs to be done is at the UI level and is trivial.

If, in the future, the services need to be implemented with the asynchronous sockets, the integration would be trivial due to the fact that the use of the inversion of control container isolates the impact of a change. The probability of switching to an implementation of the streaming services with asynchronous sockets is high. They are low level components and offer a higher degree of performance. The WCF offers high flexibility, but has a minor effect on performance. Moreover, if the asynchronous socket is considered, we need to take into consideration the security aspect.

With WCF, the security aspects can be configured to fit the needs of the project. With asynchronous sockets however, we need to build a component that handles the encryption and decryption. Another aspect that needs to be considered is the authorization rules. In the future, the product may evolve on different branches, and offer different functionalities for each branch. Thus, for the free product, the product can offer limited functionalities, while the premium branch offers all the functionalities. With asynchronous sockets, we would need to handle this manually, but WCF's built in authorization mechanism helps us accomplish this, while keeping the code clean, and not clouding it with components that, from the users perspective, do not add value to the product.

Table 6. WCF with MSMQ vs Self-hosting WCF vs Asynchronous Socket

	Advantages	Disadvantages
Self-Hosting WCF (used by our system)	No inter-process communication	-
WCF with MSMQ	No need for thread synchronization	Performance reduced due to slow operations with MSMQ
Asynchronous sockets	Could obtain higher performance	Programming overhead. Need to deal with unmanaged code.

DirectShow vs Expression Encoder. The SDK provides to build in modules that handle the content capturing. The two

modules, in essence, are wrappers over two open-source frameworks.

It is important to mention the framework that is being used by the module. It's called DirectShow and it's a multimedia framework and API produced by Microsoft to perform various operations with media files or streams. It is based on the COM framework and it provides a common interface for media across various programming languages. Despite other frameworks that provide a friendlier interface, DirectShow offers higher performance due to the fact that it provides access to hardware devices.

Table 7. DirectShow vs Expression Encoder

	Advantages	Disadvantages
DirectShow (used by our system)	Low level library. Provides access to hardware device.	Cannot access stream only via snapshot => static streaming.
Expression Encoder	Friendly interface.	Cannot access stream in memory, only by exporting it into a file or streaming on webservice.

The second module handles the audio capture. The Framework that is used by this module is called NAudio. It is an open-source audio library that offers many functionalities that increase the development speed of the application. The main reason for choosing this library, besides the friendly interface, is simply because it can be easily integrate with the application and the development environment facilitates this.

3.3. Implementation Aspects

In this section we are going to present some important implementation aspects on our project. We will take a look at the server side design, client side design and the detailed architecture of the SDK, and then, we will go into each major component. The project contains three important components:

- The centralized server
- The SDK (which uses the centralized server to locate peers)
- The client application (uses the functionality provided by the SDK)

The below figure emphasis this components and gives a hint about how they are used.

Considering all the advantages of the hybrid peer-to-peer network, the whole system is modeled to achieve this structure.

The SDK hosts in the application domain specialized service that handle incoming multimedia requests. There are individual services for each major functionality i.e. for video, audio, text messaging and signaling. The reason for this is because we don't want to overwhelm a single port with all the information.

Figure 1. Conceptual overview of the major components

Figure 2. System architecture

This would be a bottleneck that would impair the performance of our application. Thus all the service are hosted on different endpoint, they listen for incoming requests at the hosts IP but on different ports. Multimedia streaming is resource expensive if high quality is considered. This high cost is the sum of all the operations performed, starting from the capturing device, where we need to capture frames with a high frequency but also high quality, to the services that wrap-up the content, secure it and send it across the wire to another machine that needs to decrypt, unwrap the content and process it.

All this steps need to happen at the same time in order for high quality communication to be achieved. Thus, we need to expose the right degree of parallelization to achieve higher performance, but also avoid communication overhead between the threads. Each web service is hosted individually

in the application domain, and that each web service is self-contained from the rest of the web services and from the application. The working principle is the following: the Web-services act as receivers; they are responsible of capturing all incoming data and forward it to the application. Each web service runs on a separate thread, and does not depend on the other services. This provides high flexibility for the SDK allowing the end user to realize different combinations of multimedia streaming like , for instance, adding a capability of screen sharing while in a video call and so on. Each web-service handles incoming data and forwards it to the application to be processed.

Thus we can observe an abstraction layer can be created that ensures extensibility and maintainability, but we will talk about this subject in more depth when we'll present the architecture of the SDK. A proxy server is created for each

existing web service. The proxy servers act as senders. They have the responsibility of sending whatever data they receive, and nothing more. As we'd expect, all the proxy servers need to run in the same time, in parallel to offer high transmission throughput.

Once again, another layer of abstraction can be observed; all the proxy servers do the same thing, but with different data types. By adding this abstraction layer, we minimize the future impact of adding another streaming functionality to the application like screen sharing or file transfer. The gains in this case are self-evident. Both the web servers and proxy servers rely on abstraction to form a common way of working, facilitating the adding of new features. All of them in concept behave the same but, one of them acts different at a lower level. The Signaling web-server and proxy server, as the name suggests, is responsible of for the initialization and for the termination of a connection.

This signaling process occurs right before a call starts and ends. It does not do any streaming, but do to the higher abstraction layer that we mentioned early on in this subchapter, it can be modeled as the rest of the services. When then user initiates a call, in fact, it delegates the work to the Proxy service. He fulfills his responsibility and signals the receiver with a call request. The receiving peer, in turn, initiates a response action. This prompts the proxy service to send a Call response to the Signal receiving web service. This, in turn, prompts the actor of the call response and also sends a signal to the peer that it will start the streaming.

The receiving peer acknowledges this and starts the streaming as well. When one of the peers send an end call request, the signaling service intercepts this, prompts the user and notifies the proxy signaling service to send a signal to stop the streaming. The signaling mechanism is crucial for a clean closing of the connection. The absence of this mechanism could leave the system in an inconsistent state, generating memory leaks, open connections and ghost processes.

3.3.1. The Server

In a peer-to-peer network we need to be able to locate peers. There are three approaches to this problem: structured network, unstructured networks and hybrid peer-to-peer networks. Taking into considerations the advantages of the hybrid peer-to-peer architecture motivates the existence of the centralized server. The role of the centralized server is to provide the endpoints to the peers such that a peer can find any other peer in the network. Furthermore, the benefits of the centralized server extend by allowing functionalities such as user login, register and CRUD operations on its own contact list. The centralized server component hosts in the Internet Information Service (IIS), the web service that exposes all the above mentioned functionalities.

The web service is created via WCF. It exposes a service contract which allows access to the desired functionalities. The design of the functionalities that the web service encapsulates is a modular one, composed of three layers: Endpoint layer, Domain Layer and entity layer. In what follows we will describe in detail the implementation of each layer and the purpose it fulfills.

Endpoint Service Layer. This layer relies on the WCF technology to expose some functionality through *IEndpointService* which acts as a service contract. The data contract consists of data transport objects that are modeled with respect to the entity model.

This is a slim layer in the sense that its role is to expose the remote functionality and delegate the rest of the responsibilities to lower layer. By doing so we respect the Single Responsibility Principle, the layer does only one thing. This way, we ensure extensibility, reuse and maintainability. By not including business logic code in our application, we can respond better to change. Technologies come and go, and we need to take this into consideration. For instance, the WCF could be replaced with Web API or Signal R. If this is the case, we have the business logic separated into another layer which will be used by the Web API. Maintainability and reuse is also guaranteed due to the fact that this layer, like all other layer in fact, is governed by an inversion of control container.

Figure 3. Endpoint service class diagram

Figure 4. Domain model class diagram

Domain Model Layer. This layer is used directly by the Endpoint service layer. It's responsible for all the business logic the centralized server needs to offer. In some projects, database changes are frequent; therefore we need to isolate the domain model layer from these changes. If changes in the database manage to bubble up to the domain model layer then the problem is obvious: maintainability is compromised. The Single Responsibility Principle, besides stating that a class should have only one responsibility dictates that a class should have only one reason to change. In our case, the classes in the domain model are responsible for fulfilling a business rule and should change if and only if the business rules change. Thus, by isolating the business logic from the entity model logic, we prevent changes to bubble-up from the database up to the Domain model.

This is done by creating a set of data transfer objects which reflect the current state of the entity model, and two additional classes that handles the mapping between the entity objects and the data transfer objects, and vice-versa. Figure 4 shows the class diagram of the Domain model layer. It can be clearly seen that any dependencies between components are abstracted such that changes are encapsulated. This respects the dependency inversion principle that suggests depending upon abstraction, because abstract things tend to change less compared to concrete things. In order to encourage reuse and maintainability the main component of this layer, the data service, is implemented in a generic fashion.

All the operations that it needs to perform are uniform across all data transfer objects. The data service has the generic parameter *T*, which is constraint to be of type *IDataTransferObject*. This abstraction allows a high degree of reusability while maintaining type safety so that the object could not be used with generic types that are not suited to be handled. The classic approach would be to define a data service for each type of data transfer object and implement the necessary methods. This is code duplication. It impairs maintainability and reuse. If the business changes we would need to make the same modifications in each data service. Since we built a generic type, the business logic is only in one place, works for all data transfer objects and increases code coverage tremendously. The purpose of this layer is to establish a connection with the database. It uses Entity Framework to do so. Based on the database model the edmx is generated which defines the conceptual model and the mapping between these models. The Entity model is used by the Domain model to retrieve desired datasets.

Entity Model Layer. The Repository is built in a very similar way to the *DataService* from the Domain Model. It is built having in mind extensibility, reusability and maintainability. The Repository takes advantages of the generics that the .NET Framework offers, to avoid duplicate code. It uses the *DbContext* generated by Entity Framework to implement the CRUD operations generically, for each entity.

Figure 5. Entity Model Layer

Throughout the data model, the Repository is never instantiated, but it is registered within the Unity container and resolved wherever it is needed. This way of working ensures flexibility and it prepares the code for future changes. If, for instance, the ORM would be replaced, a new repository will have to be created, but due to the fact that component layer relies on abstraction, the replacement of the old component with the new component is very easy, the affected places in the Domain model layer being reduced to only one line of code. The Domain model is a simple one because this is not a database centric application. However, we need to consider the key points of the data model that are likely to change and try to encapsulate them right from the database. Thus, the fewer the changes are at the database level, the smaller is the impact on the entity layer. To achieve this, we needed to make an educated guess about what are the regions of the database that are likely to change. Of course, in the pessimistic case, the whole structure of the database may change leading to major modifications at the entity layer.

During the application lifecycle, new and new features are added iteratively, generating constant change. In the case of our application, the most probable area that is subjected to change are the communication services (ex. Video, audio, text, file transfer, etc.). We need to consider this when designing our database and try to encapsulate future changes. By doing so, the *UserEndpoints* table was broken into 5 more tables that contain the endpoint information, but it may contain additional information as the project progresses. Adding column to a table generates a smaller impact on the entity later, in contrast with adding a new entity [6].

3.3.2. The SDK

This topic covers the core of the project. All the other major components, the centralized server and the client

application revolve around the SDK. The purpose of this software development kit, as stated in the Project Objectives, is to provide functionalities like video, audio and text communication available with a friendly interface, that is easy to use, customize and extends in order to fulfill the user needs[7].

Throughout this subchapter we will present the block diagram of the SDK, we will provide an overview for each of them and after, we will dive into the implementation details of each component.

The major components of the SDK are the Communication component, the Driver component and the Endpoint Services component. From what we can see, most of the components are self-contained, limiting the communication between them. This clearly provides low coupled components, and implicitly high cohesion, that encourage reuse and facilitate maintainability. In any system interaction between components cannot be avoided. If this is not tailored appropriately we would end up with entangled code that will turn up to be unusable. When implementing the communication between two components we need to remind ourselves about some basic principles and guidelines that lead the way to a good design, like the ones described previously.

In our SDK, the only communication that is needed is between the drivers that capture the multimedia content and the Send services which send the data. For low coupling to be maintained, a third component was created which aggregates the two components into one new component that provides the desired functionality. The gain is that maintainability is assured by not mixing the driver component code with the send service component code. For instance, if in the future, we would have to change the source of the multimedia content, we would be needed to build a new component from scratch. But with the current solution, we need just to inject

the new component in the system and it should work just fine.

In what follows we will take a closer look at each individual component, discussing about how they are implemented and their advantages and disadvantages.

The purpose of this module is to capture multimedia content and make it available so that other services can process it. In this paper, when we use the term driver, we think of a component that can capture content from an output device. Our project uses two drivers, one for audio and one for video. They are both wrappers over some third party libraries that have been described in chapter four. The basic idea of the wrapper/driver is to have some control signal that specify to start and stop capturing data, and another signal that is triggered when a individual unit of multimedia can be output. From this we can easily notice an abstraction over the drivers. Thus, a contract has been established saying that each driver should expose the functionality to start stop and export the content that is captured. Due to the fact that each specific driver exports different multimedia units, we need to find an abstraction in order to preserve the overall genericity. In order to obtain a signaling functionality that can return in an asynchronous manner a generic data type we will use the delegate constructs and more specifically the *EventHandler* delegate.

In what follows we will describe the implementation details of the each driver and state its advantages and disadvantages.

Audio Driver. The audio driver has two responsibilities. The first is to capture multimedia units and export them. The second is to play whatever multimedia unit it receives.

As specified in section Technological Perspective, the library that is used to achieve our desired level of functionality is called Naudio. For our first need, the capturing of a multimedia unit, Naudio exposes an object called *WaveInEvent*. This object represents the core of the audio driver. In order of it to meet the desired behavior we need to specify the input device, the recording format and how many milliseconds a multimedia unit represents. The input device is selected by default as the first detected audio device. Adding the capability of selecting the desired input device is the subject of future work. The buffer Milliseconds represents the total time need in order for the content buffer to fill. When this buffer is full it triggers an event with the content collected, it clears the buffers and, in the meantime, while all of this operations are executed, the recording doesn't stop, but it records on a separate buffer and when the buffer is full, it switches with the empty buffer. The recording format is provided by a codec. A codec defines the way the content is encoded and decoded. Our application provides a few codecs that can be used to decode/encode the multimedia unit. Adding the capability of selecting the codec preferred is the subject of future work.

The implementations of each codec are imported from an open-source project that was built in order to add extra capabilities to the Naudio library. All the implementations of the codecs are imported into the application and refactored to

meet the project standards. Figure 6 present the structure of all the codecs implemented so far. The difference in all of these codecs is the decode/encode method, the bits per second parameter which describes the quality of the recording and the WaveFormat.

Returning to the first responsibility, the part of the driver that handles the content capturing is called *AudioCaptureService*. It is exposed as a service via a specific interface which is derived from the *IDriver* interface. This approach allows data encapsulation and functional encapsulation and along with it high cohesion and low coupling.

The second responsibility of the codec is to read a multimedia unit. This functionality is also exposed as a service via its interface, gaining all the above mentioned benefits. Due to the powerful Naudio library, this component may seem somewhat thin. But because of its contract design, the encapsulation that was achieved allows easy extension and facilitates maintainability. Because this is the first version of the product, the audio functionality is rudimentary but extension points were provided for future development.

Video Driver. Unlike the audio driver, the video driver has only one functionality that of capturing frames. As discussed, the video driver is just a wrapper over a third party library, DirectShow. Other third party libraries were taken into considerations, but DirectShow won after a critical comparison. Despite this, a major disadvantage of all third party libraries, that were mentioned and cover this subject, is that they do not allow access to the dynamic stream that is captured, in memory. The user can access the stream in only 3 ways: by writing the stream into a file and then reading the file and converting it, by exposing it with a web service at a specific endpoint, or by taking snapshots of the stream.

The ideal way of accessing the dynamic stream is in memory. It's the fastest, non-resource consuming and secure way to capture the data. Unfortunately, none of the known third parties offer this feature, so we need to compromise, decide which of the third ways of capturing the data is suited for our project and then optimize it. The first method, that in which we write and read into a file is very expensive. It is common knowledge that I/O operations, like writing to this is very expensive. It's time consuming and resource consuming, it would continuously write/read from disk while in a video call. The second method is both resource expensive and exposes a security issue. It suggests that we expose a web-service through which the content is made available. This raises several problems like consuming network resources with no good reason, it exposes sensitive information on a web-service that is susceptible to unauthorized access, and lastly the operation of writing/reading from/to the web-service is expensive and can impact the overall performance.

This leaves us with option number three. To take snapshots of the stream, this operation is nowhere near as expensive as continuously writing/reading from/to the disk/web-service. Furthermore, we can take advantage of the fact the stream is thread safe to take multiple snapshots with

multiple threads. This may increase the overall video streaming quality of the system considerably.

The downside to this method is that we end-up with static streaming. This impacts us in the sense that we cannot include features like codecs and compression methods. This is what we sacrificing for performance's sake and we will use it in the project until the third library is available that suites our needs. Until then, we will try to optimize as much as possible. It is important to know that this service runs on a

different thread that captures 35 frames per second. For now we limit this to only one thread to limit the network workload and the synchronization overhead. Moreover, at some point it would be impractical to increase the number of fps because the limits of the human eye.

Communication. The build blocks for this module are abstracted in two basic components: a service that handles the sending of multimedia unit and a service that handles the receiving of a multimedia unit.

Figure 6. The Codec component

Figure 7. Send Service class diagram

The structure of the *Send Service* is how in Figure 7. The responsibility of the *SendService* is to send a multimedia unit to a specific endpoint. This is accomplished by using the proxy service to create a channel between the two peers.

The first idea that comes to mind when implementing this is to create separate service for each type of multimedia unit. But we can abstract this and transform it into a single service with generic data types. The *ISendService<T>* interface, where *T* is a service type, from Figure 7 shows us how the abstraction was realized. The generic parameter *T* represents the Service type. The Service Type is nothing more than then service contracts exposed by the web-services that the peer hosts. The *T* parameter is used by the implementation of the *ISendService<T>* to establish the type cast of the *ChannelFactory* and when the *Initialize* method is called, the peer endpoint is known and a channel can be open between the two peers so that the streaming can begin.

The second responsibility of the module is to receive and forward to the upper layer whatever messages it receives. Based upon this, we can again abstract this functionality so that we don't end up with duplicate code and four receiving

services that do the same thing but with different data types.

When the *Receive Service* is created, in start to *Listen* for incoming requests. It is important to mention that each is hosted on a different thread and is listening on the current users IP on different ports. This provides a high degree of parallelism in the sense that incoming video, audio and text requests are process in parallel, on different threads. After the incoming data has been received the receiving service forwards it to the upper layer that is managed by the main thread application. This passing of information between threads is done via *EventHandlers*.

It is worth mentioning that the genericity with which the send and receive services have been built offers extensibility in the sense that if future types of services are needed, we only need to create and use their data types. With the reduction of duplicate code maintainability is assured and by not creating a separate class for each individual service, the code coverage is highly increased together with the cohesion of the classes. And finally, due to the fact that the project uses an IOC container, the instantiation of the services is easy because uniformity is guaranteed and the reuse is

self-evident because the whole module, as well as the whole system, relies on abstraction which facilitates dependency injection.

Another important part of the *Communication module* that we need to talk about is the *Transfer components*. The purpose of this class is to realize the communication between the *Drives* and the *Send Services*. This is needed to decouple the two modules so that we can reuse them in other contexts if needed. Basically it aggregates specific instances of the drives and Send Services in order to connect the output of the driver to the input of the send service. Although the *Send Services* are generic, the drivers are not. This means that we need to create separate Transfer service for audio, video and text.

Endpoint Services. This component is a wrapper around the centralized server component, more specifically around its proxy. We could use this component directly in our application but then we would be adding a big dependency of the centralized server. If the Service or data contract of the centralized server were to change, then the impact of that change would be huge, if it is used all around our application. This is why we need to create an adapter that encapsulates changes and minimizes the impact of possible changes.

The centralized web service exposes the methods need by the application but they are not logically grouped. When we create a service around the proxy we will group the functionalities logically based on what they do. That way we will obtain a high cohesive service.

Taking a closer look at the beginning of this chapter, where we described the centralized server component, we can see that we can split the functionalities of the centralized server into to services: *UserService* and *SearchService*. The *UserService* is responsible for all the CRUD operations on the *UserDto* structure and the *SearchService* is responsible for searching and retrieving other *UserDtos*.

3.3.3. The Client Application

The client application has been built as a proof of concept, to show the functionalities of the SDK and to show how the SDK is used. The client application can be of any type, starting from a classic desktop application, and all the way to web and mobile applications. We choose to implement our client application as WinForm application out of convenience. The scope of this project is not that of web or mobile programming.

We might think the important part is over and once we've implemented the SDK our work is done. But that's not true. It's important to do a *proof of concept* showing that the SDK actually work. This can provide us several benefits like revealing functional and design flaws in our SDK that could

lead to drastic refactoring of the SDK. Now that we've established why the development of a client application is important we will continue by presenting the architecture of the client application and how the SDK is used.

The SDK follows the Prism methodology simply because it encourages reuse and high modularity. There is no reason why our client application wouldn't do the same. If at a later point in the development of the SDK, drastic changes are made, the work to maintain the client application, as expected, would increase. Plus when new functionalities are added to the SDK and need to be reflected in the client application, they can be easily added. Although the Prism methodology was intended for WPF applications, if we remind ourselves that these are just guidelines we can end-up by tailoring them to suite our needs for a Winforms application. This step may seem to be an overhead and it is not in the scope of our project but it is worth it on the long run because it is well know that the development of a desktop application is the fastest.

Each functionality of the SDK has an end result in the UI an associated ModelView-Presenter component that separates the logic. The Model-View-Presenter is a UI pattern that is derived from the popular MVC pattern. As the interaction with the model is handled exclusively by the presenter and the view is updated exclusively by the presenter. Because this project uses an IOC container, it is a best practice to program against interfaces, so that the injection can be made relying on abstraction. This way we can easily replace entire MVPs without impacting other areas of the code.

In what follows we will present only the key functionalities of the SDK that are exposed via the client application due to the fact that the methodology is that well established that adding functionalities is just repetitive work and is uninteresting.

We can see that the MVP simply uses all the needed component of the SDK to implement its functionality. As discussed in another section of this chapter *ITransfer<VideoData>* captures and sends the video to a *UserDto* that is the endpoint. Also, this service provides a hook so that we can multiplex the static stream for preview functionality. When the user begins the call functionality, the *InputService* kicks in and prompts the receiving peer of the attempt to start a call session. If the call is accepted, the *IReceiveService<VideoData>* is initialized and start listening for incoming connections.

The audio and text functionality are implemented in a similar manner. The only difference is in the service of the SDK that they use.

Figure 8. Video call functionality

Table 8. System requirements

	Minimum requirements	Recommended
CPU	Intel Pentium Single Core @1.2 GHz	Any Quad-Core processor
RAM	3 GB	4 GB
Sound Card	Needed	
Webcam	Any camera compatible with DirectShow	
Microphone	Needed	
OS	Windows XP SP2	Windows 7 SP1 or higher
IIS	Needed	
MSSQL	Needed	

4. Discussion

The SDK is shipped as a portable class library [8] along with its dependencies (third party DLLs). Hence the user only needs to add it like any other reference. The centralized server comes as a project in order for it to be hosted on IIS as a *Virtual Directory*. Once it's up and running you need to specify its endpoint into the *app.config* file of your application. This is needed by the SDK so that it knows how to access the centralized server.

In software testing world, the Smoke test is a common approach and that we've performed on the prototype. The Smoke testing is a test suite that was performed on the most important features of the application. This can reveal simple failures severe enough to reject the release of the application. A subset of test cases that cover the most important functionality of our system is selected and run, to ascertain if the most crucial functions of a program work correctly (Call User, Send Message, Incoming call, etc.). The main advantage of the Smoke testing is that it can be done on each build, and in the future this can be automated and integrated in the suite of Integration Tests.

Also, for future work our project can use *Integration tools* which are responsible to trigger build events when a new

functionality is added and run the Smoke Test cases automatically. This is an important capability that provides continuous integration of our application with new functionalities. Furthermore from the set of test cases that we've developed will be part of a Regression. The *Regression testing* is developed to find the bug in areas of the applications that, in theory were not touched but the risk of them being affected exists. In other words, the regression tests certain functional areas of the application that were not affected by the new functionality but there is always the risk of bugs appearing in that area. This, in turn could also be automated due to the fact that there are small chances to alter a perfectly functional area. An important observation would be to avoid the integration of test cases in a regression suite that are not stable, i.e. that are frequently impacted by change. This avoids certain maintainability issues of the test cases.

When creating a test case we need to take into consideration the traceability of the requirements. For instance if requirement x is changed, test cases x and y are also affected and they need to be updated. In what follows we will present the test cases for the major functionalities of our project and after that we will present the measurements to see the performance counters of our system implemented with Asynchronous Sockets.

5. Conclusions and Further Developments

The project raised many challenges, from low level issues like determining the transport protocol and dealing with COM objects on different architectures, to higher level issues such as choosing the appropriate mixture of technologies that best suites the project's needs.

The most important part of the project, its *architecture*, was a big challenge in the sense that it we needed to come up with a design that suits all of our current needs and can accommodate many of the future needs. Furthermore in order for the SDK to be usable we needed to make a good design that is flexible and can satisfy the user's needs. Although the entire system design seem to be straight forward remember that '*simple is not easy*'. Easy is a minimum amount of effort to produce the result' and 'simple is the removal of everything except what matters'. Moreover, the integration of many different third party libraries in our system did not affect the structure of the system design. The design was kept clean despite the interfaces of the third party libraries and hence uniform development was maintained. Also we've taken into considerations possible future changes of the third party library and modeled the design in such a way that it allows pluggable components, i.e. we can easily replace the old libraries with other libraries.

At the moment our system supports the basic functionalities such as Call notifications, Video call, Audio call, and rich text messaging in unicast mode, together with a contact list that can be managed. To prove the capabilities of the SDK a client application was built to reflect these basic functionalities.

As the system grows it's important to focus, right from the beginning on the core architecture, to try to improve the system performance and also try to build an architecture that facilitates the integration of common functionalities.

Probably the most important improvement that the system needs is related to the communications infrastructure. As the image quality grows, so do the expectations of the end users. Hence we need to prepare our system to have a high transmission throughout. We can achieve this by implementing our own RTCP to overcome the limits of the TCP. This is necessary because no open source library exists that offers the features of the RTCP. Although this may seem an overhead, the gain is that we are going to be owners for that implementation and this eliminates the risks that a third party library implies, and moreover, we can reuse it in other projects or commercialize it. Furthermore we need to implement a lower level transport system that offers us higher performance. By dropping the WCF services, and using our own, we will need to handle a few extra things that WCF handled for us by default like, transport based security and message based security.

Although it seems like a lot of work changing the communication pipes, it has its benefits. It allows us full

control over the system, and moreover, we can improve the performance of the system in key areas with custom implementations.

From a topological point of view, the system will not change. The current topology benefits from both the advantages of structure and unstructured network and minimizes their overhead.

From a security perspective, the current implementation fits our current needs, but if decided to change the communication pipes, then we will be needed to implement a security module as well. This again, may seem like an overhead, but in fact it's an opportunity to optimize. WCF offers great security at great performance, but that performance can be optimized. WCF doesn't provide security behaviors that bet fit the continuous streaming context. There is an entire literature regarding security in a continuous streaming context, hence, the opportunity to optimize.

REFERENCES

- [1] Li, B., Wang, Z., Liu, J., and Zhu. Two Decades of Internet Video Streaming: A Retrospective View. *ACM Trans. Multimedia Computing and Communications. Appl.* 2, 3, 2013
- [2] Schulzrinne, H., Casner, S., F Rederik, R., AND Jacobson, V. RTP: a transport protocol for real-time applications, RFC 1889. 1996
- [3] Wang, B., Kurose, J., Shenoy, P., and Towsley, D. Multimedia streaming via TCP: An analytic performance study. *ACM Trans. Multimedia Comput. Commun. Appl.* 4, 2, Article 16 , pp45-61,2008
- [4] Chi-Fai Wong Wai-Lam Fung Chi-Fai Jack, Tang S.-H. Gary Chan, TCP Streaming for Low-Delay Wireless Video, *Proceedings of the 2nd Int'l Conf. on Quality of Service in Heterogeneous Wired/Wireless Networks (QShine '05)*, pp356-364, IEEE 2005
- [5] Martin C. Robert, *Agile-Principles-Patterns-and-Practices-in-C#, Prentice Hall*, 2006
- [6] Martin Fowler, Refactoring improving the design of existing code, *Addison-Wesley Pearson Education* ,2002
- [7] Michael C. Feathers, Working with Legacy Code, *Prentice Hall* ,2004
- [8] Steve McConnell, Code Complete, *Microsoft Press*, 2nd Edition. Redmond 2004
- [9] www.goober.com
- [10] www.avspeed.com
- [11] www.ekiga.org
- [12] www.skype.com