Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Aplikace pro decentralizovanou komunikaci

(Application for decentralized communication)

Diplomová práce

Autor: Bc. Tomáš Malík

Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D.

Jilemnice duben 2022

Prohlášení

Prohlašuji, že jsem diplomovou práci na téma „Aplikace pro decentralizovanou komunikaci“ zpracoval samostatně a s použitím uvedených zdrojů.

.....

V Jilemnici dne 24. dubna 2022 Tomáš Malík

Poděkování

Tímto bych chtěl poděkovat svému vedoucímu diplomové práce doc. Mgr. Tomáši Kozlovi, Ph.D. za metodické vedení, odborné připomínky a cenné rady ke zpracování této práce.

Anotace

Cílem této diplomové práce je popsat tvorbu aplikace pro decentralizovanou komunikaci a zároveň implementovat ukázkové řešení. Diplomová práce je rozdělena do dvou celků. První přibližuje problematiku centralizovaných a decentralizovaných služeb, šifrované komunikace a vývoje mobilních aplikací. Nachází se zde také kapitola věnující se popisu technologií, které lze při vývoji decentralizovaných aplikací využít. Druhý celek práce je věnován návrhu konkrétního řešení decentralizované komunikační aplikace, jeho implementaci a následnému otestování. V závěru práce je vytvořené řešení zhodnoceno a jsou navržena jeho další možná rozšíření.

Annotation

Title: Application for decentralized communication

The goal of this diploma thesis is to describe the creation of an application for decentralized communication and to implement an example solution. The thesis is divided into two parts. The first one introduces centralized and decentralized services, encrypted communication, and mobile application development. There is also a chapter dedicated to the description of technologies that can be used in the development of decentralized applications. The second part of the thesis is dedicated to the design of an example decentralized communication application, its implementation, and subsequent testing. At the end of the thesis, the developed solution is evaluated, and further possible extensions are proposed.

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

1 Úvod

Ve 21. století patří online komunikace k běžné součásti našich životů. Může se jednat např. o emaily od zaměstnavatele, video-hovor využívající technologii VOIP[1], či webový formulář určený k reklamaci zakoupeného zboží na e-shopu. Všechny tyto metody spojuje využívání moderních informačních technologii, díky kterým nezáleží na tom, zda se protějšek nachází ve stejné místnosti či na druhé straně světa. Užitečnost online komunikace ukázala také pandemie nemoci COVID-19, která zasáhla celý svět. Součástí proti-pandemických opatření byla i nutnost dodržování izolace, pokud člověk onemocněl nebo se nacházel v kontaktu s někým, kdo byl pozitivně testován na přítomnost nemoci. Díky tomu mohly nastat situace, kdy nebylo možné s lidmi komunikovat fyzicky a muselo být využito alternativních metod. Jednou z nich je právě online komunikace. Výzkum sledující využívání technologií během pandemie odhalil, že až 46 % lidí během této doby značně zvýšilo své používání internetových služeb pro komunikaci. [1]

Tento způsob komunikace byl však hojně využíván již před příchodem pandemie nemoci COVID-19. V roce 2015 v Německu proběhl výzkum, který na vzorku 2418 uživatelů po dobu 4 týdnů sledoval využívání jednotlivých aplikací v jejich telefonech. Bylo zjištěno, že téměř 20 % času stráveného na smartphonu v průběhu dne připadá na používání internetové komunikační aplikace WhatsApp. [2] Autoři konkurenční aplikace Telegram zase uvádí, že v roce 2021 jejich síť přesáhla hranici 500 miliónů aktivních uživatelů. [3]

Z předchozích řádků plyne, že online komunikační aplikace se mezi lidmi těší velké oblibě. Při jejich používání je však potřeba položit si několik otázek. Je např. dobré vědět, jak provozovatel aplikace zachází s daty uživatelů nebo zda jsou jednotlivé zprávy během komunikace šifrovány. Pro některé uživatele může být důležitá i dostupnost dat v případě, že dojde k ukončení provozu aplikace. Bohužel hned několik z nejpoužívanějších komunikátorů výše zmíněné problémy neřeší a uživatelé se tak jejich používáním zbytečně vystavují rizikům. Je proto vhodné zamyslet se nad tím, zda by nebylo možné najít alternativní řešení, která daná rizika eliminují.

[1] **VOIP** je zkratka slovního spojení „Voice over Internet Protocol“. Jde o obecné označení protokolu, který umožňuje hlasovou komunikaci pomocí technologií internetu.

2 Cíl práce

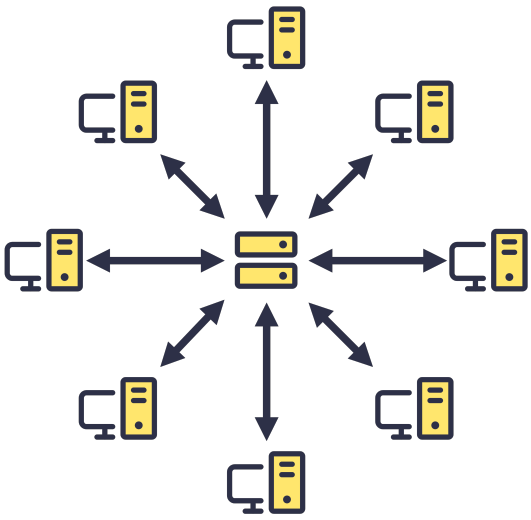
Cílem této diplomové práce je popsat tvorbu aplikace pro decentralizovanou komunikaci. Zároveň by v rámci práce měla vzniknout ukázková aplikace demonstrující funkčnost navrhovaného řešení. Výsledná aplikace se bude skládat ze serverové a klientské části.

3 Centralizované a decentralizované služby

Při návrhu a implementaci komunikační služby lze využít celé řady architektur. Tato kapitola představí několik nejběžněji používaných architektur a uvede příklady služeb na nich postavených.

3.1 Centralizované služby

V současné době patří centralizovaná architektura k těm nejběžněji využívaným. Centrem celé služby bývá jediný server (nebo síť serverů vystupujících jako jeden server), ke kterému se uživatel ze svého zařízení připojuje. Existují příklady centralizovaných komunikačních služeb, které uživateli umožňují provozovat službu na vlastním serveru, avšak většina nejpoužívanějších aplikací využívá pro komunikaci pouze servery provozované vývojáři dané služby.



Obrázek 1: Grafická reprezentace centralizované služby, Zdroj: [autor]

Jedním z důvodů, proč se centralizované služby těší takové oblibě, je snadnější vývoj softwaru. V případě dokončení nové verze služby je snadné aktualizovat server, protože jeho provozovatelem bývá entita, která službu vytváří. Klienti jsou následně nuceni k aktualizaci na nejnovější verzi aplikace, protože např. v případě změny komunikačního protokolu[2] by přišli o možnost službu využívat. Výhodou jediného centrálního serveru je také fakt, že ho musí využívat všichni uživatelé aplikace. Díky tomu je teoreticky možné, aby uživatel kontaktoval jakéhokoliv dalšího uživatele služby. [4]

[2]**Protokol** představuje sadu pravidel, které jednoznačně definují, jakým způsobem bude probíhat komunikace.

Pokud zvolená komunikační služba nepodporuje možnost jejího provozování na vlastním serveru, tak může provozovatel služby omezit či úplně zakázat přístup ke službě klientům třetích stran. Další nevýhodou v takovém případě je, že data uživatelů se nachází na serveru, který v jejich správě. Uživatelé tak musí věřit provozovateli, že jsou jejich data dostatečně zabezpečená a že je s nimi nakládáno tak, jak bylo definováno v podmínkách používání aplikace. [4]

Centralizovaná architektura má velké zastoupení mezi službami pro komunikaci v reálném čase. Dvě nejpoužívanější jsou vlastněny společností Meta Platforms. První z nich je aplikace WhatsApp, kterou aktivně využívají 2 miliardy lidí. [5] Druhá nejpoužívanější aplikace je Facebook Messenger, kterou využívá přibližně 1,3 miliardy lidí. [5] Následující dvě příčky jsou obsazeny službami WeChat a QQ Messenger, které se těší popularitě především v Číně a jsou aktivně využívány 1,2 miliardami resp. 591 milióny lidí. [5] Významný podíl na trhu mají také aplikace Telegram s 550 milióny aktivních uživatelů a Snapchat, který používá 538 miliónů lidí. [5] Mezi další významné služby používající centralizovanou architekturu patří aplikace Skype, Discord, Microsoft Teams a Viber.

3.2 Decentralizované služby

Kromě centralizovaných architektur existují i alternativní řešení, která nespoléhají na jeden centrální prvek. Označují se jako decentralizované architektury a v rámci této kapitoly budou blíže přiblíženy dva různé příklady takovýchto řešení.

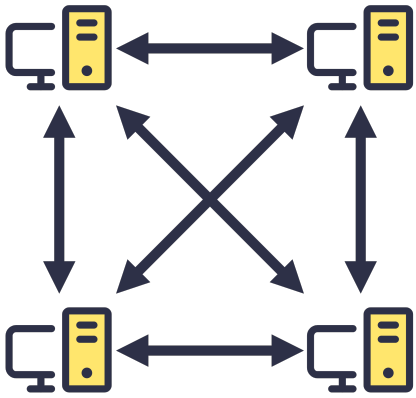
3.2.1 Peer2Peer řešení

P2P[3] architektura nespoléhá na žádné prostředníky a komunikace probíhá přímo mezi klienty. Pro zahájení komunikace mezi klienty existuje několik možností. Jednou z nich je využití distribuované hašovací tabulky (zkr. DHT). Ta obsahuje informace, které lze použít k navázání spojení s jednotlivými klienty. [6] Další způsob, jak navázat P2P komunikaci, je využití WIFI nebo Bluetooth protokolu. V případě potřeby lze k zahájení P2P spojení také využít centrální server. [4]

Kromě výhody v podobě nezávislosti na třetí straně během komunikace sebou přináší P2P architektura i řadu úskalí. Zjevnou nevýhodou je fakt, že pro odeslání a příjem zprávy musí být oba klienti připojeni současně. Nefunguje zde mechanismus zpožděného doručení, který by zajistil, že příjemce dostane zprávu, i když je odesílatel odpojen od sítě. Je také obtížné implementovat funkce jako mazání doručených zpráv, protože není možné ověřit jejich smazání druhou stranou. Díky nutnosti udržování spojení mezi klienty může

[3] **P2P** je zkratka slovního spojení Peer2Peer („rovný s rovným“).

u mobilních zařízení docházet také k rychlejšímu vybíjení akumulátoru. Monitorováním přímých spojení lze také zjistit, kteří dva klienti spolu komunikují. U centralizované služby všichni klienti komunikují s centrálním serverem a na první pohled nelze říci, kteří dva uživatelé vedou konverzaci. K řešení tohoto problému v P2P architektuře je možné využít služby jako VPN[4] či Tor[5]. [4]



Obrázek 2: Grafická reprezentace P2P služby, Zdroj: [autor]

Mezi aktivně vyvíjené služby využívající P2P architekturu patří např. aplikace Briar. [7] Ta umožňuje komunikaci v reálném čase a je určena pro mobilní zařízení. Zajímavostí je, že Briar pro zasílání zpráv mezi klienty dokáže využít nejen internetové připojení ale i Bluetooth či WiFi. [7] Za zmínku stojí také sociální síť Scuttlebutt [8] či dnes již nevyvíjená aplikace BitTorrent Bleep. [9]

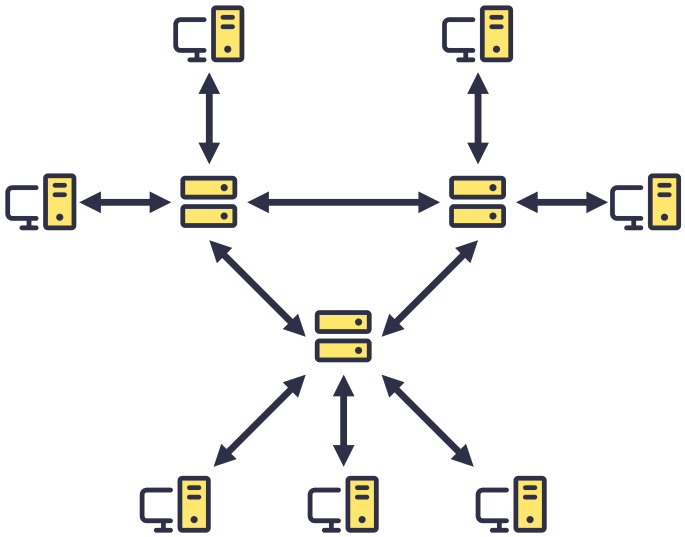
3.2.2 Federovaná řešení

Druhou zde představenou decentralizovanou architekturou bude federovaná architektura. Jde o kombinaci mezi centralizovanou a P2P architekturou. Oproti P2P službám zde komunikace nemůže probíhat bez serverů. Na druhou stranu ale tato komunikace není závislá na jednom centrálním serveru. Většinou se zde nachází celá síť serverů, které jsou v ideálním případě spravovány rozdílnými entitami. Federovaná architektura může být implementována různými způsoby. Je např. možné stanovit, že klient bude pro veškerou komunikaci v rámci služby využívat pouze svůj domácí server[6]. Tento přístup je vyobrazen na obrázku č. 3. Je však také možné federovanou službu navrhnout tak, že klient pro interakci se službou bude využívat několik serverů. Základem federovaných služeb tak bývá protokol specifikující fungování dané služby.

[4] **VPN** představuje bezpečnou soukromou síť provozovanou na veřejném připojení.
[5] **TOR** představuje protokol pro anonymní komunikaci na veřejném připojení.
[6] **Domácí server** je takový server, který si uživatel zvolí jako výchozí pro komunikaci ve federované službě.

Největší výhodou federovaných služeb je, že uživatel jednoho domácího serveru může komunikovat nejen s uživateli na svém serveru, ale také s jakýmkoliv dalším uživatelem, který se nachází na serveru připojeném k síti. Každý uživatel může být provozovatelem svého domácího serveru, což mu umožňuje mít dohled nad svými daty. Díky jasnému stanovení komunikačních protokolů je možné vytvářet implementace klientů či serverů za pomoci různých vývojových platforem. [4]

Stejně jako předchozí popisované architektury má i federovaná své nevýhody. Pokud uživatel neprovozuje svůj vlastní domovský server, ale využívá server třetí strany, tak je nucen podobně jako u centralizované architektury věřit jeho provozovateli. Díky tomu, že do sítě služby může svůj server připojit kdokoliv, tak je zde také riziko rozesílání spamu[7]. Testování a adoptování aktualizací ve federované službě může představovat problém, protože ne každý provozovatel serveru bude ochoten aktualizovat svůj server hned po vydání nové verze služby. V krajním případě může provozovatel serveru aktualizaci odmítnout úplně. [4]



Obrázek 3: Grafická reprezentace federované služby, Zdroj: [autor]

Nejznámějším zástupcem federované architektury je email. Interakce s emailovými službami je prováděna pomocí protokolů SMTP, IMAP, POP3 a v budoucnu možná také pomocí JMAP. [10] Existují různé implementace emailových serverů a klientů. Příkladem federované služby pro komunikaci v reálném čase je síť Matrix. [11]

I přes své nevýhody představuje určitá forma federované architektury dobrý kompromis mezi centralizovanou a P2P architekturou. Z tohoto důvodu bude také využita při tvorbě aplikace popsané v této práci.

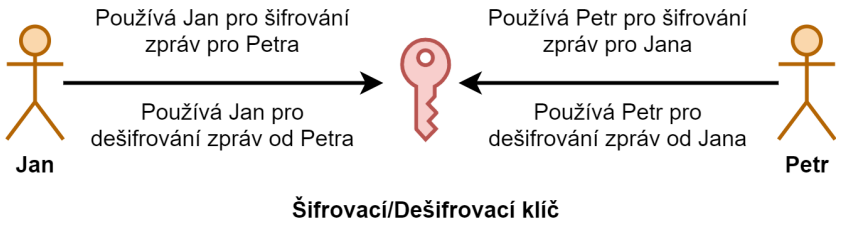
[7] **SPAM** je označení pro nevyžádanou zprávu.

4 Šifrování a hašování

Šifrování zpráv představuje důležitou součást dnešních komunikačních aplikací. Bez šifrování jsou zprávy celou dobu přenosu po síti volně čitelné. [12] V případě, že by někdo komunikaci na síti odposlouchával, tak má přístup k obsahu zpráv, a to včetně případných citlivých dat. Je proto na místě, aby komunikační služby jednotlivé zprávy před odesláním řádně zašifrovaly a rozšifrovaly je až na cílovém zařízení příjemce. Takto bude zajištěno, že i při případném odposlechu komunikace nedojde k úniku citlivých dat. Algoritmy pro šifrování se dělí na symetrické a asymetrické. [13]

4.1 Symetrické šifrování

Symetrické šifrování spočívá v použití stejného klíče pro šifrování i dešifrování zpráv. Tento klíč musí být tajný a mimo koncové uživatele ho nikdo jiný nesmí znát. Velkou výhodou algoritmů pro symetrické šifrování je jejich rychlost. Nevýhodou je nutnost předání šifrovacího/dešifrovacího klíče pomocí nějakého dalšího bezpečného kanálu. [14]



Obrázek 4: Princip symetrického šifrování, Zdroj: [autor]

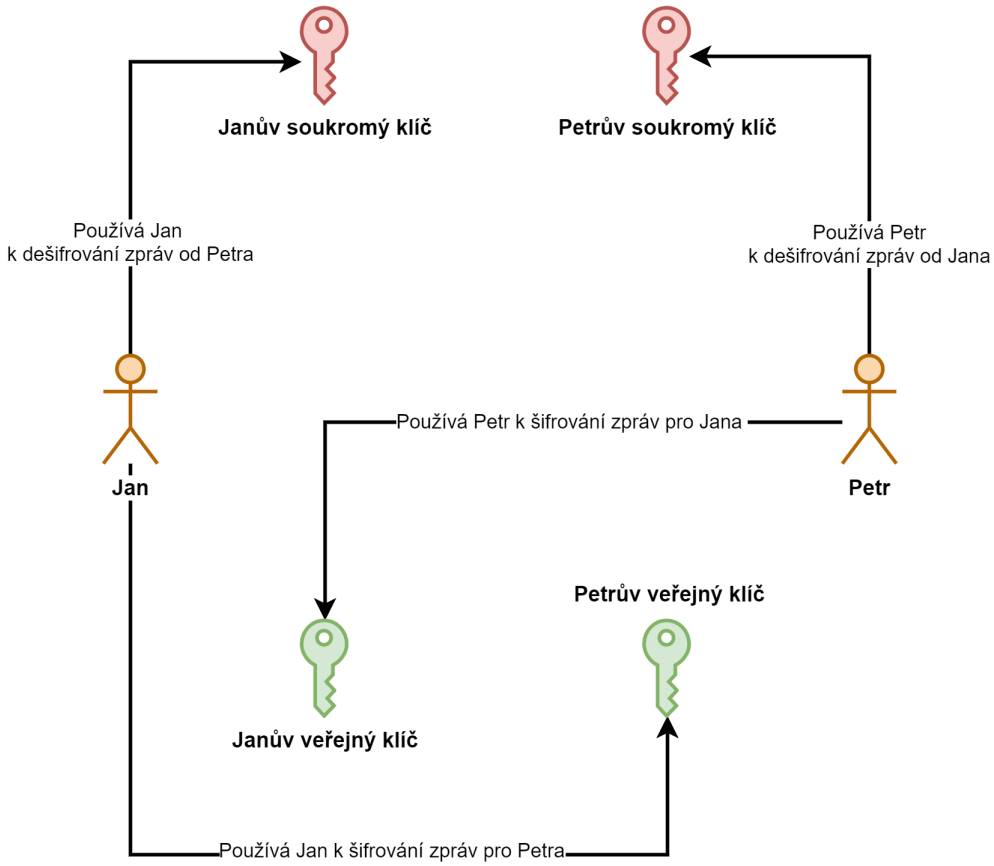
Symetrické šifrovací algoritmy se dělí na **blokové** [15] a **proudové** [16]. Bloková šifra funguje tak, že se algoritmu předá n bitů textu a n bitů šifrovacího klíče. Tento algoritmus pak vytvoří zašifrovaný text o délce n bitů. [15] Mezi blokové šifry patří např. **DES**, [17] **AES** [17] a **Blowfish** [18]. Při použití proudových šifer je zpráva šifrována po jednotlivých bitech či bajtech. [16] Nad každým bitem či bajtem je obvykle provedena operace XOR.

AES (Advanced Encryption Standard) je symetrická bloková šifra s blokem o velikost 128 bitů. Šifrovací klíč může mít velikosti 128, 192 nebo 256 bitů. Vstup o velikosti 128 bitů je uspořádán do matice o velikosti 4×4, kde každou hodnotu představuje jeden bajt. Nad touto maticí jsou následně prováděny operace, které povedou k zašifrování vstupu. Mezi tyto operace patří XOR s šifrovacím klíčem, substituce[8] a rotace[9]. [19] Proces využití operací je nutné několikrát opakovat v závislosti na velikosti šifrovacího klíče. [20]

[8] **Substituci** definujeme jako operaci nahrazení prvku jiným podle předem daných pravidel.
[9] **Rotací** nazýváme operaci posunu prvků v dané množině.

4.2 Asymetrické šifrování

Principem asymetrického šifrování je využití páru soukromého a veřejného klíče pro každého z účastníků komunikace. Soukromý klíč existuje pouze na straně daného uživatele a nesmí být nikde zveřejněn. Naopak veřejný klíč se sdílí veřejně druhému uživateli. [21]



Obrázek 5: Princip asymetrického šifrování, Zdroj: [autor]

Šifrování je zahájeno volbou asymetrického šifrovacího algoritmu a vygenerováním páru soukromého a veřejného klíče obou účastníků konverzace. Veřejné klíče si účastníci vzájemně vymění. Po dokončení výměny je vše připraveno a šifrovaná komunikace může být zahájena. Pro zašifrovaní zprávy vezme uživatel veřejný klíč svého protějšku, za pomoci domluveného algoritmu provede šifrování a následně doručí zprávu druhému uživateli, který vezme svůj soukromý klíč a pomocí domluveného algoritmu zprávu dešifruje. Celý proces je pak možné provádět obousměrně.

Algoritmy pro asymetrické šifrování by měly využívat tzv. jednocestných funkcí. Jedná se o matematické funkce, u kterých je jednoduché získat funkční hodnotu, ale velmi obtížné až téměř nemožné z funkční hodnoty odhadovat vstupní hodnotu. K nejdůležitějším asymetrickým šifrovacím algoritmům patří **RSA** [18] a algoritmy využívající eliptických křivek [22].

Pří využití algoritmu RSA probíhá generování veřejného a soukromého klíče následovně. Uživatel si nejprve zvolí dvě prvočísla p, q a vypočítá n, kde $n = pq$. Dalším krokem je získání hodnoty Eulerovy funkce $\phi(pq) = (p-1)(q-1)$. Poté se zvolí číslo e tak, aby jediným společným dělitelem $\phi(pq)$ a e bylo číslo 1. V praxi se často volí $e = 2^{16} + 1 = 65537$. Dále je nutné zjistit hodnotu výrazu $de \equiv 1(\text{mod } \phi(n))$. Číslo d pak představuje soukromý klíč. Veřejný klíč se skládá z částí n a e. Šifrování zprávy je zahájeno převedením textové zprávy na číslo m (např. pomocí ASCII[10] tabulky). Je důležité, aby platilo, že $m < n$. Výpočtem $c \equiv m^e(\text{ mod } n)$ získáme zašifrovanou zprávu. Tu je nyní bezpečné odeslat protějšku, který ji dešifruje za pomoci výrazu $c^d \equiv m(\text{mod } n)$ a získá tak původní číselnou podobnu zprávy m. [23]

4.3 Hašování

Jak již bylo nastíněno, tak jednocestné funkce jsou takové funkce, kde je výpočetně jednoduché získat funkční hodnotu, ale zároveň velmi obtížné získat hodnotu původní. Tyto funkce dělíme na **funkce s pevnou délkou výstupu** a **funkce s proměnlivou délkou výstupu**. [24] Při použití funkce s pevnou délkou výstupu je bitová délka výstupu stejně dlouhá jako bitová délka vstupu. Naopak délka výstupu funkce s proměnlivou délkou výstupu není závislá na bitové délce vstupu. Funkce s proměnlivou délkou výstupu dále dělíme na **kompresní** či **expanzivní** podle toho, zda je délka výstupu kratší či delší než délka vstupu. [25]

Jednocestné funkce kromě šifrování nachází využití i při úlohách zvaných hašování. Hlavním úkolem hašovací funkce je vytváření hašovacího kódu, který má konstantní délku. Mezi známé hašovací funkce patří např. **SHA** (Secure Hash Algorithm) [26], **MD5** [25], **WHIROPOOL** [27] a **TIGER** [27]. Hašovací funkce lze využít např. pro digitální podpis, generování náhodných hodnot či porovnání otisků dvou souborů.

[10] **ASCII** představuje kódovací tabulku, která znakům anglické abecedy přiřazuje číselnou hodnotu.

5 Vývoj mobilních aplikací

Vývoj mobilních aplikací má oproti tvorbě desktopového a serverového softwaru určitá specifika. Jako příklady lze uvést omezenou konektivitu a závislost na akumulátoru. Existuje jich však mnohem více. Z tohoto důvodu je potřeba dobře znát jednotlivé mobilní platformy a také to, jak se k daným specifikům staví.

5.1 Mobilní operační systémy

Od uvedení prvních smartphonů na trh vzniklo a zaniklo již velké množství mobilních operačních systémů. Některé z nich se podobaly operačním systémům známým z osobních počítačů, jiné zase naopak volily nekonvenční způsoby ovládání. V současné době však většinu podílu na trhu s mobilními zařízeními drží dva operační systémy. Jsou jimi Android a iOS. [28]

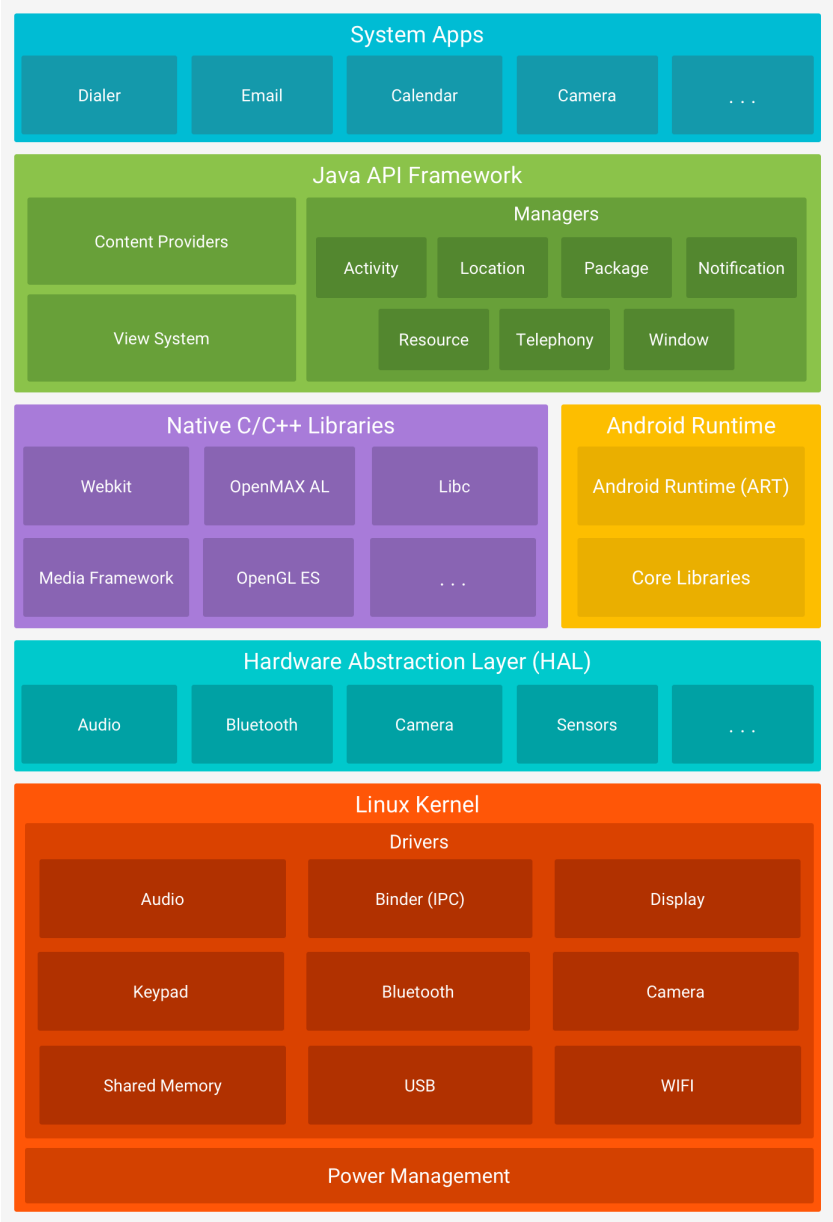
5.1.1 Android

Android je operační systém založený na Linuxu a přizpůsobený pro široké množství rozličných zařízení. [29] Jeho základ tvoří **Linuxové jádro**, které poskytuje nízkoúrovňové služby vyšším vrstvám systému. Běhovému prostředí aplikací např. umožňuje pracovat s vlákny či operační pamětí. Velkou výhodou Linuxového jádra je jeho bezpečnost a široká podpora různých hardwarových komponent. [29]

Vrstva pro odstínění od Hardwaru (HAL) poskytuje vysokoúrovňovému frameworku Java API rozhraní pro práci s jednotlivými hardwarovými komponentami zařízení. HAL se skládá z množství modulů a knihoven, které implementují rozhraní konkrétních zařízení jako fotoaparát či Bluetooth modul. Když je v aplikaci pomocí Java API zavolána nějaká funkce zařízení, tak Android použije příslušnou knihovnu HAL dané komponenty. [29]

Operační systém Android do verze 5.0 využíval pro běh aplikací běhové prostředí Dalvik. Od této verze je každá aplikace spuštěna jako vlastní proces v novém běhovém prostředí **Android Runtime (ART)**. Toto prostředí umožňuje aplikacím využívat např. *„garbage collector“* pro automatickou správu paměti. [29]

Některé služby systému jako HAL či ART vyžadují pro své fungování nízkoúrovňové nativní knihovny. Tyto knihovny mohou využívat i vývojáři aplikací v podobě tzv. **Native Development Kit (NDK)**. Díky NDK je možné vyvíjet software pomocí nízkoúrovňových jazyků jako jsou C a C++. [29]



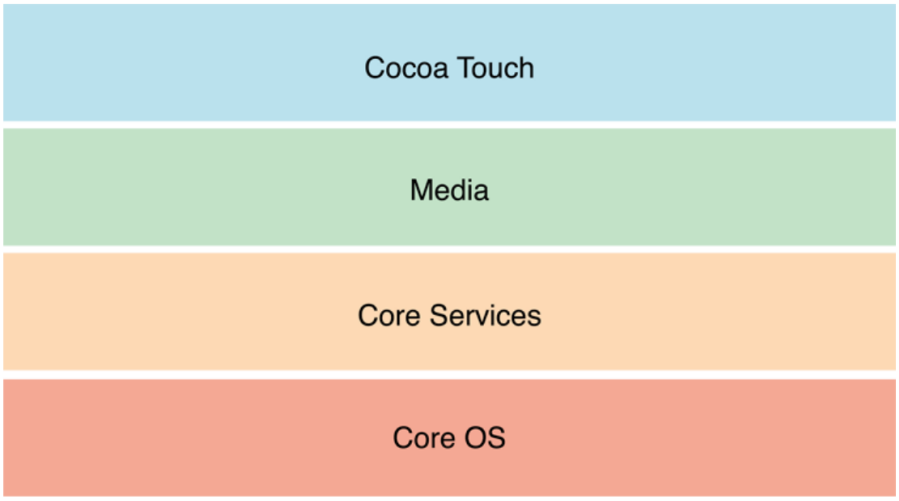
Obrázek 6: Architektura operačního systému Android, Zdroj: [29]

Veškeré součásti operačního systému Android jsou pro vývojáře dostupné skrze vysokoúrovňové Java API. Příkladem takto poskytovaných součástí mohou být knihovny pro tvorbu uživatelského rozhraní, interakci s komponentami zařízení, vytváření oznámení a správu životního cyklu aplikace. [29]

Systém Android obsahuje sadu předinstalovaných systémových aplikací, které uživateli umožňují využívání základních funkcí zařízení. Jedná se např. o software pro zasílání textových SMS zpráv, telefonování či prohlížení webu. Výrobci samotných zařízení pak často přidávají ještě další vlastní aplikace jako alternativu k těm systémovým. Zároveň má uživatel možnost do zařízení instalovat software podle vlastní potřeby. [29]

5.1.2 iOS

Architektura operačního systému iOS může na první pohled působit odlišně od architektury systému Android, ale ve skutečnosti jsou si oba systémy velmi podobné. Rozdíl je především v tom, že na iOS jsou funkce jednotlivých vrstev daleko přímočařejší. Vrstvy jsou hierarchicky uspořádané a žádné dvě nestojí na stejné úrovni. Díky tomu také platí pravidlo, že každá vyšší vrstva využívá tu v hierarchii nižší. [30]



Obrázek 7: Architektura operačního systému iOS, Zdroj: [30]

Cílem vrstvy **Cocoa Touch** je poskytnou vývojářům rozhraní pro interakci se součástmi zařízení jako jsou např. kamera, kontakty a oznámení. Zároveň umožňuje přistupovat k nativní knihovně pro tvorbu grafického uživatelského rozhraní. [30]

Vrstva s názvem **Media** umožňuje využívání grafických API (OpenGL a Metal) a zvukových API (OpenAL a Core Audio). Zprostředkovává též přístup k funkci AirPlay umožňující streamování zvukových a vizuálních médii do dalších zařízení. [30]

Core services je vrstva představující abstrakci nad základními komponentami systému. Umožňuje např. využití síťových služeb, šifrování nebo správu embedded databáze SQLite. [30]

Nejnižší vrstvou systému iOS je **Core OS**. Tato vrstva operuje přímo nad jádrem operačního systému a slouží jako prostředník mezi ním a vyššími vrstvami. Při vývoji uživatelských aplikací vývojáři většinou využívají služeb vyšších vrstev a k přímému použití vrstvy **Core OS** se uchylují jen v nejnutnějších případech. [30]

5.1.3 Další mobilní operační systémy

Kromě výše uvedených operačních systémů existují i další s menším podílem na trhu. Tizen od společnosti Samsung je v současné době využíván především pro nositelnou elektroniku a televizory. [31] SailfishOS vyvíjený společností Jolla usiluje o to stát se přímou alternativou k zavedeným systémům např. pomocí možnosti spouštět aplikace původně vytvořené pro Android. [32] Zajímavé řešení představuje také KaiOS, jehož výhodou jsou nízké požadavky na hardwarové prostředky. Díky tomu je možné systém nasadit i na ta nejlevnější mobilní zařízení. [33] Jedná se o přímého pokračovatele systému FirefoxOS, jehož vývoj byl již v minulosti ukončen. [34]

5.2 Přístupy k vývoji mobilních aplikací

K vývoji mobilních aplikací je možné přistupovat řadou různých způsobů. Ty se liší použitými technologiemi, požadavky na zařízení, ale také obtížností vývoje. Obsah této podkapitoly bude věnován popisu jednotlivých přístupů a porovnání jejich výhod či nevýhod.

5.2.1 Nativní aplikace

Nativní aplikace je taková aplikace, která je vytvořena pro konkrétní operační systém za pomocí nástrojů poskytnutých jeho výrobcem. Výhodou takto vyvinutých aplikací je vysoká rychlost, přístup ke všem prostředkům zařízení a oficiální podpora knihoven používaných při vývoji. Nevýhodou je závislost na dané platformě. Uživatelské rozhraní takovéto aplikace je pak tvořeno pomocí výrobcem poskytnutých grafických knihoven.

Oficiálně podporovaným způsobem vývoje aplikací na operační systému Android je využití programovacích jazyků postavených nad JVM (Java Virtual Machine). [29] Jde o jazyky Java [29] a Kotlin. [35] Pro vývoj nativních aplikací na operační systému iOS je možné využít jazyky Objective-C [36] či Swift [37].

5.2.2 Progresivní webové aplikace

Progresivní webové aplikace razí oproti těm nativním zcela odlišný přístup. Jde o webové aplikace, které se z pohledu uživatele tváří jako dobře integrované s operačním systémem. Mohou např. fungovat offline a mají svou ikonu na domovské obrazovce zařízení. Uživatelské prostředí těchto aplikací je vytvářeno pomocí technologii HTML, CSS a Javascript. Velkou výhodou je, že jsou multiplatformní a zároveň k jejich provozu stačí

webový prohlížeč. Nevýhodu představuje fakt, že tyto aplikace nemají přístup ke všem součástem operačního systému. Díky tomu stále existují příklady softwaru, který je nutné tvořit jako nativní.

5.2.3 Hybridní aplikace

Aplikace vytvořené jako hybridní se snaží kombinovat výhody obou výše popsaných způsobů. Kromě kompletního přístupu k operačnímu systému, jako mají nativní aplikace, je vývoj těchto aplikaci jednodušší. To je způsobeno tím, že většina kódu vyvíjené aplikace není závislá na konkrétním operačním systému, ale pouze určité její části jsou přizpůsobovány pro danou platformu.

Tvorba uživatelského rozhraní se u různých hybridních frameworků značně liší. Některé pro vykreslování ovládacích prvků využívají nativní knihovny. Sem lze zařadit frameworky React Native [38] a Xamarin.Forms [39]. Apache Cordova naopak volí webové technologie. [40] Framework Flutter zase využívá možnosti vykreslování uživatelského rozhraní pomocí grafické knihovny Skia. [41]

V rámci práce vytvářená ukázková aplikace bude naprogramována pouze pro operační systém Android. Bude však vytvořena jako hybridní, aby ji bylo možné v případě potřeby jednoduše převést i na další platformy.

6 Představení použitých technologii

Důležitým bodem při návrhu jakékoliv aplikace je volba technologií, které budou k jejímu vývoji použity. Vzhledem k tomu, že se aplikace pro decentralizovanou komunikaci bude skládat ze serverové a klientské aplikace, tak je nutné zvolit technologie pro obě tyto části.

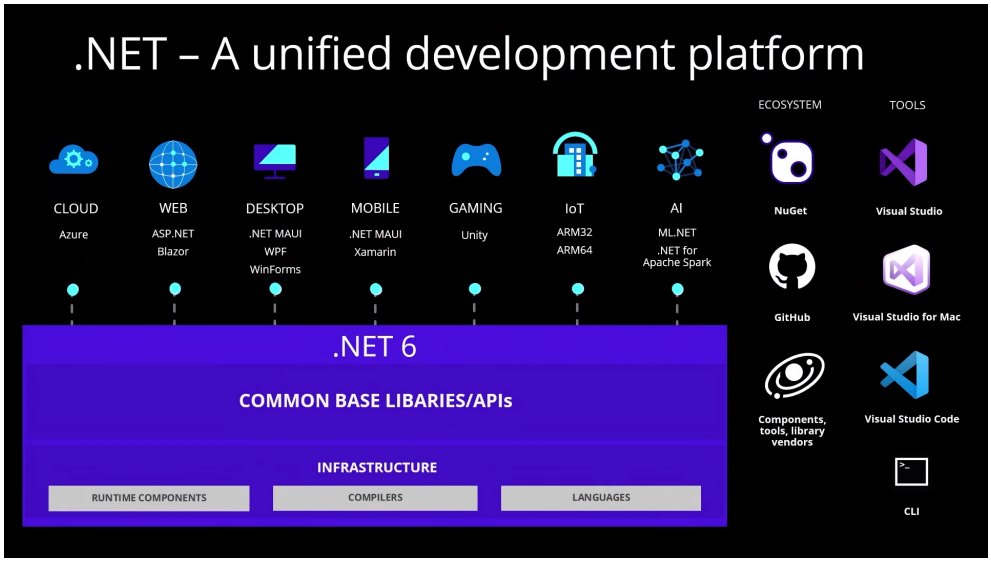
6.1 .NET

.NET je vývojová platforma spravovaná společností Microsoft a nezávislým konsorciem .NET Foundation. Základním kamenem této platformy je virtuální stroj. Stojí tak mezi nízkoúrovňovými programovacími platformami (např. C++ a Rust) a vysokoúrovňovými (např. Python a Javascript). U nízkoúrovňových jazyků je nutné výsledný program kompilovat pro každý operační systém a procesorovou architekturu zvlášť. U vysokoúrovňových jazyků je zase výsledný program šířen jako zdrojový kód a pro každou platformu musí existovat interpret, který bude tento kód za běhu překládat. Jazyky s virtuálním strojem jsou potom určitou kombinací obou přístupů. Výsledný program je šířen v podobě tzv. mezikódu, který je platformě nezávislý. Tento mezikód je pak pomocí virtuálního stroje za běhu překládán do nativního kódu zvolené platformy. Překlad tohoto mezikódu je daleko rychlejší než překlad zdrojového kódu u interpretovaných jazyků. Software vytvořený za pomoci platformy s virtuálním strojem běží jen o něco málo pomaleji, než kdyby byl vytvořen v nízkoúrovňovém jazyce, ale zároveň je jeho vývoj podobně pohodlný jako v případě vysokoúrovňových jazyků. [42]

Původní verze .NET frameworku byly vázané pouze na operační systém Microsoft Windows. Od vydání .NET Core 1.0 je framework multiplatformní a lze ho provozovat na celé řadě operačních systémů (např. Windows, Linux, macOS, Android a iOS), ale také procesorových architektur (x86, x64 a ARM). [42]

Microsoft na platformě .NET oficiálně podporuje tři programovací jazyky. Jsou jimi C#, F# a Visual Basic. Jazyky C# a F# jsou aktivně vyvíjeny, kdežto Visual Basic již ne. C# je primárně navržen jako objektově orientovaný jazyk, avšak v několika posledních verzích dostává i vylepšení, která by se dala zařadit spíše do funkcionálního programování. F# na to jde obráceně a jedná se o primárně funkcionální jazyk. Nicméně pro interoperabilitu s ostatními jazyky .NETu podporuje i objektové paradigma. [42]

.NET lze využít k tvorbě celé řady typů aplikací. Jako několik příkladů lze uvést grafické aplikace pro osobní počítače a mobilní zařízení, serverové služby, videohry a embedded aplikace. [42]



Obrázek 8: Přehled vývojové platformy .NET, Zdroj: [43]

6.1.1 ASP.NET Core

ASP.NET Core je oficiálně podporovaný framework pro vývoj webových služeb. Využívá návrhového vzoru MVC, který bude podrobněji přiblížen později v této práci. Zjednodušeně se jedná o způsob návrhu softwaru, který umožňuje vysokou modularitu jeho komponent. Díky tomu pak není obtížné aplikaci rozvíjet i v případě, že naroste do velkých rozměrů. Tento framework je určen k tvorbě kompletních řešení na straně serveru, ale také k tvorbě strojově zpracovatelných API, které jsou konzumovány klientskými webovými či mobilními aplikacemi. Díky úsilí, které společnost Microsoft do vývoje ASP.NET Core vkládá, patří tato technologie ke špičce mezi konkurencí v rychlosti a množství zpracovaných dotazů za jednotku času. [44]

6.1.2 Xamarin.Forms

Podobně jako v případě ASP.NET Core pro tvorbu webových služeb má platforma .NET také oficiálně podporovaný framework pro tvorbu mobilních aplikací. Ten se nazývá Xamarin.Forms a výsledkem jeho použití jsou hybridní multiplatformní mobilní aplikace. Uživatelské rozhraní vytvořené v rámci tohoto frameworku využívá nativních knihoven daných platforem. Samotné definování jednotlivých částí prostředí pak probíhá pomocí značkovacího jazyka XAML, který není závislý na konkrétním operačním systému. Logiku Xamarin.Forms aplikace lze vytvářet pomocí jazyků C# a F#. V průběhu roku 2022 bude tento framework nahrazen novým s názvem .NET MAUI [45]. Ten však ze Xamarin.Forms přímo vychází. [46]

6.2 Databázové systémy

Důležitá je též volba technologií, které budou použité k ukládání perzistentních dat. Tento výběr je nutné provést jak na klientské, tak na serverové části aplikace, protože každá z nich si bude sama spravovat svá data.

6.2.1 PostgreSQL

PostgreSQL je open-source relační databáze. Množstvím funkcí a stabilitou konkuruje databázím jako jsou Microsoft SQL server, Oracle DB či IBM DB2. V různých testech rychlosti se PostgreSQL vyrovná, či dokonce přesahuje většinu nejpoužívanějších open-source a komerčních databází. Nevýhodou PostgreSQL je její velikost, která po nainstalování přesahuje 100 MB. Díky tomu není vhodnou volbou pro desktopové či mobilní aplikace. Pro správu PostgreSQL je možné využít nástroj pro příkazovou řádku **psql** či alternativu s grafickou nadstavbou v podobě **pgAdmin**. PostgreSQL je samozřejmě možné spravovat i pomocí řady dalších nástrojů, které však nejsou oficiálně podporovány. Díky oblíbenosti této databáze existují „databázové drivery“ pro velké množství programovacích jazyků a nástrojů. Jako příklady lze vybrat .NET, Java, PHP, Python, Ruby či LibreOffice. Mezi databázové objekty podporované PostgreSQL kromě základních tabulek patří také např. pohledy, funkce, sekvence, triggery a rozšíření. [47]

6.2.2 LiteDB

LiteDB je embedded NoSQL databázový systém pro platformu .NET, který nevyžaduje běžící databázový server. Celá databáze je uložena v jediném souboru ve formátu BSON. Díky své jednoduchosti je vhodnou volbou při vytváření prototypů aplikací. Alternativou k LiteDB může být další embedded databáze s názvem SQLite, která je však plnohodnotnou relační databází. [48]

6.3 Podpůrné technologie

Kromě technologií, které jsou pro vývoj decentralizované komunikační aplikace nutné, je vhodné použít i další podpůrné nástroje. Ty mohou např. usnadnit vývoj nebo pomoci s organizací souborů projektu.

[10] **NoSQL** databáze při svém fungování nespoléhá na principy relačních databází.

6.3.1 Git

Pro verzování kódu aplikace bude využit systém Git. Ten funguje na principu tzv. commitů, které vývojář vytváří po provedení změn ve zdrojovém kódu projektu. Commit si lze představit jako otisk stavu souborů projektu v danou chvíli. V případě, že vývojář po provedení commitu udělá v kódu další změny a následně zjistí, že jsou tyto změny nevhodné, tak se může jednoduše vrátit ke stavu kódu před provedením těchto změn. Výhodou Gitu je také možnost větvení, která umožňuje provádět změny kódu bez toho, aby byla ovlivněna hlavní větev. Po dokončení všech prací v dané větvi může být její obsah sloučen zpět do větve hlavní. Git je také možné kombinovat s dalším softwarem. Příkladem může být automatická kompilace kódu[11] a nasazení na produkční server[12] po provedení commitu do hlavní větve. Tento postup se nazývá „Continuous Integration/Delivery“. [49]

6.3.2 Docker

Kontejnerizaci aplikací lze využít pro nasazení vyvíjených aplikací do produkčního prostředí, ale také pro provozování vývojových nástrojů na počítači vývojáře. Díky Kontejnerizaci je např. možné lokálně spustit obraz vybrané databáze bez nutnosti instalace dalšího softwaru. Jednou z nejoblíbenějších kontejnerizačních platforem je **Docker**, ale existují i alternativy jako např. **Podman**. [50]

[11] **Kompilace kódu** je proces vytvoření spustitelného souboru aplikace.
[12] **Produkční server** je server, pomocí kterého přistupují k aplikaci koncoví uživatelé.

7 Analýza a návrh aplikace

V předchozích částech této práce byly přiblíženy architektury komunikačních služeb, šifrování, vývoj mobilních aplikací a technologie, které je možné při vývoji použít. V této kapitole bude navrhnuta aplikace pro decentralizovanou komunikaci, která popsaných znalostí využívá. Budou specifikovány funkční a non-funkční požadavky, programové rozhraní serverové aplikace a způsob tvorby uživatelského rozhraní mobilní aplikace.

7.1 Požadavky aplikace

Požadavky na aplikaci můžeme dělit do dvou skupin na požadavky funkční a non-funkční. Funkční požadavky definují funkce aplikace, které může uživatel využívat. V případě bankovní aplikace může jít např. o zobrazení zůstatku na účtu. Non-funkční požadavky naopak vyjadřují požadavky, které přímo nesouvisí s funkcionalitou aplikace. To si lze opět představit na bankovní aplikaci, kdy může být např. požadováno dvoufaktorové ověření[13] při potvrzování transakcí.

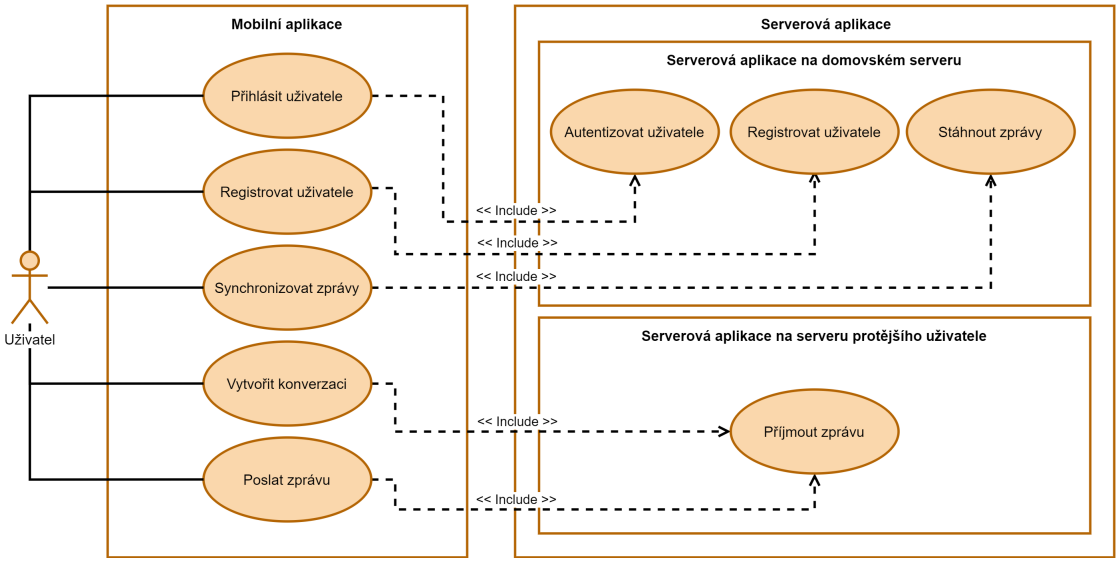
7.1.1 Funkční požadavky

Aplikace pro decentralizovanou komunikaci umožní uživateli provádění úkonů spojených s autentizací, autorizací a správou konverzací a zpráv. Funkční požadavky na aplikaci jsou rozděleny do dvou kategorii. Jsou jimi **požadavky na mobilní aplikaci** a **požadavky na serverovou aplikaci**.

Uživatel bude přímo interagovat s klientskou aplikací, která bude implementovat případy užití související s autentizací a správou konverzací. Při registraci si uživatel zvolí server, který bude sloužit jako domovský. Po dokončení registrace dostane své autentizační údaje v podobě id a šifrovacího klíče. Tento klíč bude využívat pro komunikaci s domovským serverem. Pro přihlášení je požadována volba domovského serveru a identifikační údaje, které uživatel obdržel při registraci. Případ užití pro synchronizaci zpráv řeší stažení všech příchozí zpráv od ostatních uživatelů služby. Vytvoření konverzace je případ užití, který po zadání identifikačních údajů a serveru protějšího uživatele zajistí zahájení šifrované komunikace. K tomu využívá případu užití „Přijmout zprávu“ na instanci serverové aplikace protějšího uživatele. Případ užití „Poslat zprávu“ umožňuje uživateli odeslat zprávu v rámci konverzace svému protějšku. Opět k tomu využívá případ užití „Přijmout zprávu“ na serverové části aplikace.

[13] **Dvoufaktorové ověření** vyžaduje kromě hesla ještě druhé ověření např. pomocí SMS či otisku prstu.

Všechny případy užití klienta na pozadí využívají serverové části aplikace. Tam je možné rozdělit případy užití do dvou skupin podle toho, zda klient komunikuje se svým domovským serverem či se serverem protějšího uživatele. Případy užití týkající se registrace, přihlášení a stažení zpráv využívají domovský server uživatele. Naopak případ užití umožňující přijetí zprávy od uživatele využívá serverovou aplikaci protějšího uživatele. Je důležité podotknout, že pokud budou oba uživatelé využívat stejný domovský server, tak je pro obě skupiny případů užití využíván stejný server.

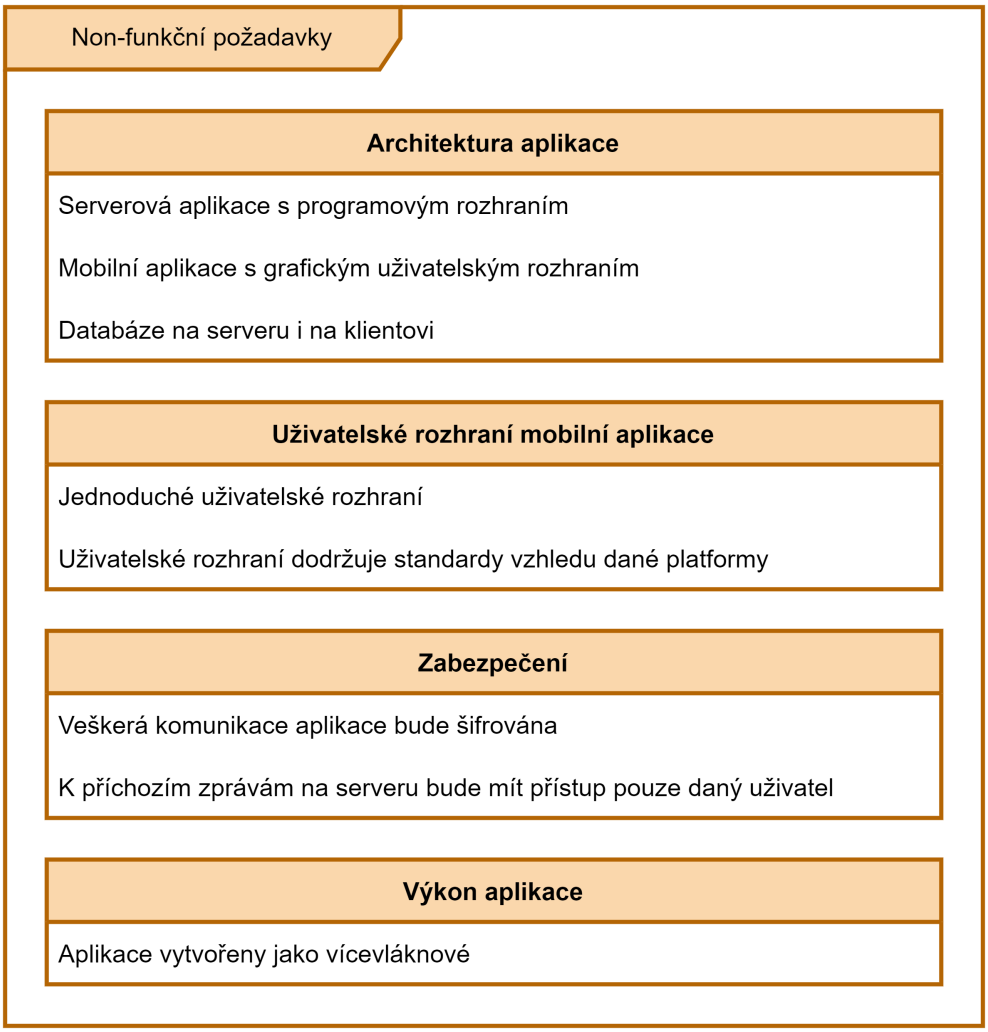


Obrázek 9: Diagram funkčních požadavků aplikace, Zdroj: [autor]

7.1.2 Non-funkční požadavky

Kromě funkčních požadavků je nutné definovat i požadavky non-funkční. Ty jsou specifikovány v diagramu zobrazeném na obrázku č. 10. Jednotlivé požadavky jsou rozděleny do kategorii podle toho, jaké části aplikace se dotýkají.

Architektura aplikace bude navržena tak, aby serverová část poskytovala programové rozhraní jednoduše konzumovatelné dalšími aplikacemi. Mobilní klient poskytne jednoduché grafické uživatelské rozhraní respektující designový jazyk dané platformy. Server i klient budou mít svou vlastní databázi. Důležitou součástí bude zabezpečení, kde je nutné zajistit šifrování komunikace. V rámci serverové části dojde k implementaci autorizace před stažením příchozích zpráv ze serveru. Při vývoji bude brán na zřetel non-funkční požadavek požadující vícevláknovost.

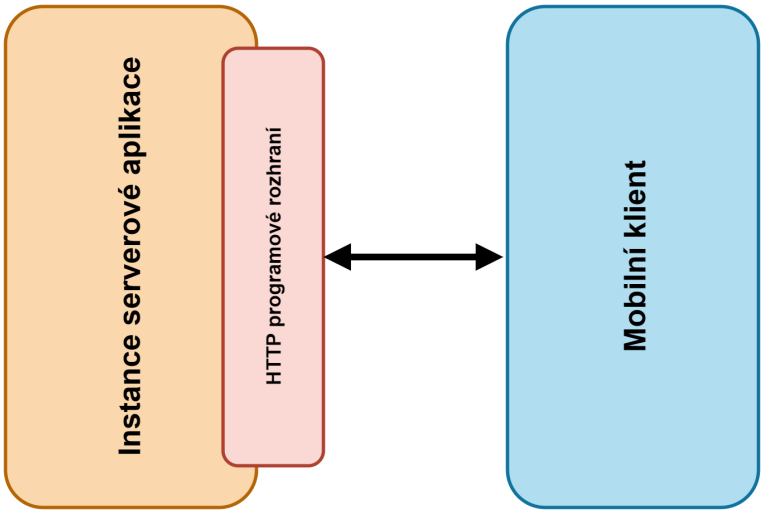


Obrázek 10: Diagram non-funkčních požadavků aplikace, Zdroj: [autor]

7.2 Architektura aplikace

Aplikace pro decentralizovanou komunikaci bude rozdělena na klientskou a serverovou část. Jako decentralizovaná architektura bude využita federovaná architektura, která byla podrobně přiblížena ve 3. kapitole. Serverová část bude obsahovat programové rozhraní využívající protokolu HTTP. Klientská aplikace bude tvořena jako mobilní aplikace pro operační systém Android.

HTTP je protokol aplikační vrstvy síťového modelu ISO/OSI. Tato vrstva využívá služeb nižších vrstev, které pro ni zařídí přenos dat po síti. HTTP využívá procesu „request/response“. Nejdříve klient naváže spojení se serverem ležícím na zvolené adrese. Následně odešle požadavek, který obsahuje HTTP metodu, URL, verzi HTTP protokolu a případně další metadata či tělo požadavku. Server následně odešle odpověď, která obsahuje verzi HTTP protokolu, stavový kód udávající úspěšnost či neúspěšnost požadavku a případná další metadata či tělo odpovědi. [51]



Obrázek 11: Diagram architektury aplikace, Zdroj: [autor]

7.3 Uživatelské rozhraní mobilní aplikace

Material Design je designovým jazykem společnosti Google vyvinutým pro operační systém Android. Není však nutně omezen pouze na tento operační systém. Google ho často využívá např. při tvorbě svých webových aplikací. [52]

Nejdůležitějším prvkem Material Designu je virtuální papír. Jednotlivé části prostředí jsou navrženy tak, aby co nejvíce připomínaly právě skutečný papír a na uživatele tak prostředí působilo přirozeně. Mezi další důležité prvky patří animace, které mají navozovat plynulost při různých přesunech a akcích. Podstatné jsou též stíny, které lze využít ke zvýraznění určitých částí grafického prostředí. Součástí Material Designu je rodina písem Roboto, kterou se Google pokusil navrhnout tak, aby na displejích zařízení působila co nejpřirozeněji. Alternativní rodinou písem v rámci Material Designu je Noto. Ta je např. využita jako výchozí rodina fontů na operačním systému ChromeOS. [52]

Mobilní klient aplikace pro decentralizovanou komunikaci bude využívat a dodržovat pravidla stanovená designovým jazykem Material. Jednotlivé obrazovky budou také vytvořeny s ohledem na non-funkční požadavek jednoduchého ovládání.

8 Popis implementace aplikace

Následující kapitola bude věnována implementaci aplikace pro decentralizovanou komunikaci. Kapitola začne popisem založení jednotlivých projektů aplikace. Dále bude pokračovat objasněním implementace serverové a mobilní části aplikace. Závěr kapitoly pak představí implementaci šifrované komunikace a protokol Diffie-Hellman, který byl v rámci vyvinuté aplikace využit.

8.1 Založení projektů

Proces založení projektu na platformě .NET se skládá z několika důležitých kroků. Prvním z nich je příprava prostředí pro vývoj. Toto prostředí umožní využívání programátorských nástrojů platformy. Dále dojde k založení jednotlivých projektů. Po dokončení procesu vytváření je ještě nutné nově vzniklé projekty vzájemně propojit.

8.1.1 Příprava prostředí

Pro vytvoření projektu je potřeba splnit předpoklad v podobě instalace SDK[14] platformy .NET. Aplikace pro decentralizovanou komunikaci bude implementována v .NET 6.0, která je v době psaní nejaktuálnější stabilní verzí SDK. Aby bylo možné vyvíjet mobilní aplikaci, tak je také nutné nainstalovat vývojové prostředí Visual Studio a s ním nástroje Xamarin pro tvorbu mobilních aplikací na platformě .NET. Tyto nástroje mimo jiné obsahují „Android Emulator“ a „iOS Simulator“, které umožňují testování vyvíjených aplikací bez nutnosti použití fyzických mobilních zařízení. Protože mobilní klient aplikace pro decentralizovanou komunikaci bude vytvářen pouze pro operační systém Android, tak není nutné nastavovat iOS Simulator a stačí pouze vytvořit instanci virtuálního zařízení pomocí Android emulátoru.

8.1.2 Vytvoření a správa projektů

.NET nabízí hned několik způsobů, jak vytvářet a spravovat projekty vyvíjené aplikace. Je možné využít grafického průvodce ve vývojovém prostředí Visual Studio. Ten umožňuje efektivní správu projektů pomocí několika jednoduchých grafických formulářů. Kromě grafického prostředí lze také využít nástrojů pro příkazovou řádku. Ty jsou integrované v samotném .NET SDK a není tak nutná instalace dalších součástí. Zároveň jsou nástroje pro příkazovou řádku dostupné např. i pro operační systém Linux, kdežto gra-

[14] **SDK** představuje ucelenou sadu nástrojů a knihoven, které slouží k vývoji aplikací.

fické prostředí Visual Studio je možné použít pouze na operačních systémech Microsoft Windows a macOS. V této práci bude popsán postup tvorby projektů pomocí nástrojů pro příkazovou řádku, ale podobný postup je možné aplikovat i pomocí grafického prostředí Visual Studia.

Vytvoření nového .NET projektu pomocí nástrojů pro příkazovou řádku je provedeno příkazem „`dotnet new`“. Pro jeho dokončení je nutné zadat výchozí šablonu projektu. Šablona představuje předpřipravený projekt, který umožňuje jednoduché zahájení vývoje aplikace daného typu. Pomocí parametru „`-n`“ je možné specifikovat název projektu. Vytvoření nového projektu je zobrazeno v ukázce zdrojového kódu č. 1.

```
1 dotnet new <sablona-projektu> -n <nazev-projektu>
```

Ukázka kódu 1: Vytvoření nového projektu pomocí .NET CLI, Zdroj: [autor]

Aby bylo možné v projektu využít kód nacházející se v jiném projektu, tak je nutné vytvořit tzv. referenci. To se provede zavoláním příkazu „`dotnet add ... reference`“. Tuto změnu je také možné provést manuální úpravou XML souboru projektu s koncovkou „`.csproj`“. Použití příkazu v terminálu je však daleko pohodlnější. Kompletní přidání reference projektu znázorňuje ukázka zdrojového kódu č. 2.

```
1 dotnet add <projekt> reference <reference>
```

Ukázka kódu 2: Přidání reference projektu pomocí .NET CLI, Zdroj: [autor]

První blok ukázky kódu č. 3 popisuje vytvoření jednotlivých projektů pro serverovou část aplikace. Projekt uvedený na prvním řádku bude sloužit jako implementace serveru. Druhý a třetí řádek definují projekty pro sdílený kód a jeho testování. Zbytek kódu pak ukazuje přidání referencí mezi projekty.

```
1 dotnet new web -n "DiplomaThesis.Server"
2 dotnet new classlib -n "DiplomaThesis.Core"
3 dotnet new mstest -n "DiplomaThesis.Core.Tests"
4
5 dotnet add DiplomaThesis.Server/DiplomaThesis.Server.csproj
6     reference DiplomaThesis.Core/DiplomaThesis.Core.csproj
7
8 dotnet add DiplomaThesis.Core.Tests/DiplomaThesis.Core.Tests.csproj
9     reference DiplomaThesis.Core/DiplomaThesis.Core.csproj
```

Ukázka kódu 3: Vytvoření projektů vyvíjené aplikace, Zdroj: [autor]

Problém nastává při vytváření projektů pro mobilního klienta. Framework Xamarin.Forms je závislý na nástrojích Visual Studia a tak je nutné projekty využívající tento framework vytvářet pomocí grafického průvodce. Tento problém bude vyřešen po vydání frameworku .NET MAUI, který bude vyžadovat pouze standardní součásti .NET SDK. Pro vytvoření nového projektu mobilní aplikace ve Visual Studiu je nutné zvolit šablonu Xamarin.Forms. Následně je potřeba specifikovat název aplikace jako „DiplomaThesis.MobileApp“. V posledním kroku je nezbytné vybrat prázdnou šablonu mobilní aplikace a jako vyvíjenou platformu označit operační systém Android.

8.1.3 Struktura vytvořených projektů

Projekt s názvem „DiplomaThesis.Server“ po vytvoření obsahuje 4 soubory. Nejdůležitější z nich je *Program.cs*. Jeho výchozí obsah je vyobrazen v ukázce kódu č. 4. Zbylé soubory projektu obsahují pouze nastavení. Soubory obsažené v projektech „DiplomaThesis.Core“ a „DiplomaThesis.Core.Test“ nejsou pro vyvíjenou aplikaci důležité a mohou být smazány.

```
1 var builder = WebApplication.CreateBuilder(args);
2 var app = builder.Build();
3
4 app.MapGet("/", () => "Hello World!");
5
6 app.Run();
```

Ukázka kódu 4: Výchozí obsah hlavní třídy serverové aplikace, Zdroj: [autor]

Po založení nové Xamarin.Forms aplikace vzniknou dva projekty. První z nich s názvem „DiplomaThesis.MobileApp.Android“ bude využíván pro části kódu závislé na operačním systému Android. Projekt „DiplomaThesis.MobileApp“ bude zase obsahovat sdílený kód nezávislý na konkrétní platformě.

8.2 Implementace serverové aplikace

V 7. kapitole bylo specifikováno, že serverová aplikace bude obsahovat strojově zpracovatelné aplikační rozhraní využívající protokolu HTTP. Jako formát pro výměnu strukturalizovaných dat mezi klientem a serverem byl zvolen JSON.

8.2.1 MVC

MVC je návrhový vzor pro modularizaci aplikací. Dělí komponenty do tří kategorii. Těmi jsou modely, pohledy a kontroléry. Úkolem modelů je především definovat datové entity v aplikaci. Pohledy představují grafické uživatelské rozhraní aplikace. Díky nim je možné uživateli zobrazit vizuální podobu výsledků jeho dotazu. Umožňují také interakci s aplikací pomocí formulářových prvků (např. tlačítka, textová pole, rozbalovací menu) zabudovaných v uživatelské rozhraní. Posledním prvkem v MVC jsou kontroléry. Ty se starají o zpracování dotazů a logiku aplikace. Kontroléry slouží jako prostředník mezi pohledy a modely. Podle parametrů dotazu uživatele připraví modely, které následně dosadí do pohledů a výsledek pošlou zpět. Často je implementován jeden hlavní kontrolér (tzv. „Front controller“), který přijaté dotazy předává podřízeným kontrolérům na základě parametrů konkrétního dotazu. [53]

Serverová část aplikace bude využívat modifikovanou podobu návrhového vzoru MVC. Změna bude především v tom, že aplikace nebude poskytovat grafické pohledy, ale pouze strojově zpracovatelnou formu modelů.

8.2.2 Databáze

Serverová aplikace bude jako svou databázi využívat PostgreSQL. Dotazy v aplikaci nebudou vytvářené přímo v jazyce SQL[15], ale pomocí abstrakční vrstvy ORM. Entity Framework, který bude v aplikaci použit, je právě jedním ze zástupců ORM na platformě .NET. Umožňuje např. jednoduché mapování objektů na databázové entity či automatickou migraci dat při změně struktury databáze. Pro využití Entity Frameworku je zapotřebí do projektu doinstalovat balíčky uvedené v ukázce kódu č. 5.

```
1 dotnet add package Microsoft.EntityFrameworkCore
2 dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL
```

Ukázka kódu 5: Instalace balíčků pro připojení k PostgreSQL, Zdroj: [autor]

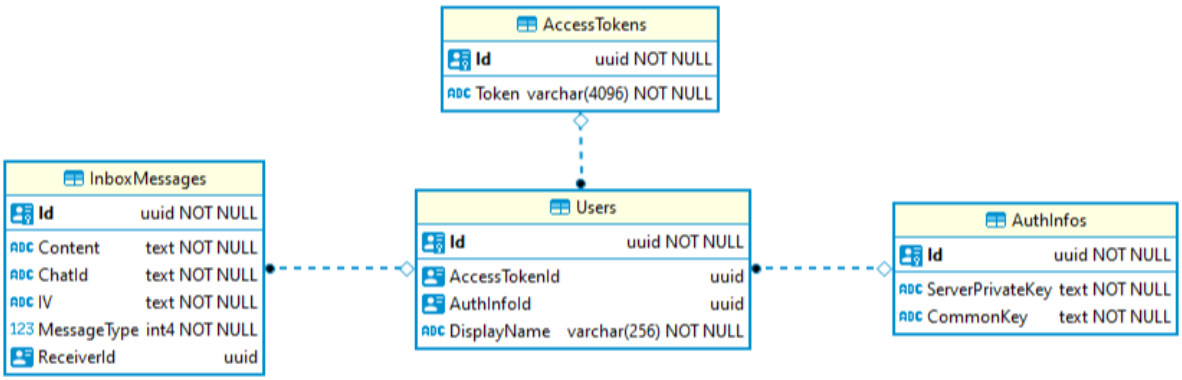
Po doinstalování balíčků je ještě nutné upravit hlavní třídu aplikace. ASP.NET Core používá pro centrální správu závislostí v rámci aplikace principu tzv. dependency injection. Díky této technice nejsou komponenty aplikace závislé na konkrétních implementacích svých závislostí, ale jen na jejich obecném popisu. Příkladem může být právě připojení k databázi. Díky dependency injection je možné jednoduše vyměnit konkrétní implementaci databáze bez toho, aby došlo ke změně fungování komponent, které databázi využívají.

[15] **SQL** je dotazovací jazyk umožňující manipulaci s relačními databázemi.

```
1  var builder = WebApplication.CreateBuilder(args);
2
3  builder.Services
4      .AddDbContext<ApplicationDbContext>(options =>
5      {
6          options.UseNpgsql(builder.Configuration
7          ["ConnectionStrings:DatabaseConnection"]);
8      });
9
10 var app = builder.Build();
11
12 new DatabaseInitializer().InitializeDb(app);
13
14 app.Run();
```

Ukázka kódu 6: Propojení aplikace s databází PostgreSQL, Zdroj: [autor]

V ukázce kódu č. 6 je uveden upravený kód hlavní třídy serverové aplikace po přidání připojení k databázi. Změnou oproti výchozímu stavu třídy, který byl definován na začátku této kapitoly, je přidání databázové služby do kontejneru pro dependency injection. To je ukázáno na řádcích 3-8. Sedmý řádek obsahuje tzv. „connection string“, který specifikuje adresu databázového serveru a přístupové údaje. Tento řetězec je po spuštění aplikace načten z textového souboru s nastavením *appsettings.json*, který byl automaticky vytvořen spolu s projektem. Na dvanáctém řádku je pak možné vidět vytvoření instance třídy *DatabazeInitializer* a její následné využití pro inicializaci databáze. Tato třída byla implementována v rámci vyvíjené aplikace a zajišťuje vytvoření databáze aplikace na databázovém serveru v případě, že ještě není vytvořena. V možném budoucím rozšíření ukázkové aplikace by také mohla provádět migrací dat při změně struktury databáze.



Obrázek 12: Diagram struktury databáze serverové aplikace, Zdroj: [autor]

Struktura databáze serverové aplikace se skládá z několika tabulek. Všechny využívají jako svůj primární klíč datový typ nazvaný GUID. Ten je představován náhodným řetězcem znaků, které je obtížné odhadnout. Této náhodnosti se využívá i z důvodu bezpečnosti. Lze to ilustrovat na následujícím příkladu. Pokud by jako primární klíč byla použita číselná posloupnost, tak by bylo velmi jednoduché odhadnout, že před objednávkou s id 100 se nacházela objednávka s id 99 a po ní bude následovat objednávka s id 101. Tato vlastnost u GUID neplatí.

Tabulka „Users“ představuje ústřední bod celé databáze. Obsahuje sloupec s jménem uživatele a cizí klíče odkazující na tabulky „AccessTokens“ a „AuthInfos“. „AccessTokens“ udržuje aktuální přístupové tokeny všech uživatelů daného serveru. Tabulka „AuthInfos“ zase obsahuje sloupce s klíči potřebnými k šifrované komunikaci mezi serverem a uživatelem. Neméně důležitou je též tabulka „InboxMessages“, která spravuje příchozí zprávy uživatelů. Sloupec „Content“ udržuje zašifrovanou podobu zprávy od komunikačního protějšku. Zbylé sloupce „ChatId“, „IV“ a „MessageType“ pak nejsou relevantní pro serverovou aplikaci a své využití nacházejí v mobilní aplikaci příjemce zprávy, kde jsou využity k dešifrování zprávy a jejího zařazení do správné konverzace.

8.2.3 Endpointy

Šablona „Web“ frameworku ASP.NET Core 6 není ve výchozím stavu nastavena jako plnohodnotná MVC aplikace. Naopak razí opačný postup, kdy vytvořený projekt neobsahuje téměř žádné závislosti a jednotlivé součásti ASP.NET Core mohou být přidávány později podle potřeby. To umožňuje velkou flexibilitu v používání komponent frameworku. MVC kontroléry je tak např. možné nahradit jednodušší formou mapování akcí na jednotlivé endpointy programového rozhraní. Ve vyvíjené aplikaci jsou jednotlivé akce zařazeny do tříd podle společné částí URL adresy dotazu. Tyto třídy jsou vytvořeny jako statické a obsahují tzv „extension“ metody, které umožňují jednoduché přidání či odebírání cest v hlavní třídě aplikace. Jak taková třída mapující endpointy vypadá je možné vidět v ukázce kódu č. 7. Ta popisuje třídu *UserRoute*, která mapuje metody směřované na URL podadresu „*/user*“. Všechny tyto třídy pro mapování endpointů jsou uloženy v adresáři *Routes*, který supluje adresář pro kontroléry v klasickém MVC schématu.

V rámci aplikace pro decentralizovanou komunikaci jsou endpointy rozděleny do tří tříd. Třída *UserRoute* obsahuje endpointy pro správu informací o uživateli. *TokenRoute* představuje třídu mapující endpointy pro získání aktuálního přístupového tokenu a pro případné vytvoření tokenu nového. Namapování endpointů pro získání uložených zpráv a pro přijetí zprávy provádí třída *InboxRoute*. Jak již bylo uvedeno, tak všechny endpointy vrací odpovědi v podobě textového formátu JSON.

```
public static class UserRoute
{
    public static WebApplication MapTokenRoute(this WebApplication app)
    {
        ...

        return app;
    }
}
```

Ukázka kódu 7: Třída mapující endpointy pro správu uživatele, Zdroj: [autor]

Ukázka kódu č. 8 představuje, jak může vypadat mapování konkrétního endpointu v rámci aplikace. ASP.NET Core obsahuje extension metody pro mapování jednotlivých HTTP metod. Ty je možné volat nad instancí třídy *WebApplication*. V ukázce je popsáno využití metody *MapGet* mapující endpoint pro získání nových zpráv uložených na serveru. Tato metoda má dva vstupní parametry. První je textový řetězec reprezentující schéma URL adresy endpointu a druhý představuje metoda zodpovědná za zpracování dotazů, které jsou poslány na zvolený endpoint. V tomto případě je metoda implementována formou tzv. lambda funkce, která vyžaduje dva vstupní parametry. Parametr v podobě instance třídy *ApplicationDbContext* je injektován pomocí dependency injection kontejneru. Textový řetězec s názvem „accessToken“ je automaticky získán z URL adresy dotazu. Tělo této metody začíná kontrolou, zda byl dodán přístupový token. V případě, že nebyl poskytnut, je odeslána chybová odpověď. Podle tokenu je v databázi vyhledán uživatel, kterému tento token patří. Pokud by nebyl nalezen žádný uživatel, tak je opět odeslána chybová odpověď. Následně dojde k vyhledání a zpracování všech doručených zpráv, které na uživatele čekají na serveru. Nakonec jsou nalezené zprávy smazány z databáze a odeslány klientovi.

```
app.MapGet("/inbox/{accessToken}", async (
    ApplicationDbContext dbContext,
    string accessToken) =>
{
    if (accessToken == null)
    {
        return Results.BadRequest(new ApiResponse<ErrorPayload>()
        {
            ResponseType = ApiResponseType.Error,
            Payload = new ErrorPayload()
            {
                ErrorType = ErrorPayloadType.WrongOrMissingParameters,
                Message = "Access token not provided"
```

```
14              }
15          });
16      }
17
18      var user = await dbContext.Users
19          .Include(x => x.AccessToken)
20          .Include(x => x.AuthInfo)
21          .FirstOrDefaultAsync(x => x.AccessToken.Token == accessToken);
22
23      if (user == null)
24      { ... }
25
26      var dbMessages = await dbContext.InboxMessages
27          .Include(x => x.Receiver)
28          .Where(x => x.Receiver.Id == user.Id)
29          .ToArrayAsync();
30
31      var messages = new InboxMessageGetResponsePayload
32          [dbMessages.Length];
33
34      for (int i = 0; i < dbMessages.Length; i++)
35      {
36          messages[i] = new InboxMessageGetResponsePayload()
37          {
38              ChatId = dbMessages[i].ChatId,
39              Content = dbMessages[i].Content,
40              MessageType = dbMessages[i].MessageType,
41              IV = dbMessages[i].IV
42          };
43      }
44
45      dbContext.RemoveRange(dbMessages);
46
47      dbContext.SaveChanges();
48
49      var response = new ApiResponse<InboxMessageGetResponsePayload[]>()
50      {
51          ResponseType = ApiResponseType.OK,
52          Payload = messages
53      };
54
55      return Results.Ok(response);
56  });
```

Ukázka kódu 8: Metoda mapující endpoint pro získání doručených zpráv, Zdroj: [autor]

8.3 Implementace mobilní aplikace

Po dokončení implementace serverové části aplikace je potřeba vytvořit klientskou aplikaci, která bude webovou službu konzumovat. K tomu bude využit projekt Xamarin.Forms aplikace, jehož založení bylo popsáno na začátku této kapitoly.

8.3.1 MVVM

Návrhový vzor **Model-View-ViewModel (MVVM)** byl v roce 2005 navrhnut Johnem Gossmanem a měl sloužit pro modularizaci desktopových a webových aplikací. Stejně jako v případě MVC je hlavním úkolem MVVM oddělit datové modely, aplikační logiku a uživatelské rozhraní. Zásadní rozdíl oproti MVC je však absence kontrolérů, které jsou zde nahrazeny ViewModely. Jejich výhodou je, že pomocí tzv. **Bindingu** je možné jejich data obousměrně propojit s pohledy. To v praxi znamená, že změna v datech ViewModelu se automaticky projeví v uživatelském rozhraní. Tento princip funguje i obráceně, kdy se každá změna v uživatelské rozhraní projeví ve ViewModelu. Příkladem frameworku využívajícího návrhový vzor MVVM je právě Xamarin.Forms. [54]

Proto, aby bylo možné používat návrhový vzor MVVM v Xamarin.Forms, je nutné provést vytvoření třídy *BaseViewModel*. Z této třídy pak dědí všechny ViewModely ve vyvíjené aplikaci. Třída implementuje rozhraní *INotifyPropertyChanged*, jenž je součástí frameworku Xamarin.Forms. Toto rozhraní vyžaduje přidání události *PropertyChanged*, která je vyvolána při změně hodnoty nějaké vlastnosti (atributu) třídy. Dále je také nutné implementovat metodu *OnPropertyChanged*, jenž slouží k vyvolání výše popsané události *PropertyChanged*. Vytváření přímých instancí této třídy nedává smysl, proto je označena jako abstraktní.

```
1  public abstract class BaseViewModel : INotifyPropertyChanged
2  {
3      public event PropertyChangedEventHandler PropertyChanged;
4
5      public void OnPropertyChanged(string name)
6      {
7          PropertyChanged?.Invoke(
8              this,
9              new PropertyChangedEventArgs(name));
10     }
11 }
```

Ukázka kódu 9: Třída *BaseViewModel*, Zdroj: [autor]

Ukázka kódu č. 10 zobrazuje ukázkovou implementaci ViewModelu. Tato třída dědí z vytvořené abstraktní třídy *BaseViewModel*. Pro správné fungování je nutné v setteru zvolené vlastnosti (zde je to vlastnost *„Name“*) nastavit hodnotu privátního atributu, zavolat metodu *OnPropertyChanged* a jako její parametr uvést název vlastnosti. To je zde provedeno příkazem *nameof*, který automaticky získá textovou reprezentaci zvolené vlastnosti. Takto bude Xamarin.Forms vědět, že došlo ke změně hodnoty vlastnosti a aktualizuje ji ve zbytku aplikace.

```
1  public class PersonViewModel : BaseViewModel
2  {
3      private string mName;
4
5      public string Name
6      {
7          get { return mName; }
8          set
9          {
10             mName = value;
11             OnPropertyChanged(nameof(Name));
12         }
13     }
14 }
```

Ukázka kódu 10: Ukázka implementace ViewModelu, Zdroj: [autor]

Tento způsob tvorby ViewModelů je plně funkční a validní, ale při velkém množství vlastností může být značně nepohodlný na implementaci. To lze řešit doinstalováním balíčku *PropertyChanged.Fody*, díky kterému bude stačit zápis v následující zjednodušené podobě: „`public string Name { get; set; }`“. Není tak nutné v setteru vlastnosti manuálně volat metodu *OnPropertyChanged*, protože balíček *PropertyChanged.Fody* přidá toto volání automaticky během kompilace kódu.

```
1  dotnet add package PropertyChanged.Fody
```

Ukázka kódu 11: Instalace balíčku *PropertyChanged.Fody*, Zdroj: [autor]

Po nainstalování balíčku *PropertyChanged.Fody* je ještě nutné vytvořit soubor *FodyWeavers.xml*. Ten obsahuje nastavení automatického vkládání volání metody *OnPropertyChanged*. Jeho obsah je možné vidět v ukázce kódu č. 12. Při spuštění kompilace projektu dojde ještě k vytvoření souboru *FodyWeavers.xsd*. V tuto chvíli je projekt mobilní aplikace připraven pro vývoj za pomoci návrhového vzoru MVVM.

```
<Weavers
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="FodyWeavers.xsd">
    <PropertyChanged />
</Weavers>
```

Ukázka kódu 12: Soubor *FodyWeavers.xml*, Zdroj: [autor]

8.3.2 Databáze

Pro ukládání perzistentních dat v mobilní aplikaci je využita databáze LiteDB. Ta byla již přiblížena v kapitole č. 6, která se věnovala představení technologií použitých při vývoji aplikace pro decentralizovanou komunikaci. Pro její použití je nutné do Xamarin.Forms projektu doinstalovat patřičný balíček.

```
dotnet add package LiteDB
```

Ukázka kódu 13: Instalace balíčku *LiteDB*, Zdroj: [autor]

LiteDB ukládá datové entity ve formě strukturovaných objektů typu BSON. Díky tomu není nutné používat ORM frameworky pro mapování objektů na databázové entity a naopak. K použití LiteDB databáze stačí vytvořit novou instanci objektu *LiteDatabase* v rámci *using* tvrzení[16]. Pro získání kolekce entit, do které má být nová entita vložena, je potřeba nad instancí databázového objektu zavolat metodu *GetCollection*. Ta vyžaduje specifikaci generického parametru, který udává datový typ jednotlivých objektů kolekce entit. Jako vstupní parametr metody je nutné uvést cestu k souboru, kde se databáze nachází. Poté na této kolekci stačí zavolat metodu *Insert* a jako parametr ji dodat instanci objektu reprezentující vkládanou entitu.

```
using (var db = new LiteDatabase("DatabaseName.db"))
{
    var collection =
        db.GetCollection<CollectionType>("CollectionName");

    collection.Insert(itemInstance);
}
```

Ukázka kódu 14: Ukázka uložení dat do LiteDB, Zdroj: [autor]

[16] ***Using* tvrzení** zajistí automatické uzavření a uvolnění zdrojů po dokončení požadovaných operací.

Proces získání dat z databáze je z větší části velmi podobný tomu při vkládání dat. Opět je nutné vytvořit instanci databázového objektu a získat kolekci objektů. Pro nalezení jedné instance objektu je možné využít metodu *FindOne*. Ta vyžaduje jako vstupní parametr specifikaci vlastnosti, podle které bude instance vyhledána. V případě, že je požadována kolekce datového typu *List* všech entit zvoleného typu, tak stačí nad databázovou kolekcí za sebou zavolat metody *FindAll* a *ToList*.

```
1  using (var db = new LiteDatabase("DatabaseName.db"))
2  {
3      var collection =
4          db.GetCollection<CollectionType>("CollectionName");
5
6      var singleItem = collection.FindOne(x => x.Id == id);
7
8      var allItemsInList = collection
9          .FindAll()
10         .ToList();
11 }
```

Ukázka kódu 15: Ukázka získání dat z LiteDB, Zdroj: [autor]

V databázi mobilní aplikace jsou vytvořeny tři kolekce. První z nich je *Account*, kde jsou uložena data uživatele. Sem patří např. přihlašovací údaje k domovskému serveru. V kolekci *Chat* lze nalézt identifikační informace k jednotlivým konverzacím. Kromě nich jsou zde také uloženy údaje umožňující šifrovanou komunikaci. Poslední kolekce *Message* zase udržuje zprávy ze všech konverzací.

8.3.3 Struktura aplikace

Mobilní klient aplikace pro decentralizovanou komunikaci je rozdělen do dvou .NET projektů. Projekt *DiplomaThesis.MobileApp.Android* obsahuje kód specifický pro operační systém Android. Jeho adresáře *Assets* a *Resources* slouží k ukládání platformě specifických neprogramových souborů. *MainActivity.cs* pak představuje výchozí třídu pro Android aplikaci, která po spuštění obstará načtení a nastavení frameworku Xamarin.Forms.

Projekt *DiplomaThesis.MobileApp* obsahuje části kódu sdílené mezi platformami. V jeho struktuře je možné spatřit soubory s příponou „*.xaml*“. Ty využívají upravenou verzi značkovacího jazyka XML nazvanou jako XAML, která je určena pro vývoj aplikací na platformě .NET. Dále se zde také nachází soubory s příponou „*.xaml.cs*“. Jedná se o částečné implementace tříd, které doplňují třídy definované v „*.xaml*“ souborech.

Obrázek 13: Diagram struktury mobilní aplikace, Zdroj: [autor]

Nejdůležitější třídou celé aplikace je *App*. Jde o hlavní třídu Xamarin.Forms aplikace, která je načtena po spuštění. Obsahuje např. metody *OnStart*, *OnSleep* a *OnResume*, jenž umožňují reagovat na změnu stavu aplikace. V rámci aplikace pro decentralizovanou komunikaci není využití těchto metod nutné, a tak je možné je ponechat v jejich původní implementaci. Daleko zajímavější je třída implementovaná v souborech s předponou *AppShell*. Jedná se o třídu, která definuje vizuální a logickou navigaci v rámci aplikace. Ukázka kódu č. 16 detailně popisuje její obsah. Xamarin.Forms *Shell* jako výchozí způsob navigace v aplikaci preferuje využití bočního menu, které je možné vyvolat tlačítkem na navigační liště v horní části obrazovky. Vyvíjená aplikace však tento způsob ovládání využívat nebude a přechody mezi obrazovkami bude řešit manuálním přepnutím v rámci programového kódu. Z tohoto důvodu bude v hlavním „tagu“ s názvem *Shell* nastaven atribut *FlyoutBehavior* na hodnotu *Disabled*. V rámci tohoto tagu jsou pak definovány vnitřní tagy typu *ShellContent*. Ty popisují jednotlivé cesty dostupné v rámci aplikace a mají nastavenou hodnotu u tří atributů. *Title* udává název cesty, *Route* definuje její adresu a atribut *ContentTemplate* zase stanovuje šablonu stránky, která bude zobrazena po přechodu na zvolenou cestu.

```
<?xml version="1.0" encoding="utf-8" ?>
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:pages="clr-namespace:DiplomaThesis.MobileApp.Pages"
       x:Class="DiplomaThesis.MobileApp.AppShell"
       FlyoutBehavior="Disabled">

    <ShellContent Title="Login"
        Route="login"
        ContentTemplate="{DataTemplate pages:LoginPage}" />

    <ShellContent Title="Register"
        Route="register"
        ContentTemplate="{DataTemplate pages:RegisterPage}" />

    <ShellContent Title="Chats"
        Route="chats"
        ContentTemplate="{DataTemplate pages:ChatsPage}" />

    <ShellContent Title="AddNewChat"
        Route="addnewchat"
        ContentTemplate="{DataTemplate pages:AddNewChatPage}" />

    <ShellContent Title="Chat"
        Route="chat"
        ContentTemplate="{DataTemplate pages:ChatPage}" />

</Shell>
```

Ukázka kódu 16: Soubor *AppShell.xaml*, Zdroj: [autor]

Soubor *FodyWeavers.xml* slouží k nastavení knihovny *PropertyChanged.Fody* a byl blíže přiblížen v předchozí části této kapitoly. *AssemblyInfo.cs* obsahuje informace o požadovaných oprávněních a nastaveních, která jsou při kompilaci přenesena do platformě specifických projektů. Adresáři *Encryption* bude věnována celá samostatná podkapitola později. Složka s názvem *Helpers* je určena pro zdrojové kódy, jenž přímo nesouvisí s žádnými komponentami projektu, ale usnadňují některé často prováděné operace. Například soubor *HttpHelper.cs* definuje třídu *HttpHelper*, která zjednodušuje volání HTTP endpointů a podle uvedeného generického typu provádí serializaci dotazu do JSON formátu či deserializaci odpovědi na správný datový typ. Je zde také umístěn soubor s třídou *LiteDbHelper*, jenž obsahuje metodu pro snadné získání cesty k souboru s lokální databází. Tato metoda je připravena tak, aby respektovala pravidla pro ukládání souborů na dané platformě. Může tak dojít k tomu, že výsledná cesta k databázi se bude na různých ope-

račních systémech lišit. *Models* představuje adresář určený pro modelové třídy aplikace. Podadresář *Database* obsahuje třídy definující databázové entity. Zajímavějšími jsou ale podadresáře *Api* a *ApiPayload*. V prvním zmíněném se nachází soubory *ApiRequest.cs*, *ApiResponse.cs* a *ApiResponseType.cs*. *ApiRequest.cs* definuje třídu popisující dotazy odesílané na serverovou část vyvíjené aplikace. Uvnitř *ApiResponse.cs* lze zase nalézt třídu, která určuje strukturu odpovědi odeslané ze serveru zpět do klientské aplikace. Ukázka kódu č. 17 tuto třídu přímo zobrazuje. Při vytváření její instance je nutné definovat typ „nákladu“, který objekt ponese, a samozřejmě ho do třídy dodat. Kromě toho je nutné specifikovat vlastnost *ResponseType*. Ta může nabývat hodnot výčtového datového typu *ApiResponseType* specifikovaného ve zmíněném souboru *ApiResponseType.cs*. Podsložka *ApiPayload* obsahuje definice jednotlivých typů „nákladů“, které mohou být do tříd *ApiRequest* a *ApiResponse* dosazeny. Posledními dvěma podadresáři složky *Models* jsou *Inbox* a *Messaging*. Ty obsahují soubory s třídami vymezujícími typy zpráv, které si uživatelé mezi sebou mohou zaslat.

```
1  public class ApiResponse<TPayload>
2  {
3      public ApiResponseType ResponseType { get; set; }
4
5      public TPayload Payload { get; set; }
6  }
```

Ukázka kódu 17: Třída *ApiResponse*, Zdroj: [autor]

Adresář *Pages* obsahuje šablony jednotlivých obrazovek aplikace. Jejich popisu bude věnována podkapitola 8.3.4. Složka *Resources* je určena k ukládání neprogramových souborů aplikace. Je zde uložen soubor s textovým fontem definujícím ikony použité při tvorbě grafického uživatelského rozhraní. *Services* je adresář obsahující soubory s třídami *AuthService* a *MessageService*. *ValueConverters* zase slouží k ukládání speciálních tříd implementujících rozhraní *IValueConverter*. Tyto třídy jsou úzce spjaté s uživatelským rozhraním a umožňují definovat jednoduché převádění hodnot mezi různými datovými typy. Lze tak např. implementovat převod proměnné datového typu Boolean[17] na barvu v uživatelském rozhraní. Poslední adresář projektu s platformě nezávislým kódem se nazývá *ViewModels*. Ten obsahuje ViewModely aplikace. Jak ViewModely fungují bylo popsáno v podkapitole 8.3.1. Podadresář *Base* obsahuje definici základního ViewModelu *BaseViewModel*. V hlavním adresáři *ViewModels* se nachází pět ViewModelů, kde každý přísluší k jedné ze šablon definovaných ve složce *Pages*.

[17]**Boolean** je datový typ, který může nabývat hodnot *true* (pravda) a *false* (nepravda).

8.3.4 Uživatelské rozhraní

Z pohledu uživatele patří grafické rozhraní k nejdůležitějším částem aplikace. Uživatel jeho prostřednicím aplikaci ovládá a využívá její funkce. Proto je nutné, aby bylo uživatelské rozhraní navrhnuto co nejpřehledněji a nejintuitivněji. Jak bylo zmíněno v předchozích částech práce, tak framework Xamarin.Forms používá k definování grafického uživatelského rozhraní upravenou verzi značkovacího jazyka XML nazvanou jako XAML. Z každého XAML souboru je při kompilaci vygenerována standardní C# třída. Soubory s příponou „*.xaml.cs*“, které doplňují definici tříd specifikovaných v souborech „*.xaml*“, se nazývají „Code-Behind“.

Xamarin.Forms rozděluje uživatelské rozhraní aplikace do celků nazvaných obrazovky. Ty obsahují prvky, se kterými může uživatel interagovat. Jde např. o různá textová pole, tlačítka či menu. Aby bylo možné aplikaci pomocí grafického rozhraní ovládat, je nutné ho propojit s ViewModelem, který bude obstarávat provádění logiky. Může jít např. o provedení akce po kliknutí na tlačítko. Pro toto propojení je nejprve potřeba vytvořit obrazovku (v Xamarin.Forms jde o třídu dědící z třídy *ContentPage*) a k ní příslušný ViewModel. Třída *ContentPage* obsahuje vlastnost s názvem *BindingContext*. Ta specifikuje objekt, ze kterého bude obrazovka čerpat data. Ukázka kódu č. 18 ukazuje dosazení nové instance ViewModelu *LoginPageViewModel* do vlastnosti *BindingContext* v konstruktoru obrazovky *LoginPage*. Alternativně lze *BindingContext* nastavit přímo v XAML kódu obrazovky. To by se provedlo nastavením hodnoty párového tagu s názvem *ContentPage.BindingContext*. Tento tag se musí nacházet uvnitř párového tagu *ContentPage*.

```
1  public partial class LoginPage : ContentPage
2  {
3      public LoginPage()
4      {
5          InitializeComponent();
6
7          BindingContext = new LoginPageViewModel(this);
8      }
9  }
```

Ukázka kódu 18: Code-Behind třídy *LoginPage*, Zdroj: [autor]

Po vytvoření tohoto spojení mezi ViewModelem a obrazovkou je možné začít implementovat logiku aplikace. Nyní je potřeba ve ViewModelu vytvořit vlastnost, která má být použita v grafickém rozhraní. V ukázce kódu č. 19 jde o vlastnost *Server* ve ViewModelu *LoginPageViewModel*.

```
1  public class LoginPageViewModel : BaseViewModel
2  {
3      ....
4      public string Server { get; set; }
5      ...
6  }
```

Ukázka kódu 19: Vytvoření vlastnosti *Server* v *LoginPageViewModel*, Zdroj: [autor]

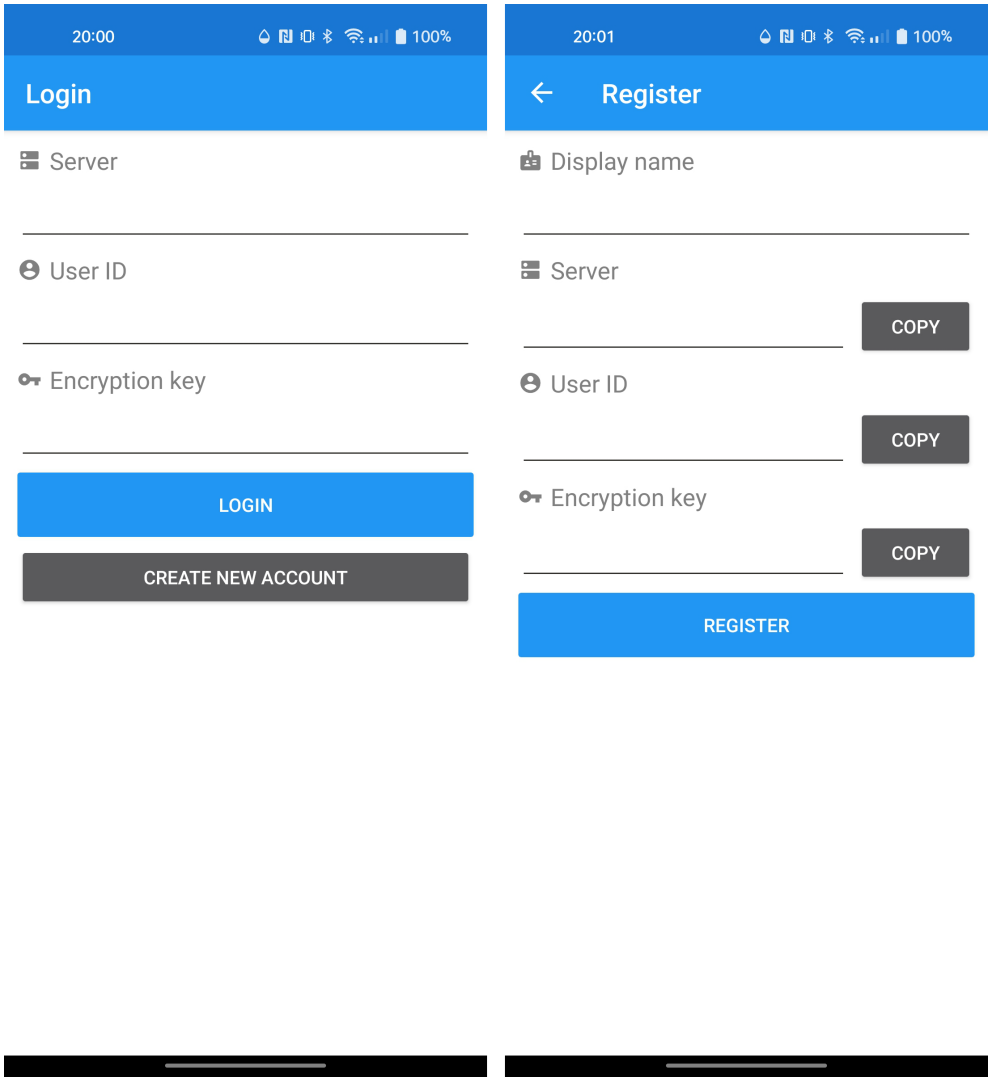
Pro propojení hodnoty vlastnosti ViewModelu s prvkem uživatelského rozhraní stačí již pouze definovat tzv. *Binding*. Ten se specifikuje přímo na atributu prvku grafického rozhraní. Do jeho hodnoty se uvede následující výraz ve složených závorkách: *„{Binding Vlastnost}“*. *Vlastnost* je pak nahrazena konkrétní vlastností propojeného ViewModelu. V *Bindingu* je možné uvést i další parametry. Jedním z nich je *Mode*, který umožňuje specifikovat směr *Bindingu*. Pokud je *Mode* nastaven na hodnotu *OneWay*, tak se změny v hodnotně vlastnosti ViewModelu projeví v uživatelském rozhraní, ale změny v uživatelském rozhraní se neprojeví ve ViewModelu. Mód *OneWayToSource* provádí opak předchozího a změny v uživatelském rozhraní se projeví v hodnotě vlastnosti ViewModelu, ale změny v hodnotě vlastnosti ViewModelu se neprojeví v uživatelském rozhraní. *TwoWay* pak představuje mód, kdy se změny na obou stranách projevují zase na obou stranách. Dalším zajímavým nepovinným parametrem je *Converter*, který umožní v případě potřeby hodnotu vlastnosti ViewModelu převést na požadovanou hodnotu pro grafický prvek. V ukázce kódu č. 20 je možné vidět provedení *Bindingu* vlastnosti *Server* s atributem *Text* na grafickém prvku *Entry*, který představuje textové pole. Mód *Bindingu* je zvolen jako obousměrný (*TwoMay*).

```
1  <Entry Text="{Binding Server, Mode=TwoWay}"
2         TextColor="Gray"/>
```

Ukázka kódu 20: Vytvoření *Bindingu* v XAML kódu, Zdroj: [autor]

Při návrhu vyvíjené aplikace bylo specifikováno, že uživatelské prostředí bude respektovat designový jazyk Material Design. Výchozí vzhled grafických prvků frameworku Xamarin.Forms na operačním systému Android je vytvořen tak, aby odpovídal právě jazyku Material Design. Nebylo proto nutné vytvářet žádné vlastní styly a stačilo použít výchozí vzhled definovaný frameworkem.

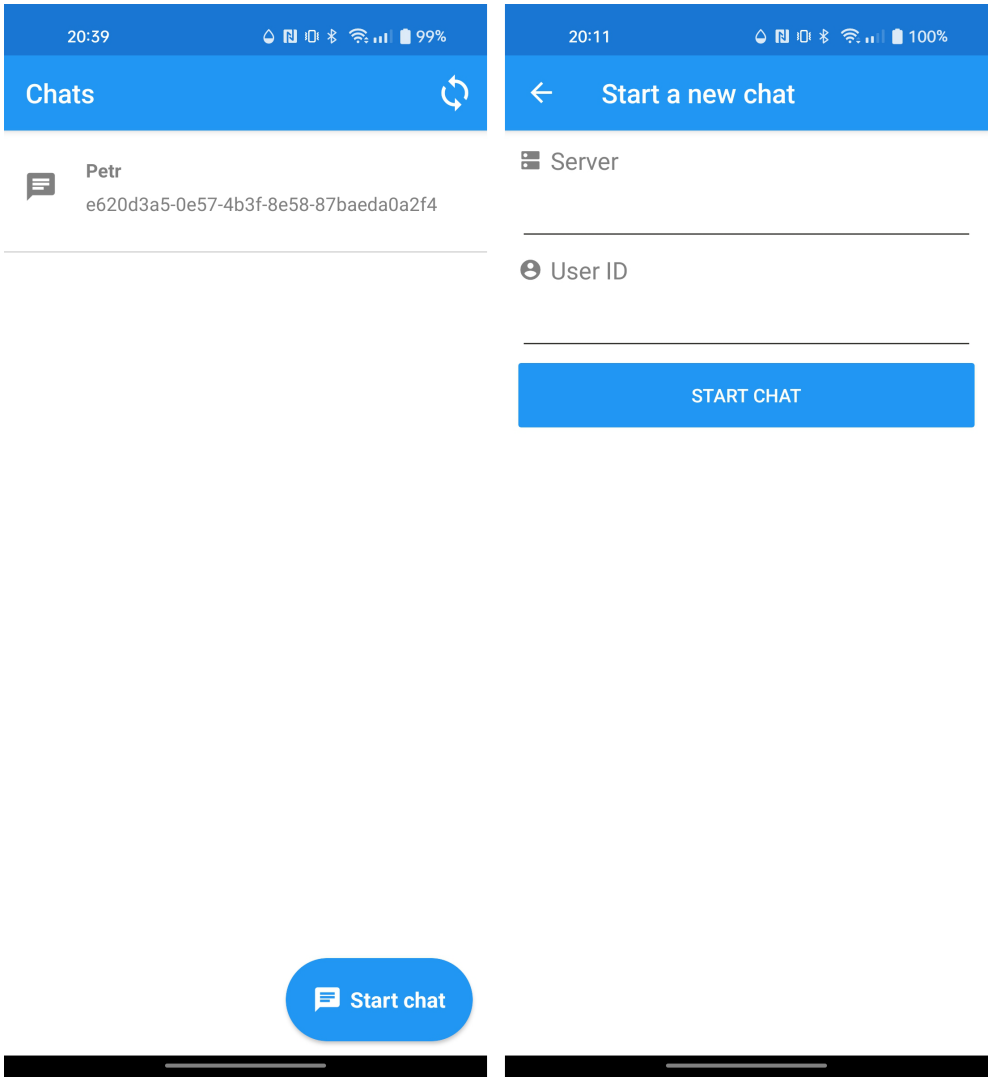
Uživatelské rozhraní vyvíjené mobilní aplikace se skládá z pěti obrazovek. Každá z nich představuje svůj případ užití. V následujících odstavcích budou jednotlivé obrazovky a jejich úloha v aplikaci detailně popsány.



Obrázek 14: Obrazovky pro přihlášení a registraci uživatele, Zdroj: [autor]

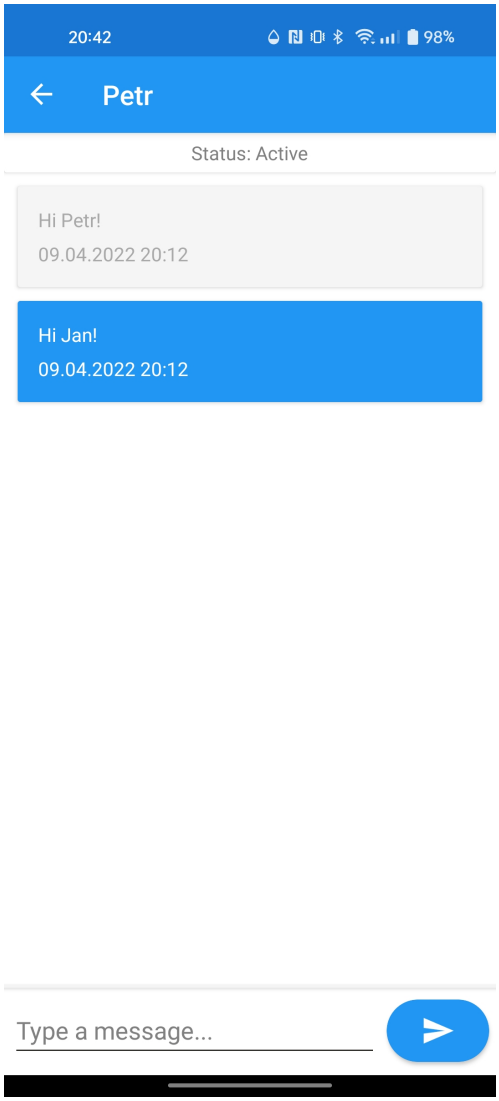
V levé části obrázku č. 14 je možné vidět podobu obrazovky pro přihlášení definovanou ve třídě *LoginPage*. Tuto obrazovku uživatel uvidí při prvním spuštění aplikace. Jejím cílem je poskytnout formulář, do kterého uživatel uvede přihlašovací údaje ke svému účtu na domovskému serveru. V horní části obrazovky se nachází světle modrý pruh reprezentující menu s akcemi. V případě této obrazovky je zde uveden pouze její název. V samotném těle jsou pak umístěna tři textová pole a k nim příslušící popisky definující, k čemu daná pole slouží. Konkrétně specifikují adresu domovského serveru uživatele, jeho identifikační číslo a klíč pro šifrovanou komunikaci s uvedeným serverem. Pod nimi se nachází dvě tlačítka. Modré s názvem „Login“ po kliknutí vyvolá akci, jenž se podle zadaných údajů pokusí navázat spojení se zvoleným serverem a přihlásit k němu uživatele. Pokud je přihlášení úspěšné, tak dojde k přesunu na instanci obrazovky *ChatsPage*. Tmavě šedé tlačítko s označením „Create new account“ slouží k přesunu na instanci obrazovky *RegisterPage*.

Druhou obrazovkou znázorněnou na obrázku č. 14 je obrazovka *RegisterPage*. Jejím úkolem je poskytnout uživateli formulář pro registraci nového účtu na zvoleném domovském serveru. Oproti obrazovce pro přihlášení se zde v horním menu akcí nachází tlačítko s ikonou šipky směřující vlevo. Kliknutím na tuto šipku je uživatel navrácen zpět k přihlášení. V těle obrazovky se nachází čtyři textová pole. První udává jméno uživatele. Druhé slouží k definování domovského serveru. Třetí a čtvrté pole s identifikačním číslem uživatele a šifrovacím klíčem jsou nastavena pouze pro čtení a data do nich jsou automaticky dosazena po dokončení registrace. Ta je vyvolána stisknutím modrého tlačítka „Register“ nacházejícího se pod popsanými textovými poli. Na obrazovce jsou ještě umístěna tři šedá tlačítka s názvem „Copy“. Každé z nich je umístěno u konkrétního textového pole a po provedení kliknutí dojde ke zkopírování hodnoty daného pole do schránky. Uživatel si tak může své údaje jednoduše uložit na nějaké bezpečné místo. Po dokončení registrace je možné se vrátit zpět na stránku pro přihlášení.



Obrázek 15: Obrazovky se seznamem konverzací a s novou konverzací, Zdroj: [autor]

Obrázek č. 15 ukazuje podobu obrazovek s výpisem konverzací a zahájením nové konverzace. První zmíněná je instancí třídy *ChatsPage*. V menu akcí má na pravé straně tlačítko s ikonou dvou šipek. Toto tlačítko po stisknutí naváže spojení s domovským serverem a stáhne všechny zprávy, které uživatel nově obdržel. Tělo této obrazovky je specifické v tom, že seznam konverzací je možné pomocí dotyku posouvat. Po kliknutí na konkrétní konverzaci je uživatel přesměrován na obrazovku definovanou třídou *ChatPage*. Druhým specifikem obrazovky konverzací je tlačítko pro zahájení nové konverzace. To je vždy v popředí na pravé spodní straně obrazovky a neposouvá se při interakci se seznamem konverzací. Po kliknutí na toto tlačítko je uživateli zobrazena obrazovka pro zahájení nové konverzace. V menu akcí obrazovky *AddNewChatPage* je opět umístěno tlačítko pro návrat zpět. Tělo obrazovky obsahuje textová pole specifikující domovský server a identifikační číslo uživatele, se kterým má být konverzace zahájena. Tlačítko „Start chat“ provede zahájení nové konverzace.



Obrázek 16: Obrazovka s konverzací, Zdroj: [autor]

Obrazovka, kterou je možné vidět na obrázku č. 16, má za cíl zobrazit konkrétní konverzaci. V menu akcí se nachází tlačítko pro návrat k seznamu konverzací a popisek, který nese jméno komunikačního protějšku. Pod tímto menu je umístěn řádek označený jako „Status“. Ten informuje o tom, v jakém stavu se konverzace aktuálně nachází. Následuje historie konverzace v podobě seznamu přijatých a odeslaných zpráv. Podobně jako v případě seznamu konverzací je i zde možné se v něm pohybovat pomocí dotyku. Přijaté zprávy jsou pro snadnější rozeznání obarveny modře. Na spodní straně obrazovky je umístěno textové pole, jenž slouží jako prostor pro text zprávy, která má být odeslána. Dále je zde ještě tlačítko pro samotné odeslání zprávy.

8.3.5 Logika aplikace

Předchozí podkapitola 8.3.4 se věnovala popisu uživatelského rozhraní aplikace. V rámci tohoto prostředí bylo definováno několik tlačítek, která reagovala na interakci od uživatele. Proto, aby k této reakci došlo, je nutné provést *Binding* atributu *Command*. Do něj je nutné dosadit instanci třídy implementující rozhraní *ICommand*. Xamarin.Forms má vlastní implementaci tohoto rozhraní v podobě třídy *Command*.

```
1  public class LoginPageViewModel : BaseViewModel
2  {
3      ....
4
5      public ICommand OpenChatCommand { get; }
6
7      ...
8
9      public ChatsPageViewModel(ChatsPage chatsPage)
10     {
11         ...
12
13         OpenChatCommand = new Command<Chat>(async (chat) =>
14         {
15             await Shell
16                 .Current
17                 .Navigation
18                 .PushAsync(new ChatPage(chat));
19         });
20     }
21 }
```

Ukázka kódu 21: Využití třídy *Command* pro akce v uživatelském rozhraní, Zdroj: [autor]

Ukázka kódu č. 21 zobrazuje vytvoření instance třídy *Command*. Ta je dosazena do vlastnosti *OpenChatCommand* a jejím cílem je zajistit přechod na obrazovku se zvolenou konverzací. V ukázce je specifikován generický parametr, jenž však není povinný. Pokud je použit, tak slouží k definici typu vstupního parametru metody, která je vytvořenou instancí třídy *Command* reprezentována. Právě metoda je jediným parametrem, jenž je nutné při vytváření instance dodat. V uvedené ukázce kódu je definována pomocí lambda funkce a jako vstupní parametr přijímá objekt typu *Chat*. Klíčová slova *async* a *await* slouží pro jednodušší využívání asynchronních metod aplikace. Kód v těle metody pak vyvolá přechod na obrazovku se zvolenou konverzací.

Předchozí uvedený příklad použití třídy *Command* demonstroval způsob vytváření akcí v aplikacích využívajících framework Xamarin.Forms. V rámci mobilního klienta vyvíjené aplikace je implementována celá řada instancí této třídy, které provádějí složitější logiku než jen změnu obrazovky aplikace. Příkladem může být *SyncCommand* ve třídě *ChatsPageViewModel*, který provede stažení zpráv z domovského serveru a jejich následné zpracování.

8.4 Šifrovaná komunikace

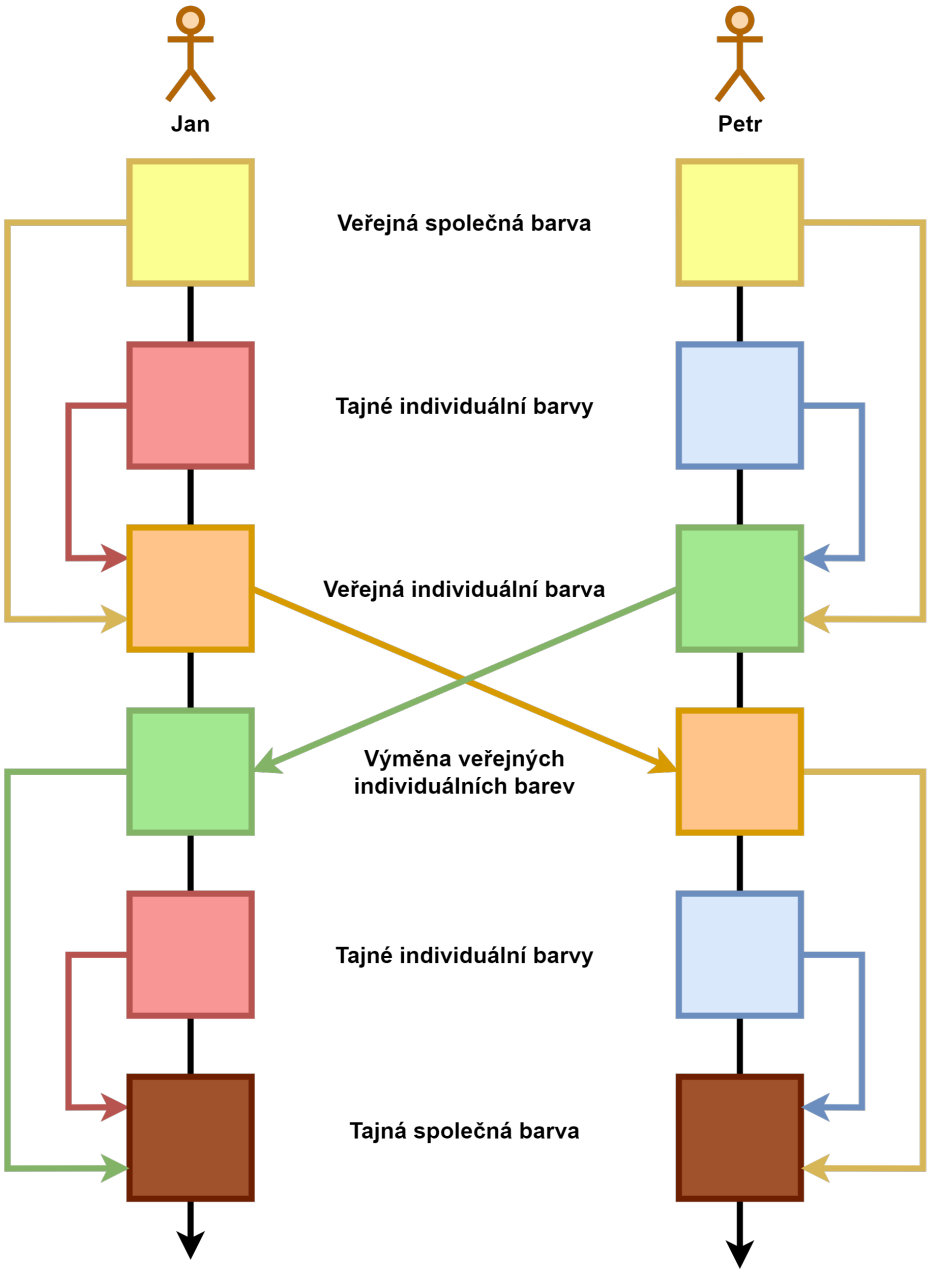
Jedním z non-funkční požadavků uvedených v kapitole č.7 bylo šifrování veškeré probíhající komunikace. Úkolem této podkapitoly bude přiblížit konkrétní implementaci šifrování v aplikaci pro decentralizovanou komunikaci.

8.4.1 Diffie-Hellman

Vyvíjená aplikace bude využívat symetrického šifrování. To vyžaduje stejný klíč při šifrování i dešifrování. Zaslání tohoto klíče po nezabezpečeném připojení nedává příliš smysl, protože pokud by došlo k jeho zachycení, mohl by být zneužit k dešifrování následně odeslaných zpráv. Řešením by bylo klíč druhé straně doručit pomocí nějakého zabezpečeného kanálu. Problémem je, že ten nemusí být vždy dostupný. Existuje však způsob, který zajistí, že obě strany budou mít stejný klíč, ale zároveň nebude nutné jeho přímé zaslání po síti.

Diffie-Hellman je protokol, jenž se tímto problémem zabývá. Obrázek č. 17 zobrazuje zjednodušenou formu tohoto protokolu pomocí systému míchání barev. Na začátku procesu je požadavek na získání stejné barvy na obou stranách komunikace bez toho, aby bylo nutné tuto barvu druhé straně přímo sdělit. Výsledná barva je analogií k symetrickému klíči, který je použit k šifrování a dešifrování zpráv na obou stranách. Nejdříve se oba účastníci dohodnou na jedné společné barvě, jenž je veřejnou informací. Každý z

účastníků si také zvolí svou tajnou barvu, kterou nebude nikomu sdělovat. Obě strany komunikace pak vezmou dohodnutou společnou barvu (žlutá) a smíchají ji se svojí tajnou barvou (červená u Jana, modrá u Petra). Tyto nové barvy (oranžová u Jana, zelená u Petra) jsou veřejnou informací a účastníci si je vzájemně předají. Mělo by platit, že z těchto smíchaných barev nelze jednoduše určit původní barvy, jež byly k jejich namíchání použity. V tuto chvíli zbývá již jen poslední krok k získání tajné společné barvy. Každý z účastníků vezme namíchanou barvu, kterou získal od protějšku, a smíchá ji se svoji tajnou barvou. Nyní mají oba účastníci stejnou výslednou barvu bez nutnosti jejího přímého přenosu přes nezabezpečené připojení. [55]



Obrázek 17: Schéma protokolu Diffie-Hellman, Zdroj: [autor]

8.4.2 Implementace Diffie-hellman

Pro implementování protokolu Diffie-Hellman je nejprve nutné zvolit dvě čísla g a p. Jako hodnota čísla g by mělo být zvoleno malé prvočíslo. Naopak číslo p by mělo být z důvodu bezpečnosti velmi velké prvočíslo. Je vhodné využít některou ze skupin definovaných v *RFC-3526*. [56] Vyvíjená aplikace používá skupinu *8192-bit MODP Group* uvedenou ve zmíněném dokumentu. Čísla g a p jsou veřejnou informací. Za to čísla a a b představují soukromé klíče účastníků komunikace. Jedná se o náhodná čísla, která mobilní klient generuje za pomocí vlastní třídy *BigIntegerGenerator*. Nyní musí první účastník vypočítat výraz „$A = g^a \bmod p$“ a druhý účastník výraz „$B = g^b \bmod p$“. [57] Tím získají čísla reprezentující jejich veřejné klíče. Je důležité poznamenat, že z čísel A a B nelze jednoduše získat původní čísla a a b. Programová implementace tohoto výpočtu je zobrazena v ukázce kódu č. 22.

```
1  public BigInteger CalculatePublicKey(
2      BigInteger primeNumber,
3      BigInteger generator,
4      BigInteger secretKey)
5  {
6      return (generator ^ secretKey) % primeNumber;
7  }
```

Ukázka kódu 22: Výpočet veřejného klíče v protokolu Diffie-Hellman, Zdroj: [autor]

Po dokončení výpočtů veřejných klíčů A a B si je účastnici konverzace vzájemně vymění přes nezabezpečené připojení. Poté obě strany provedou výpočet konečného společného čísla. Na straně prvního uživatele je vypočten výraz „$K = B^a \bmod p$“. [57] Druhý uživatel zase vypočítá výraz „$K = A^b \bmod p$“. [57] Programová implementace v rámci vyvíjené aplikace je ukázána v ukázce kódu č. 23. Obě popsané metody *CalculatePublicKey* a *CalculateCommonKey* se nachází ve třídě *DiffieHellman*, která je umístěna v adresáři *Encryption*.

```
1  public BigInteger CalculateCommonKey(
2      BigInteger primeNumber,
3      BigInteger peerPublicKey,
4      BigInteger secretKey)
5  {
6      return (peerPublicKey ^ secretKey) % primeNumber;
7  }
```

Ukázka kódu 23: Výpočet společného klíče v protokolu Diffie-Hellman, Zdroj: [autor]

8.4.3 Šifrování zpráv

Vyvíjená aplikace bude pro symetrické šifrování využívat algoritmus *AES-256*. Implementace algoritmu *AES* je součástí standardní knihovny platformy .NET. Pro využití varianty *AES-256* je nutné dodat klíč o velikosti 256 bitů. To je zajištěno využitím metody *HashCommonKey*, která je popsaná v ukázce kódu č. 24. Ta jako vstupní parametr přijímá společný klíč získaný po dokončení výměny klíčů pomocí algoritmu Diffie-Hellman. Hašovací funkce *SHA-256* převede společný klíč na klíč o délce 256 bitů. Hlavním důvodem použití hašování je fakt, že výstup bude vždy 256 bitů dlouhý, a to nehledě na to, jak dlouhý byl společný klíč, který do metody vstupoval.

```
1 public byte[] HashCommonKey(BigInteger commonKey)
2 {
3     using (var sha256 = SHA256.Create())
4     {
5         return sha256.ComputeHash(commonKey.ToByteArray());
6     }
7 }
```

Ukázka kódu 24: Hašování společného klíče pomocí funkce *SHA-256*, Zdroj: [autor]

Po výpočtu hashe ze společného klíče je možné přistoupit k samotnému šifrování zpráv. V adresáři *Encryption* se nachází třída *SymmetricEncryptor*, jenž obsahuje metody pro šifrování a dešifrování. Šifrování je možné provést pomocí metody *EncryptMessage* zobrazené v ukázce kódu č. 25. Jako vstupní parametr přijímá šifrovací klíč v podobě pole bajtů a zprávu, která má být zašifrována. Tělo metody začíná převedením textové zprávy na pole bajtů a vytvořením instance třídy *AES* sloužící k šifrování. Následně dojde k nastavení vlastností *Mode* a *Key*. Dalším krokem je vytvoření „inicializačního vektoru“ zavoláním metody *GenerateIV* na instanci třídy *AES*. Tento vektor přináší do procesu šifrování náhodnost, jenž zvýší jeho bezpečnost. Po dokončení nastavení algoritmu je potřeba získat instanci objektu implementujícího rozhraní *ICryptoTransform*. Nad tímto objektem se zavolá metoda *TransformFinalBlock*, které se jako parametry dodají zpráva v podobě pole bajtů, pozice v poli, kde má být šifrování zahájeno, a množství prvků určených k zašifrování. V závěru metody je navrácena n-tice se zašifrovanou zprávou a inicializačním vektorem, jenž byl při šifrování použit.

Metoda pro dešifrování zprávy *DecryptMessage* pak vykonává opačný proces k šifrování. Jako vstupní parametry je potřeba dodat dešifrovací klíč, inicializační vektor a zašifrovanou zprávu. Pro korektní dešifrování zvolené zprávy je nutné, aby byl metodě *DecryptMessage* dodán inicializační vektor, který byl použit při jejím šifrování.

```
public (byte[] EncryptedMessage, byte[] IV) EncryptMessage(
    byte[] encryptionKey,
    string message)
{
    var encodedMessage = Encoding.UTF8.GetBytes(message);

    using (Aes aes = Aes.Create())
    {
        aes.Mode = CipherMode.CBC;
        aes.Key = encryptionKey;

        aes.GenerateIV();

        using (var cryptoTransform = aes.CreateEncryptor())
        {
            var encryptedMessage = cryptoTransform
                .TransformFinalBlock(
                    encodedMessage,
                    0,
                    encodedMessage.Length);

            return (encryptedMessage, aes.IV);
        }
    }
}
```

Ukázka kódu 25: Šifrování zprávy pomocí algoritmu *AES-256*, Zdroj: [autor]

8.4.4 Průběh komunikace

Před zahájením konverzace si zařízení uživatelů musí vyměnit několik zpráv. Ve chvíli, kdy je vytvořena nová konverzace, jsou vygenerovány soukromý a veřejný klíč. V první zprávě dojde k odeslání veřejného klíče uživatele a čísel použitých pro generování. Tato zpráva je označena jako *Initial*. Po jejím přijetí zařízení druhého uživatele vygeneruje své klíče a odešle zpět zprávu typu *Accept* s veřejným klíčem. Po přijetí zprávy zařízením prvního uživatele mají obě strany všechny potřebné informace k výpočtu společného klíče a jejich konverzace může začít. Běžné zprávy v konverzaci nesou označení *Standard*.

9 Výsledky

Předchozí kapitola popsala implementaci klientské a serverové částí aplikace pro decentralizovanou komunikaci. Nyní ještě zbývá řádně otestovat celou aplikaci a popsat případné nedostatky odhalené při testování.

9.1 Příprava testování

Serverová část aplikace bude spuštěna ve dvou nezávislých instancích. Ty se připojí každá k vlastní databázi. Jako testovací prostředí se použije lokální počítač. Na portech 5000 a 5001 dojde ke spuštění serverových aplikací. Porty 5432 a 5433 budou zase vyhrazeny pro databáze.

Jako instance mobilního klienta aplikace se použijí dvě fyzická zařízení. Zástupcem méně výkonných bude smartphone *HMD Nokia 6.1* s operačním systémem *Android 10*. *Realme 9 Pro+* s předinstalovaným operačním systém Android 12 zase zastoupí novější zařízení. Podrobnější parametry obou zařízení je možné vidět v tabulce č. 1.

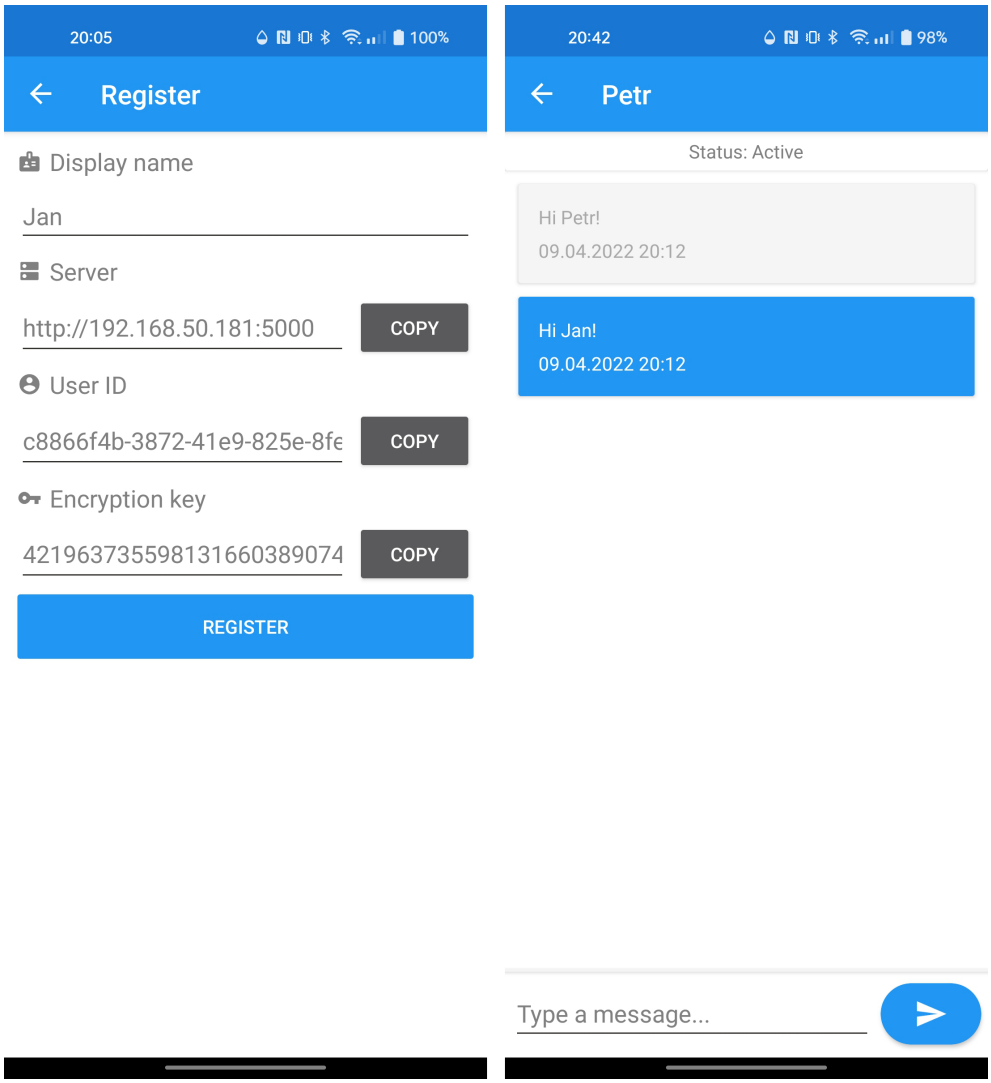
Název	HMD Nokia 6.1	Realme 9 Pro+
OS	Android 10	Android 12
SOC	Snapdragon 630	Dimensity 920
RAM	3 GB	8 GB
Úložiště	32 GB	256 GB
Rozlišení	1080 x 1920 px	1080 x 2400 px

Tabulka 1: Základní specifikace zařízení použitých pro testování, Zdroj: [autor]

Z pohledu fungování aplikace dojde k vyzkoušení registrace a přihlášení k domovskému serveru. Každý z klientů k tomu využije jinou instanci serverové aplikace. Velmi důležitou částí bude testování systému výměny zpráv mezi uživateli, v rámci kterého dojde také k ověření správného fungování šifrování a dešifrování zpráv. Testování uživatelského rozhraní proběhne vyzkoušením jednotlivých grafických prvků všech obrazovek vyvinuté aplikace.

9.2 Výsledky testování

Provedené testy ukázaly, že základní funkcionalita vyvinuté ukázkové aplikace funguje dle očekávání. Zároveň však testování odhalilo další možnosti rozšíření. Tato rozšíření nemají vliv na základní funkcionalitu aplikace, ale jejich implementace by zvýšila komfort při používání. Bylo by např. vhodné přidat funkce pro import a export lokální databáze. Takováto schopnost by našla využití např. při přenosu všech dosavadních zpráv a konverzací do nového mobilního zařízení. Další zajímavé rozšíření představuje přidání možnosti využívat více uživatelských profilů zároveň. Díky tomu může být umožněno, aby různé konverzace procházely skrze různé domovské servery. Vizuální ukázku korektního fungování registrace, zahájení konverzace a posílání zpráv ukazuje obrázek č. 18.



Obrázek 18: Výsledky testování, Zdroj: [autor]

10 Závěr

Cílem této diplomové práce bylo popsat tvorbu decentralizované komunikační aplikace. Zároveň mělo dojít k návrhu a implementaci ukázkové aplikace využívající představených principů.

Nejprve se práce věnovala seznámení s centralizovanými a decentralizovanými službami a poukázala na výhody a nevýhody jednotlivých řešení. Následující kapitola popsala a porovnala symetrické a asymetrické šifrování. Zároveň tato kapitola seznámila čtenáře s problematikou hašovacích funkcí. Další dvě kapitoly se věnovaly vývoji mobilních aplikací a představení technologií, které byly použity pro vývoj ukázkové aplikace. V sedmé kapitole došlo k analýze a návrhu ukázkové aplikace. Osmá kapitola popsala implementaci navržené aplikace. Závěr práce se zabýval otestování vyvinuté aplikace.

Průběh implementace se obešel bez zásadnějších problémů. Jediný menší problém nastal při implementaci kódu, který by mohl být sdílen mezi serverovou a klientskou částí aplikace. Framework Xamarin.Forms bohužel nepodporuje nejnovější specifikaci knihoven kódu .NET, takže např. kód obstarávající šifrování se mezi serverovou a klientskou aplikací mírně liší. Na funkčnost výsledné aplikace však tyto rozdílné implementace vliv nemají.

Testování vyvinuté aplikace potvrdilo, že navrhované řešení je funkční. Zároveň však poukázalo na možná vylepšení. Získané poznatky a samotnou aplikaci lze použít jako výchozí bod při tvorbě aplikací podobného charakteru.

Literatura

[1] Nguyen, M. H.; Gruber, J.; Fuchs, J.; et al. Changes in Digital Communication During the COVID-19 Global Pandemic: Implications for Digital Inequality and Future Research. *Social Media + Society*, díl 6, č. 3, Červenec 2020: str. 205630512094825, ISSN 2056-3051, 2056-3051, doi:10.1177/2056305120948255. Dostupné z: http://journals.sagepub.com/doi/10.1177/2056305120948255

[2] Montag, C.; Błaszkiewicz, K.; Sariyska, R.; et al. Smartphone usage in the 21st century: who is active on WhatsApp? *BMC Research Notes*, díl 8, č. 1, Prosinec 2015: str. 331, ISSN 1756-0500, doi:10.1186/s13104-015-1280-z. Dostupné z: http://www.biomedcentral.com/1756-0500/8/331

[3] Urman, A.; Ho, J. C.-t.; Katz, S. Analyzing protest mobilization on Telegram: The case of 2019 Anti-Extradition Bill movement in Hong Kong. *PLOS ONE*, díl 16, č. 10, Říjen 2021: str. e0256675, ISSN 1932-6203, doi:10.1371/journal.pone.0256675. Dostupné z: https://dx.plos.org/10.1371/journal.pone.0256675

[4] Privacy Guides. Real-Time Communication. Říjen 2021. Dostupné z: https://www.privacyguides.org/real-time-communication/

[5] Statista. Most popular global mobile messenger apps as of October 2021, based on number of monthly active users. Říjen 2021. Dostupné z: https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/

[6] Varvello, M.; Steiner, M. Traffic Localization for DHT-Based BitTorrent Networks. V *NETWORKING 2011*, upraveno J. Domingo-Pascual; P. Manzoni; S. Palazzo; A. Pont; C. Scoglio, Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2011, ISBN 978-3-642-20798-3, str. 40–53, doi:10.1007/978-3-642-20798-3_4.

[7] Sublime Software Ltd. How it works - Briar. 2021. Dostupné z: https://briarproject.org/how-it-works/

[8] Secure Scuttlebutt Consortium. Scuttlebutt - About. 2021. Dostupné z: https://scuttlebutt.nz/about/

[9] BitTorrent Inc. Bleep Now Publicly Available Across All Major Platforms. Květen 2015. Dostupné z: https://www.bittorrent.com/blog/2015/05/12/bleep-private-messenger-now-on-all-major-platforms/

[10] Jenkins, N.; Newman, C. The JSON Meta Application Protocol (JMAP) for Mail. Request for Comments RFC 8621, Internet Engineering Task Force, Srpen 2019, doi:10.17487/RFC8621, počet stran: 108. Dostupné z: https://datatracker.ietf.org/doc/rfc8621

[11] Matrix.org Foundation. Matrix Specification. 2022. Dostupné z: https://spec.matrix.org/v1.2/

[12] Amirová, K. Úvod do kryptografie - Významnost Kryptografie. 2007. Dostupné z: https://sifrovani.fd.cvut.cz/index.html

[13] Amirová, K. Úvod do kryptografie - Oblasti využití a zhodnocení metod šifrování. 2007. Dostupné z: https://sifrovani.fd.cvut.cz/vyuz_zhod.html

[14] Amirová, K. Úvod do kryptografie - Symetrické algoritmy. 2007. Dostupné z: https://sifrovani.fd.cvut.cz/syme_algo.html

[15] Amirová, K. Úvod do kryptografie - Blokové šifry. 2007. Dostupné z: https://sifrovani.fd.cvut.cz/blok_sifr.html

[16] Amirová, K. Úvod do kryptografie - Proudové šifry. 2007. Dostupné z: https://sifrovani.fd.cvut.cz/prou_sifr.html

[17] Salomon, D. Block Ciphers. V *Coding for Data and Computer Communications*, upraveno D. Salomon, Boston, MA: Springer US, 2005, ISBN 978-0-387-23804-3, str. 289–309, doi:10.1007/0-387-23804-2_11. Dostupné z: https://doi.org/10.1007/0-387-23804-2_11

[18] Yassein, M. B.; Aljawarneh, S.; Qawasmeh, E.; et al. Comprehensive study of symmetric key and asymmetric key encryption algorithms. V *2017 International Conference on Engineering and Technology (ICET)*, Srpen 2017, str. 1–7, doi: 10.1109/ICEngTechnol.2017.8308215.

[19] Mustafeez, A. Z. What is the AES algorithm? Dostupné z: https://www.educative.io/edpresso/what-is-the-aes-algorithm

[20] Pound, M. AES Explained (Advanced Encryption Standard) - Computerphile. Listopad 2019. Dostupné z: https://www.youtube.com/watch?v=O4xNJsjtN6E

[21] Amirová, K. Úvod do kryptografie - Asymetrické algoritmy. 2007. Dostupné z: https://sifrovani.fd.cvut.cz/asym_algo.html

[22] Federal Office for Information Security. Technical Guideline BSI TR-03111,„Elliptic Curve Cryptography“. 2018. Dostupné z: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TR03111/BSI-TR-03111_V-2-1_pdf.pdf

[23] Katz, A.; Ng, A.; Bourg, P.; et al. RSA Encryption - Brilliant Math & Science. Dostupné z: https://brilliant.org/wiki/rsa-encryption/

[24] Robshaw, M. J. B. One-Way Function. V *Encyclopedia of Cryptography and Security*, upraveno H. C. A. van Tilborg; S. Jajodia, Boston, MA: Springer US, 2011, ISBN 978-1-4419-5906-5, str. 887–888, doi:10.1007/978-1-4419-5906-5_467. Dostupné z: https://doi.org/10.1007/978-1-4419-5906-5_467

[25] Švecová, H. OBDAI - Hašovací funkce. 2021.

[26] Pound, M. SHA: Secure Hashing Algorithm - Computerphile. Duben 2017. Dostupné z: https://www.youtube.com/watch?v=DMtFhACPnTY

[27] Mendel, F. *Analysis of Cryptographic Hash Functions*. Disertační práce, Graz University of Technology, 2010.

[28] Laricchia, F. Mobile OS market share 2021. Únor 2022. Dostupné z: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

[29] Google. Platform Architecture. Srpen 2021. Dostupné z: https://developer.android.com/guide/platform

[30] Sheriff, M. CS 4720 - Mobile Application Development. Dostupné z: https://cs4720.cs.virginia.edu/slides/CS4720-MAD-iOSArchitecture.pdf

[31] Linux Foundation. About Tizen. 2012. Dostupné z: https://www.tizen.org/about

[32] Jolla Ltd. SailfishOS - Info. 2021. Dostupné z: https://sailfishos.org/info/

[33] Kai OS Technologies. Explore KaiOS - Features. 2021. Dostupné z: https://www.kaiostech.com/explore/

[34] Technologies, K. O. KaiOS - FAQs. 2021. Dostupné z: https://www.kaiostech.com/faq/

[35] Winer, D. Android's commitment to Kotlin. Prosinec 2019. Dostupné z: https://android-developers.googleblog.com/2019/12/androids-commitment-to-kotlin.html

[36] Apple Inc. About Objective-C. Září 2014. Dostupné z: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html

[37] Apple Inc. Swift - Apple Developer. 2022. Dostupné z: https://developer.apple.com/swift/

[38] Meta Platforms. Android Native UI Components · React Native. Leden 2022. Dostupné z: https://reactnative.dev/docs/native-components-android

[39] Britch, D. Controls Reference - Xamarin. Srpen 2021. Dostupné z: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/controls/

[40] The Apache Software Foundation. Architectural overview of Cordova platform - Apache Cordova. Říjen 2018. Dostupné z: https://cordova.apache.org/docs/en/latest/guide/overview/

[41] Windmill, E.; Rischpater, R. *Flutter in action*. Shelter Island, NY: Manning Publications Co, 2020, ISBN 978-1-61729-614-7, oCLC: on1089418348.

[42] Japikse, P.; Troelsen, A. *Pro C# 9 with .NET 5 Foundational Principles and Practices in Programming*. S.l.: Apress, 2021, ISBN 978-1-4842-6939-8, oCLC: 1262670731. Dostupné z: https://learning.oreilly.com/library/view/~/9781484269398/

[43] Lander, R. Announcing .NET 6 - The Fastest .NET Yet. Listopad 2021. Dostupné z: https://devblogs.microsoft.com/dotnet/announcing-net-6/

[44] Lock, A. *ASP.NET Core in action*. Shelter Island, NY: Manning, second edition edice, 2021, ISBN 978-1-61729-830-1.

[45] Hunter, S. Introducing .NET Multi-platform App UI - .NET Blog. Květen 2020. Dostupné z: https://devblogs.microsoft.com/dotnet/introducing-net-multi-platform-app-ui/

[46] Hermes, D.; Mazloumi, N. *Building Xamarin.Forms Mobile Apps Using XAML: Mobile Cross-Platform XAML and Xamarin.Forms Fundamentals*. Berkeley, CA: Apress, 2019, ISBN 978-1-4842-4029-8 978-1-4842-4030-4, doi: 10.1007/978-1-4842-4030-4. Dostupné z: http://link.springer.com/10.1007/978-1-4842-4030-4

[47] Obe, R. O.; Hsu, L. S. *PostgreSQL: up and running: a practical guide to the advanced open source database*. Sebastopol, CA: O'Reilly Media, Inc, third edition edice, 2017, ISBN 978-1-4919-6341-8, oCLC: ocn975362904.

[48] Maurício, D.; Cassiano, S.; Leonardo, N. LiteDB - A .NET embedded NoSQL database. 2022. Dostupné z: http://www.litedb.org

[49] Software Freedom Conservancy. About Git. 2022. Dostupné z: https://git-scm.com/about

[50] Poulton, N. *Docker Deep Dive*. Poulton, Nigel, 2019, oCLC: 1137813494. Dostupné z: https://proxy.library.cornell.edu/sso/skillport?context=147451

[51] Nielsen, H.; Fielding, R. T.; Berners-Lee, T. Hypertext Transfer Protocol – HTTP/1.0. Request for Comments RFC 1945, Internet Engineering Task Force, Květen 1996, doi:10.17487/RFC1945, počet stran: 60. Dostupné z: https://datatracker.ietf.org/doc/rfc1945

[52] Clifton, I. G. *Android user interface design implementing material design for developers*. New York: Addison-Wesley, 2016, ISBN 978-0-13-419194-2, oCLC: 933274705. Dostupné z: http://proquestcombo.safaribooksonline.com/9780134191942

[53] Grove, R. F.; Ozkan, E. The MVC-web design pattern. V *Proceedings of the 7th International Conference on Web Information Systems and Technologies*, Noordwijkerhout, Netherlands: SciTePress - Science and Technology Publications, 2011, ISBN 978-989-8425-51-5, str. 127–130, doi:10.5220/0003296901270130. Dostupné z: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0003296901270130

[54] Garofalo, R. *Building enterprise applications with Windows Presentation Foundation and the Model View ViewModel pattern*. Redmond, Wash: Microsoft, 2011, ISBN 978-0-7356-5092-3.

[55] Pound, M. Secret Key Exchange (Diffie-Hellman). Prosinec 2017. Dostupné z: https://www.youtube.com/watch?v=NmM9HA2MQGI

[56] Kivinen, T.; Kojo, M. More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE). Technical report RFC3526, RFC Editor, Květen 2003, doi:10.17487/rfc3526. Dostupné z: https://www.rfc-editor.org/info/rfc3526

[57] Pound, M. Diffie Hellman -the Mathematics bit- Computerphile. Prosinec 2017. Dostupné z: https://www.youtube.com/watch?v=Yjrfm_oRO0w

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání diplomové práce

Autor: **Bc. Tomáš Malík**

Studium: I2000049

Studijní program: N1802 Aplikovaná informatika

Studijní obor: Aplikovaná informatika

Název diplomové práce: **Aplikace pro decentralizovanou komunikaci**

Název diplomové práce AJ: Application for decentralized communication

Cíl, metody, literatura, předpoklady:

Cíl: Popsat problematiku tvorby aplikace pro decentralizovanou komunikaci a vytvořit ukázkovou aplikaci.

Osnova:

1. Úvod
2. Centralizované a decentralizované služby
3. Představení použitých technologií
4. Analýza a návrh aplikace
5. Popis implementace aplikace
6. Výsledky
7. Závěr

Garantující pracoviště: Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D.

Datum zadání závěrečné práce: 26.1.2021