Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

# Bakalářská práce

# Bezpečné předávání zpráv s využitím Blockchainu

Plzeň 2021 Martin Procházka

ZÁPADOČESKÁ UNIVERZITA V PLZNI

Fakulta aplikovaných věd

Akademický rok: 2020/2021

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

| | |
|---|---|
| Jméno a příjmení: | **Martin PROCHÁZKA** |
| Osobní číslo: | **A17B0333P** |
| Studijní program: | **B3902 Inženýrská informatika** |
| Studijní obor: | **Informatika** |
| Téma práce: | **Bezpečné předávání zpráv s využitím Blockchainu** |
| Zadávající katedra: | **Katedra informatiky a výpočetní techniky** |

## Zásady pro vypracování

1. Prostudujte metody bezpečného předávání zpráv na internetu a principy distribuovaného Blockchainu.
2. Navrhněte mechanismus pro bezpečné předávání zpráv s využitím Blockchainu.
3. Implementujte návrh a vytvořte adekvátní demonstrační aplikaci.
4. Kriticky zhodnoťte dosažené výsledky. Zejména se zaměřte na analýzu potenciálních bezpečnostních rizik.

Rozsah bakalářské práce: **doporuč. 30 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování bakalářské práce: **tištěná**

Seznam doporučené literatury:

Dodá vedoucí bakalářské práce.

Vedoucí bakalářské práce: **Ing. Miloslav Konopík, Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání bakalářské práce: **5. října 2020**
Termín odevzdání bakalářské práce: **6. května 2021**

L.S.

**Doc. Dr. Ing. Vlasta Radová**
děkanka

**Doc. Ing. Přemysl Brada, MSc., Ph.D.**
vedoucí katedry

V Plzni dne 26. října 2020

# Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 12. května 2021

Martin Procházka

# Abstract

First part of this paper presents systems for secure comunication currently in use and principles of blockchain architecture as used in Ethereum network. Based on these findings algorithms using blockchain with the goal of securing communication are proposed. At last a concrete system is proposed and implemented.



# Abstrakt

První část práce seznámí čtenáře s některými používanými systémy pro bezpečnou komunikaci, principy blockchainu a sítě Ethereum. Na základě těchto poznatků jsou diskutovány možnosti využití blockchainu při zabezpečení komunikace. Součástí je návrh konkrétního systému a jeho implementace.

# Obsah

# 1 Úvod

Na zabezpečení komunikace v prostředí internetu i mimo něj je v dnešní době kladen obrovský důraz. Následky prozrazení obsahu komunikace mohou být v mnohých případech těžko napravitelné až, například v případě disidentů žijících v totalitních režimech, dokonce fatální. Typickým problémem vzdálené komunikace je také možnost záměny identity, ať už neúmyslné, nebo úmyslné. Z těchto důvodů bylo vyvinuto množství kryptografických primitiv, protokolů a systémů, které se snaží tyto problémy když ne odstratnit, tak alespoň minimalizovat. Jedním z dosud hojně používaných standardů je Pretty Good Privacy, známý pod zkratkou PGP, který staví na asymetrické kryptografii a sítích důvěry. Pro zabezpečení komunikace mezi uživatelem a webovým serverem je dnes výhradně používán protokol TSL, který řeší přenos důvery hierarchicky na bázi certifikačních autorit. Poměrně novým hráčem je protokol Signal využívaný zejména pro instant messaging.

Velkým milníkem ve světě kryptografie se v poslední dekádě stal masivní vzestup technologií postavených na blockchainu. Jedním takovým významným ekosystémem je Ethereum se svojí kryptoměnou Ether. Síla Etherea oproti například Bitcoinu je možnost vytvářejí takzvaných „chytrých kontraktů“, což jsou programy, které lze na této síti spouštět a uživatelé s nimi interagují pomocí transakcí. Takto na síti vznikají decentralizované aplikace.

Cílem této práce bude podrobnější seznámení čtenáře s aktuální stavem a technologiemi používanými při bezpečné komunikaci v prostředí internetu a principy fungování sítě Ethereum. Dále budou prozkoumány možnosti využití Etherea a chytrých kontraktů v kontextu bezpečné komunikace. Dosažené výsledky budou porovnány s již existujícími řešeními nevyužívajícími blockchain. Na závěr bude implementována demonstrace navrženého systému.

Pro porozumění textu je předpokládána obecná znalost kryptografie. Práce se bude také z části dotýkat programování chytrých kontraktů v jazyce Solidity, tyto části předpokládají znalost principů objektově orientovaného programování, znalost libovolného jazyka s „Céčkovskou“ syntaxí (C/C++, Java, JavaScript...) je výhodou.

# 2 Kryptografické vlastnosti protokolů

Při návrhu protokolu pro bezpečné předávání zpráv je nejprve nutné specifikovat jaké vlastnosti by takový protokol měl mít. V následujících řádcích budou tedy definovány kryptografické vlastnosti v tomto kontextu považovány za žádoucí, které budou uvažovány při analýze protokolů. K nastínění scénářů jsou dle zvyklostí použity jména Alice, Bob (participanti komunikace) a Eve (záškodník).

## 2.1 Základní požadavky

Aby se dalo hovořit o bezpečné komunikaci, musí protokol nabízet následující vlastnosti.

**Důvěrnost (*confidentiality*)** Důvěrnost zajišťuje, že komunikace může být čtena pouze jejími účastníky, čehož je dosaženo použitím šifry. Potřeba důvěrné komunikace historicky zavdala vzniku klasické kryptografie a steganografie. Do jisté míry je to ta nejdůležitější vlastnost, protože o udržení obsahu zprávy v tajnosti jde v prvé řadě.

**Integrita (*integrity*)** Garance integrity znamená, že změna odesílané zprávy (ať už úmyslná v rámci útoku, nebo způsobena náhodnout chybou) je ne nutně opravitelná, ale v každém případě detekovatelná. Prostředky pro zajištění integrity zahrnují kontrolní součty (ne příliš efektivní vůči úmyslným změnám), elektronické podpisy a autentikační kódy – Message authentication code (MAC).

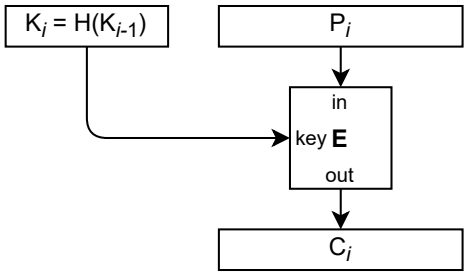
**Autentikace (*authentication*)** Uživatelé systému mají nějakou identitu, ta může mít více složek, například uživatelské jméno, různé druhy adres, klíče. Samozřejmě by bylo nerozumné považovat představenou identitu za pravdivou pouze na základě dobrého slova uživatele, který jí předkládá za svou. Zde přichází na řadu autentikace, tedy proces ověření pravdivosti této identity a dokázání vlastnictví překládaných adres nebo klíčů.

**Odmítnutelnost (*deniability*)** je vlastnost autentikovaného přenosu a jejím opakem je neodmítnutelnost. Pokud Alice kontaktuje Boba pomocí protokolu, jehož zprávy jsou neodmítnutelné, pak Bob dokáže třetí straně předložit kryptografický důkaz, že zpráva, kterou obdržel, pochází od Alice. V případě odmítnutelnosti Bob ví, že komunikuje s Alicí, ale dokázat to nemůže, tato vlastnost je v případě soukromých konverzací žádoucí. Dosáhnout jí lze použitím autentikačních kódů v kombinaci s protokolem výměny klíčů namísto elektronických podpisů – ty jsou naopak vhodným prostředkem pro dosažení neodmítnutelnosti[6].

## 2.2 Zabezpečení důvěrnosti

Protokol vyznačující se zmíněnými vlastnostmi – důvěrnost, integrita, autentikace a odmítnutelnost, by se v ideálním světě dal považovat za bezpečný. Obsah zprávy je skryt, nelze měnit, obě strany ví s kým komunikují, avšak důkaz o tom třetí straně podat nemohou. Problém nastane ve chvíli, kdy útočník necílí na protokol jako takový, nýbrž na účastníka komunikace tím, že kompromituje použité klíče. Pokud je v takové situaci zkoumán vliv úniku klíčů na důvěrnost komunikace, pak lze dojít k dvěma typů zabezpečení.

**Dopředné zabezpečení (*forward secrecy*)** Pokud únik stavových proměnných protokolu včetně klíčů v čase $t$ neohrozí důvernost komunikace která proběhla v čase $t' < t$ pak lze mluvit o dopředně zabezpečeném protokolu. Jednoduchý příklad takového schématu využívajícího pouze symetrickou šifru a hash funkci lze vidět na obrázku 2.1. Snadno si zde lze všimnout, že po odhalení konkrétního klíče může útočník generovat následné klíče, avšak ke klíčům předchozím se nedostane, protože k hash funkci není známa funkce inverzní.




Obrázek 2.1: Schéma se symetrickou šifrou splňující podmínky dopředného zabezpečení.

**Zpětné zabezpečení (*backward/future secrecy*)** Zpětné zabezpečení funguje opačným směrem. – Takto zabezpečený protokol se z úniku dokáže zotavit a obnovit důvěrnost pozdější komunikace. Příkladem takového protokolu je Signal, resp. Double Ratchet algoritmus, který bude představen v sekci 3.3. Bezpečný protokol by měl nabízet oba typy zabezpečení s co možná nejmenší granularitou. V ideálním systému vyzrazení klíče způsobí ztrátu důvěrnosti pouze pro jednu konkrétní zprávu, která byla tímto klíčem šifrována.

# 3 Existující řešení

## 3.1 OpenPGP

Pretty Good Privacy (PGP)[13] využívá kombinace asymetrické a konvenční kryptografie za účelem zabezpečení elektronické pošty a dat obecně. V původní verzi byl protokol navržen Philipem Zimmermannem roku 1991. OpenPGP je otevřený standard založený na PGP 5, popsán je v RFC4880 [5]. Jako rozšířenou implementaci protokolu uveďme Gnu Privacy Guard (GPG).

### 3.1.1 Autentizace

Každý uživatel v PGP je identifikován svým veřejným klíčem a identifikátorem (*user-id*), který se typicky skládá z e-mailové adresy a jména. Problém který je třeba vyřešit je prokázání, že člověk s proklamovanou identitou je vlastníkem klíče. Jako infrastruktura pro řešení tohoto problému zde slouží síť důvěry (*web of trust*).

Síť důvěry funguje na principu vzájemného ověřování mezi jednotlivci, důvěra je předána ve formě podpisu veřejného klíče (*signature trust*). Pokud Alice dokáže ověřit, že Bob je skutečným vlastníkem klíče, kterým se reprezentuje, pak Alice tento klíč podepíše a podpis předá Bobovi. Bob následně tento podpis přidá na svůj certifikát.

Tímto vzniká orientovaný graf, kde vrchol je konkrétní identita a hrana představuje podpis klíče. Navíc zde figuruje druhý typ důvěry – důvěra ve vlastníka (*owner trust*). Na základě individuální důvěry v uzly na cestě v grafu ke zkoumané identitě je vypočítána její platnost (*validity*). Platnost reprezentuje úroveň důvěry v to, že vlastník klíče je ten za koho se vydává. Důvěra ve vlastníka má několik diskrétních úrovní, od absolutní nedůvěry po absolutní důvěru, a vyjadřuje individuální mírů důvěry v důkladnost s jakou dotyčný ověřuje identitu před podepsáním cizího klíče. Například v situaci, kdy Alice chce komunikovat s Bobem a vidí, že Bobův veřejný klíč je podepsán pouze Eve, které však Alice nedůvěřuje, pak Alice nemůže Bobův klíč považovat za platný. Možným řešením je pro Alici osobně Bobovu identitu ověřit a podepsat jeho klíč, pokud Alice vkládá maximální důvěru ve svou schopnost Bobovu identitu ověřit, pak již lze považovat jeho klíč za zcela jistě platný. Alternativně pokud Alice důvěřuje Charliemu a Charlie podepíše Bobův klíč, pak Alice opět může považovat Bobův klíč za platný. Je

vhodné podotknout, že platnost klíče není absolutní, Alice může považovat Bobův klíč za platný, ale pro Charlieho může být Bobův klíč neplatný. Stejně tak není platnost symetrická, Bob nemusí považovat Alicin klíč za platný, i když pro Alici Bobův klíč platný je. Uvedený příklad je velmi jednoduchý, ale i tak dobře nastiňuje způsob, jakým v PGP funguje přenos důvěry, detailnější popis lze najít v odkazované literatuře[2].

Zajímavá vlastnost tohoto modelu je decentralizace a tedy absence centrální autority. Bezpečnost celého systému je však do jisté míry závislá na jeho správném porozumnění, důvěryhodnosti účastníků a schopnosti jejich důvěryhodnost posoudit.

### 3.1.2 Komunikační protokol

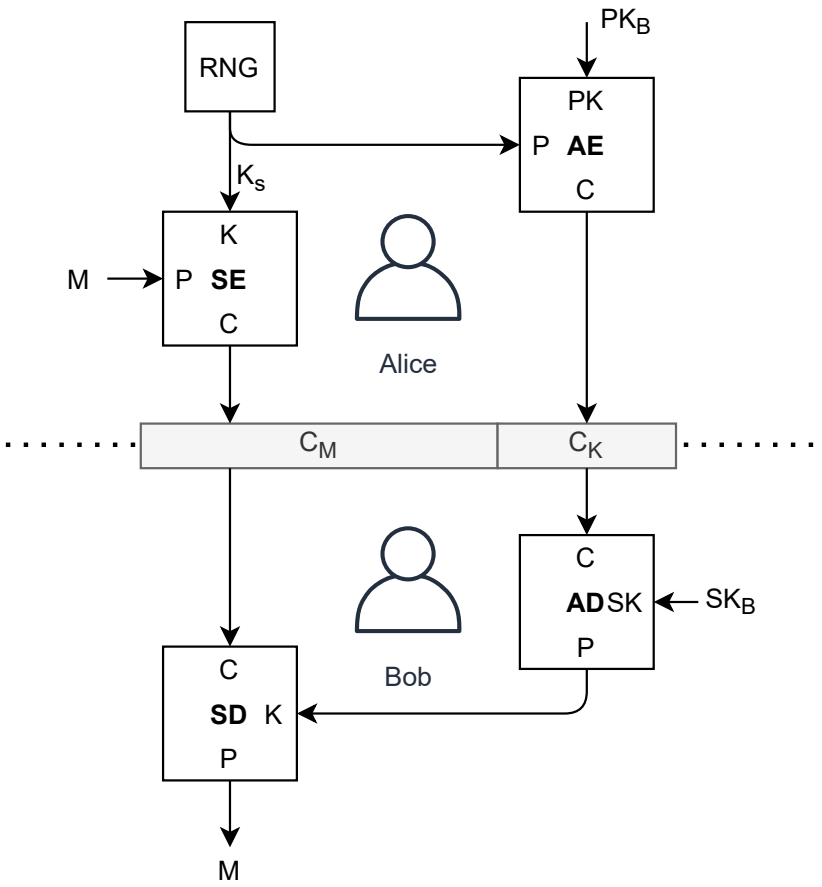
V modelové situaci, kde se Alice snaží kontaktovat Boba bude PGP postupovat následujícím algoritmem.

1. Alice vygeneruje náhodný relační klíč $K_S$ a zašifruje jím zprávu $M$.
2. Relační klíč je zašifrován Bobovým veřejným klíčem $PK_B$.
3. Oba šifrové texty $C_M$ a $C_K$ jsou předány Bobovi.
4. Bob dešifruje relační klíč $K_S$.
5. Relačním klíčem $K_S$ dešifruje šifrový text Aliciny zprávy $C_M$.

Graficky je tento proces znázorněn na obrázku 3.1. PGP má ve své specifikaci také možnost komprese a na výsledný šifrový text aplikováno *binary-to-text* kódování. Tato skutečnost je zde pro jednoduchost a nízkou relevanci zanedbána.

Ačkoliv je tento algoritmus, za předpokladu neprolomitelnosti použitých šifer, velmi robustní, slabina spočívá ve způsobu použití klíčů. Zde je patrné, že v případě, kdy se Bobův privátní klíč $SK_B$ stane z nějakého důvodu klíčem veřejným, veškerá historická komunikace využívající patřičný pár klíčů rázem pozbývá důvěrnosti. Totéž platí i pro komunikaci budoucí, pakliže je použit stejný pár klíčů. Z toho důvodu nelze říci, že by PGP poskytovalo dopředné ani zpětné zabezpečení.

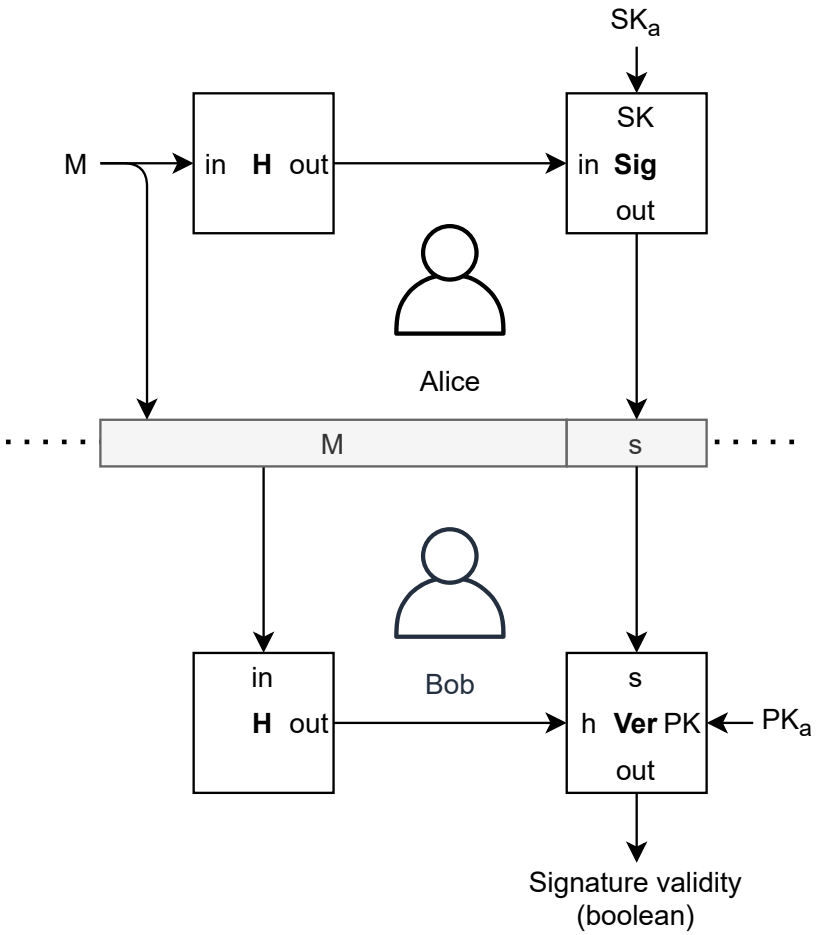
Zároveň stojí za povšimnutí, že takto konstruovaná zpráva není nijak autentikovaná a tudíž Bob jako příjemce se může pouze domnívat, kdo zprávu sestrojil. PGP toto řeší elektronickým podpisem, který se ke zprávě přiloží. Pokud Alice posílá Bobovi autentikovanou zprávu, postupuje následovně.




Obrázek 3.1: Proces šifrování zprávy $M$ v OpenPGP, odeslán je šifrový text zprávy $C_M$ spolu s šifrovaným relačním klíčem $C_K$. Význam použitý bloků je: Symetric encryption (SE) – symetrická šifra, vstupem je klíč $K$ a otevřený text $P$, výstupem je šifrový text $C$. Symetric decryption (SD) – šifrový text $C$ symetricky dešifruje. Asymmetric encryption (AE) – asymetricky šifruje Bobovým veřejným klíčem $PK_B$. Asymmetric decryption (AD) – dešifruje asymetrickou šifru Bobovým privátním klíčem $SK_B$. Random number generator (RNG) – zdroj náhodných dat.

1. Alice vyvoří hash zprávy *H(M)*, kterou se snaží autentikovat.
2. Hash je podepsán Aliciným klíčem $SK_A$.
3. Zpráva $M$ spolu s jejím podpisem $s = Sig(H(M), SK_A)$ je předána Bobovi.
4. Bob stejně jako Alice napočte hash zprávy *H(M)*.
5. Bob ověří hash zprávy *H(M)* proti podpisu s použitím Alicina veřejného klíče $PK_A$.
6. Pokud je podpis platný, zpráva je autentikována.

Využití podpisů tímto způsobem vynucuje neodmítnutelný přenos, v kontextu bezpečné komunikace je toto ovšem nežádoucí vlastnost. Z tohoto důvodu a již zmíněné absenci dopředného i zpětného zabezpečení není PGP, ačkoliv řeší stejný problém, vhodným kandidátem na potenciální blockchainovou reimplementaci.




Obrázek 3.2: Podpis a ověření zprávy v OpenPGP.

### 3.1.3 Bezpečnostní analýza

PGP nabízí důvěrnost a garantuje integritu komunikace, to je dáno použitím šifry a elektronického podpisu. Uživatelé jsou autentikováni na základě sítě důvěry, tento systém je funkční a ověřený časem. Problematická síť důvěry začíná být až z pohledu použitelnosti pro nezasvěceného uživatele.

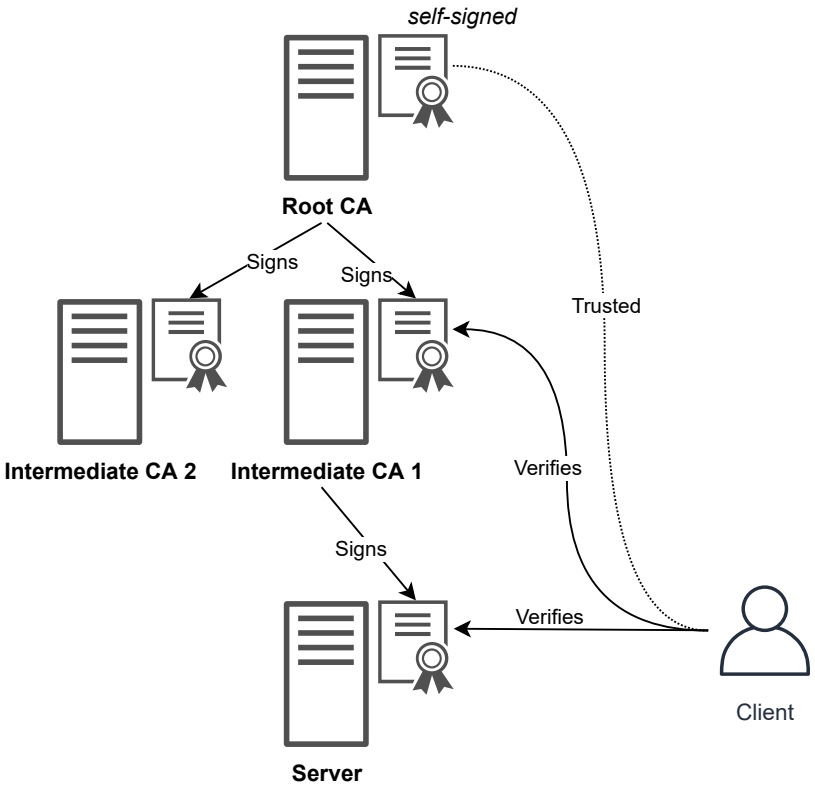
Jako závažný nedostatek lze shledat vpodstatě vynucenou neodmítnutelnost, která plyne ze způsobu použití elektronických podpisů. Vynecháním podpisu lze získat odmítnutelnost, ale zároveň se tím ztrácí garance integrity. Jako dalším zásadním problémem je nezabezpečnení důvěrnosti proti úniku privátních klíčů vyplývající ze způsobu výměny symetrických klíčů. Pokud Eve získá Bobův privátní klíč, pak může dešifrovat symetrický klíč a následne dešifrovat veškerou jeho příchozí, historickou i budoucí, komunikaci. Ze zmíněných důvodů se protokol OpenPGP tak jak definován nejeví v současnosti jako vhodný pro úvahy o reimplementaci na blockchainu.

## 3.2 TLS/SSL

Sada protokolů TLS *Transpor Layer Security* je dnes hojně používaný prostředek pro zabezpečení komunikace v architektuře klient-server. Z pohledu samotné komunikace není TLS pro účel této práce příliš zajímavé, protože řeší trochu jiný problém. Důvodem proč je zde TLS zmíněno je způsob jakým je řešena autentizace, a právě to bude obsahem této kapitoly.

Autentizace v TLS stojí na hierarchii certifikačních autorit (CA), které na základě ověření totožnosti subjetu vydávají certifikáty. Certifikát pak kryptograficky prokazuje, že totožnost byla konkrétní CA ověřena. Tento důkaz je implementován na bázi elektronických podpisů.

Certifikační autority se sdružují do hierarchií, přirozeně tak vzniká stromová struktura. Na vrcholu hierarchie stojí kořenová certifikační autorita, o patro níže jsou podřízené certifikační autority. Certifikát žadateli vydávají podřízené certifikační autority. Pokud sledujeme cestu od konkrétního certifikátu k certifikátu kořenové CA, pak se bavíme o řetězci důvěry. Pokud třetí strana má zájem ověřit důvěryhodnost certifikátu, pak ověří platnost všech certifikátů v řetězci důvěry, naznačeno na obrázku 3.3. Certifikát je uznán za platný pokud je při ověřování naraženo na důvěryhodný certifikát, to bude typicky certifikát kořenové CA. Certifikáty obecně uznávaných CA jsou běžně distribuovány jako součást operačního systému a webových prohlížečů. Ověření samotného certifikátu pak spočívá v kontrole podpisu, expirace certifikátu a přítomnosti na revokační listině. Na revokační list je certifikát přidán vydávající CA pokud ho shledá jako neplatný před datem expirace.




Obrázek 3.3: Hierarchie certifikačních autorit se serverem třetí strany, kterému byl vystaven certifikát. Klient před zahájením komunikace se serverem ověřuje řetězec důvěry.

## 3.3 Signal

Signal je instant messaging protokol s koncovým šifrováním využívající architekturu klient-server vyvinutý v roce 2013 organizací Open Whisper Systems. Pro výměnu klíče je využit protokol *Extended Triple Diffie-Hellman*[9], samotná komunikace pak funguje s využitím *Double Ratchet*[7] algoritmu. Na správu relací je použit algoritmus *Sesame*[8]. Jako kryptografická primitiva jsou použity křivky $X25519$, $X488$, symetrická šifra AES-256 v CBC módu a klíčované SHA-256. První dva zmíněné algoritmy budou zjednodušeně popsány v následujících odstavcích detailní specifikaci lze najít v citované literatuře.

### 3.3.1 Extended Triple Diffie-Hellman Handshake

Extended Triple Diffie-Hellman (X3DH) je protokol postavený na Diffieho-Hellmanově výměně klíčů s využitím eliptických křivek (ECDH). Cílem pro-

tokolu je vytvořit společné tajemství mezi vzájemně autektikovanými stranami. Mezi příjemné vlastnosti protokolu patří dopředné zabezpečení, odmítnutelnost a asynchronost[1]. K dosažení asynchronnosti je využito serveru. Tento server se stará o uložení a distribuci veřejných klíčů, ze kterých bude následně derivováno společné tajemství, resp. relační klíč pro komunikaci. Uloženy jsou tři typy klíčů:

**Identity key (IK)** – identitní (dlouhodobý) klíč

**Signed prekey (SPK)** – podepisující (střednědobý) klíč

**One-time prekey (OPK)** – jednorázový klíč

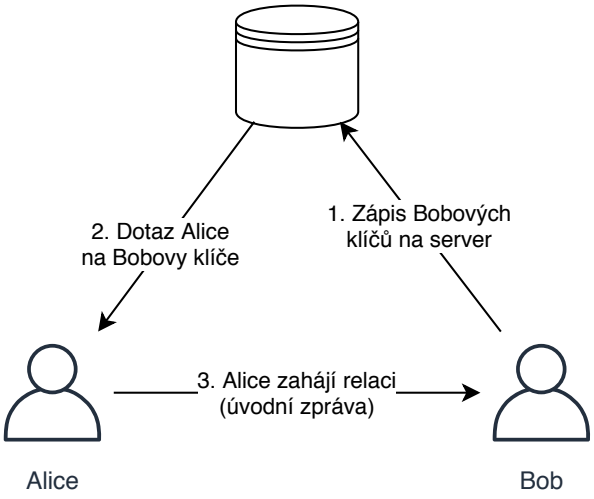
Dlouhodobý a střednědobý klíč má uživatel v každou chvíli pouze jeden. U jednorázových klíčů je situace odlišná, na server se jich nahraje několik (desítky, stovky) a každým průchodem protokolu se jeden „spotřebuje“. V algoritmu se navíc vyskytuje ještě jeden typ klíče, který se na server neposílá $EK$, dočasný (relační) klíč, který lokálně vytvoří strana navazující spojení.

Protokol má tři fáze znázorněné na diagramu 3.4. V první fázi Bob posílá své klíče na server, klíče označme $IK_B$, $SPK_B$, $OPK_B^n$. Index $B$ říká, že se jedná o klíče náležící Bobovi, krátkodobých klíčů může být několik, značeno horním indexem $n$. Navíc Bob pošle svůj identitní klíč podepsaný střednědobým klíčem. Po úvodním nasazení klíčů Bob pouze doplňuje jednorázové klíče a obnovuje střednědobý klíč s podpisem identitního klíče dle potřeby (čím častěji tím lépe).

V druhé fázi chce Alice komunikovat s Bobem, serveru tedy pošle dotaz a jako odpověď dostane sadu klíčů. Součástí této sady je $IK_B$, $SPK_B$, $OPK_B$ a podpis identitního klíče klíčem střednědobým $sig(IK_B, SPK_B)$. Alice provede ověření podpisu a pokračuje pouze pokud je podpis vpořádku. Následně vygeneruje dočasný klíč $EK_A$ a provede čtyřikrát ECDH, dle následujícího schématu.

$$
\begin{aligned}
S_1 &= ECDH(IK_A, SPK_B) \\
S_2 &= ECDH(EK_A, IK_B) \\
S_3 &= ECDH(EK_A, SPK_B) \\
S_4 &= ECDH(EK_A, OPK_B)
\end{aligned}
$$

[1]Derivace společného tajemství není závislá na dostupnosti protistrany, ta může být offline.




Obrázek 3.4: Schéma X3DH protokolu

Tímto má Alice čtyři dílčí tajemství, která použije na vstupu Key Derivation Function (KDF). V tomto případě je žádoucí jako KDF použít dostatečně bezpečnou hash funkci.

$$SK = KDF(S_1||S_2||S_3||S_4)$$

Výstupem je tajný symetrický klíč $SK$, který Alice použije pro šifrování komunikace s Bobem. Protokol lze použít i bez $OPK$, v takovém případě se vynechá $S_4$. Ve třetí fázi Alice pošle Bobovi iniciální zprávu. Její součástí jsou klíče $IK_A$, $EK_A$, $OPK_B$, pomocí nichž Bob provede výpočet $SK$. Následuje datová nálož šifrovaná klíčem $SK$.

### 3.3.2 Double Ratchet Algorithm

Double Ratchet algoritmus slouží ke generování tajných klíčů na základě společného tajemství. Algoritmus navíc poskytuje jak zpětné, tak dopředné zabezpečení, těchto žádoucích vlastností je dosaženo použitím kombinace dvou západkových mechanismů (ratchet). Pro jejich implementaci je použita KDF, která z klíče a vstupních dat generuje pseudonáhodný výstup. K tomuto účelu lze velmi dobře využít klíčovanou hash funkci. Západkový mechanismus (dále jen západka) zavádí část výstupu KDF na vstup pro klíč KDF v následující iteraci, tímto vzniká derivační řetězec (*KDF chain*). Krok západkového mechanismu v tomto kontextu znamená prodloužení derivačního řetězce.

Symetrická západka, původně představena v protokolu SCIMP, slouží ke generování šifrovacích a dešifrovacích klíčů. Alice a Bob mají oba *sending* a *receiving* řetězce, které jsou párově synchronizovány. V případě, že Alice chce poslat Bobovi zprávu, musí nejprve prodloužit svůj odesílající řetězec a zprávu šifrovat vygenerovaným klíčem. Poté, co Bob zprávu přijme, prodlouží svůj přijímací řetězec a tím dostane symetrický klíč shodný s klíčem Alicina odesílajícího řetězce.

Sám o sobě je tento mechanismus nedostatečný, protože jako vstup do KDF je použita konstanta. Pokud Eve získá libovolný klíč v KDF řetězci a použitou konstantu, pak je schopna generovat každý další následující klíč. Důsledkem je jednosměrné prolomení šifrované komunikace.

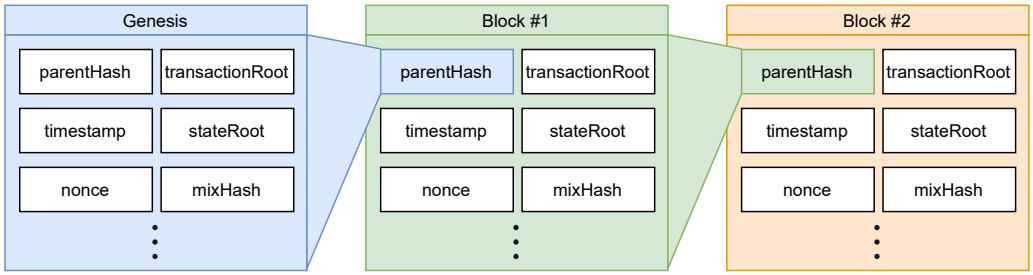
Žádoucí by samozřejmě bylo důsledky tohoto útoku minimalizovat a získat tak protokol, který se po vyzrazení klíčů sám regeneruje. K tomu účelu poslouží Diffieho-Hellmanova západka, původně představena v protokolu OTR[3]. Tato západka, jak už název napovídá, používá jako KDF Diffieho-Hellmanovu výměnu klíčů. Jednorázové klíče nutné k prodloužení tohoto řetězce si aktéři posílají v rámci jejich komunikace. Výstupy tohoto Diffie-Hellman řetězce se použijí střídavě jako vstupy odesílajícího a přijímacího řetězce, čímž dochází k jejich pravidelné regeneraci. Pokud tedy Eve prolomí Bobův přijímací řetězec, pak může číst jeho příchozí poštu pouze do dalšího kroku Diffieho-Hellmanovy západky. Tímto je zajištěno zotavení řetezců v případě odhalení některého z vygenerovaných symetrických klíčů.

# 4 Ethereum

## 4.1 Blockchain

V obecné rovině lze na blockchain nahlížet jako na spojový seznam bloků, kde spoj je realizován hashem předchůdce. Blok je tedy identifikován svým hashem a jediná operace definovaná nad tímto seznamem je přidání následníka. Již přidaný blok není možné měnit, došlo by k rozpadu blockchainu, protože odkaz následníka by s hashem bloku nesouhlasil. Za první významné využití blockchainu lze považovat kryptoměnu Bitcoin[10], kde se s jeho využitím podařilo vyřešit problém dvojité útraty[1] bez centralizované autority.

Tak jak je blockchain využíván v kontextu kryptoměn (obrázek 4.1) se lze bavit o decentralizované datové struktuře – její (alespoň částečná) kopie je součástí každého uzlu sítě. Konkrétní obsah bloku se liší v konkrétních implementacích, kromě hashe předchůdce zde bývá časové razítko, množina transakcí a důkaz vynaložených prostředků na sestavení bloku. V případě Etherea[12] v hlavičce bloku nejsou uloženy transakce přímo, místo nich je zde položka *transactionRoot*, transakce jsou uloženy v modifikovaném Merklově stromu, *transactionRoot* je hodnotou kořenového uzlu tohoto stromu.




Obrázek 4.1: Zjednodušený pohled na blockchainovou strukuturu v Ethereu obsahující tři bloky

Součástí sítí postavených na blockchainu je algoritmus konsenzu, který je uplatněn při modifikaci blockchainu. Logicky není možné aby každý uzel přidával bloky tak jak uzná on sám za vhodné. Pro řešení tohoto problému existuje množství mechanismů, nejstarším a dosud hojně používaným je PoW

[1]Nastane, pokud není možné ověřit, zda kontrétní mince nebyla utracena vícenásobně. Důsledkem zneužití tohoto nedostatku by byla nekontrolovaná inflace měny.

(*proof-of-work*), využíván je jak v Bitcoinu, tak Ethereu. Principiálně PoW funguje tak, že na síti figurují těžaři (*miners*), kteří hledají magickou konstantu *nonce*, jejíž doplněním získá hash bloku specifický tvar, například posledních $n$ bitů je nulových – nalezená konstanta představuje důkaz odvedené práce. Toto „hledání konstanty“ je z očividných důvodů výpočetně velice náročná operace, proto jsou těžaři motivování odměnou v podobě mincí nativních danému blockchainu.

Dalším, v současné době na popularitě nabývajícím konsenzuálním algoritmem, je PoS (*proof-of-stake*), jehož varianta bude použita v Ethereu 2.0. V tomto algoritmu blok není těžen, ale vytvořen konkrétním uzlem, který ve vybrán na základě heuristik konkrétní implementace. Jako záruku, že se uzel bude chovat „slušně“, vsadí své kryptojmnění. Ostatní uzly vytvořený blok validují, pokud dojde ke shodě, autor bloku získá poplatek za transakci v opačném případě přichází penalizace. Zmíněné algoritmy zdaleka nevyčerpávají možnosti hledání konsenzu na blockchainu, zmíněny jsou proto, že jsou relevantní k obsahu této práce. Jako další mechanismy stojí za zmínku například *proof-of-authority*, nebo *proof-of-capacity*.

Ethereum je decentralizovaná síť na bázi blockchainu a funguje na ní kryptoměna Ether. Na Ethereum lze pohlížet jako na transakcemi řízený automat, jehož stav je definován posloupností provedených transakcí, tedy stavem blockchainu.

## 4.2 Stav

Stav (*state*) v Ethereu představuje mapování mezi adresami a stavy uživatelských účtů. Stav je modifikován transakcemi. Uživatelský účet sestává ze čtyř položek.

**`nonce`** – Počet transakcí které byly provedeny z tohoto účtu[2].

**`balance`** – Množství Etheru na účtu.

**`storageRoot`** – Kořenový hash Merklova komprimovaného stromu obsahujícího data účtu.

**`codeHash`** – Hash bajtkódu kontraktu nasazeného na této adrese, týká se pouze kontraktových adres a je neměnný.

[2]Nesouvisí s nonce bloku, který hledají těžaři.

## 4.3 Transakce

Transakce je operace modifikující stav blockchainu, hlavním účelem transakce je přenos Etheru mezi adresami a interakce s chytrými kontrakty. Adresa je dvaceti bytový identifikátor zapisovaný hexadecimálně s `0x` prefixem, celkem 42 znaků. Adresu může vlastnit uživatel, pak se jedná o externě vlastněnou adresu, která se vypočítá jako hash veřejného klíče s ořezem na daný počet bytů. Druhým případem jsou interní adresy, na kterých jsou uloženy kontrakty.

Modifikace blockchainu je operace vyžadující nenulové prostředky, proto se za transakci platí poplatek. Tento poplatek je vyčíslen tzv. palivem (gas) – odesílatel transakce sám urči cenu v Etheru, kterou je ochoten za gas zaplatit. Při volání kontraktů se cena navyšuje dle instrukcí které budou spuštěny na EVM. Každá instrukce EVM má pevně danou cenu za zpracování. Zde by ovšem chyba programátora kontraktu, řekněme konkrétně nekonečná smyčka, znamenala „nekonečnou“ cenu za transakci. To je samozřejmě nepřípustné a proto odesílatel deklaruje maximální množství gas, které lze na zpracování transakce použít a v případě překročení této hranice transakce selže. Struktura transakce v síti Ethereum je následující:

**`nonce`** – Sériové číslo transakce z adresy odesilatele počítáno od nuly

**`recipient`** – Adresátova adresa

**`value`** – Množství Etheru které se bude přenášet

**`data`** – Obecná datová nálož, může obsahovat argumenty funkcí kontraktu při jejich volání, případně samotný kontrakt při jeho nasazování na blockchain.

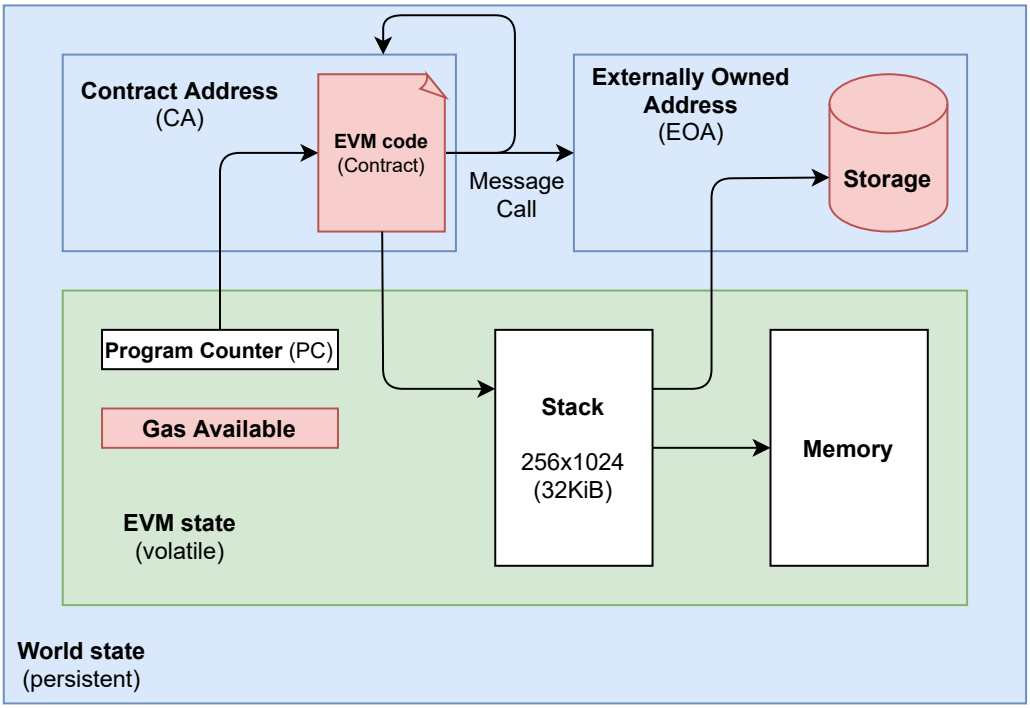
**`gasLimit`** – Maximální množství paliva, které se použije na zpracování transakce.

**`gasPrice`** – Cena za palivo v Etheru, čím je vyšší, tím ochotněji těžaři zahrnou transakci to bloku.

**`v,r,s`** – Složky ECDSA podpisu transakce

## 4.4 Ethereum Virtual Machine

Ethereum virtual machine (EVM) je turingovsky úplný virtuální stroj zásobníkové architektury, který je součástí každého uzlu sítě. Smyslem EVM je




Obrázek 4.2: Architektura Ethereum virtual machine.

zpracování bytekódu chytrých kontraktů (sekce 4.5). Stroj disponuje 32KiB zásobníkem a pamětí, *stack* a *memory* na obrázku 4.2. Tyto paměti jsou volatilní, jejich platnost končí se zpracováním každé transakce.

Instance EVM je limitována množstvím paliva, které je možné využít. Jako limit je použit nižší z uživatelského limitu (specifikován v transakci) a limitu EVM (konstantní). Palivo je spotřebováno většinou instrukcí EVM, a samotnout transakcí (poplatek za transakci). Vyjímkou jsou operace, které palivo naopak generují, sem patří například nulování *storage*. Jako *storage* je nazývána persistentní pamět blockchainu, každá adresa má přiřazenu dynamicky rostoucí *storage*. Se zápisy do *storage* je obecně snaha šetřit, protože cena za využití je relativně vysoká.

## 4.5 Specifika tvorby chytrých kontraktů

Chytré kontrakty (*smart contracts*) jsou programy určené ke spouštění na EVM. Dostupné jazyky pro vývoj kontraktů jsou Solidity, Vyper, nebo nízkoúrovňový Yul. Po sestavení se bytekód pomocí transakce nahraje na blockchain. Tato práce si v žádném případě neklade za cíl čtenáře naučit psát chytré kontrakty v jazyce Solidity, některá specifika tohoto jazyka je však žádoucí zmínit, protože pro nezasvěcené by text, zejména realizační

části, mohl být do jisté míry matoucí.

### 4.5.1 Viditelnost

Stejně jako mnoho jiných objektově orientovaných jazyků i v Solidity existuje koncept viditelnosti. Typické jsou modifikátory `private` a `public`, které jsou přítomné i v Solidity, navíc zde však máme `internal` a `external`. Význam těchto modifikátorů přístupu je:

**`external`**
: Metody deklarované jako externí lze volat pouze transakcí, na členské proměnné nelze použít.

**`public`**
: Metody deklarované jako veřejné lze volat jak transakcí, tak klasickým voláním z dalších kontraktů. Členská proměnná deklarovaná jako veřejná se chová trochu jinak, než by se dalo očekávat – při kompilaci je pro ní automaticky vygenerován veřejný getter se stejným jménem, pokud je proměnná pole, nebo mapování, pak getter příjmá index jako argument. S obsahem veřejné členské proměnné nelze manipulovat ani k němu přistupovat napřímo.

**`private`**
: Soukromá metoda nebo členská proměnná je viditelná pouze v kontraktu, kde byla deklarována.

**`internal`**
: Interní metoda nebo členská proměnná je viditelná uvnitř kontraktu, kde byla deklarována a navíc v jeho potomcích.

Termínem *externí rozhraní*, který se v textu několikrát objeví, je rozuměna množina metod kontraktu, která je volatelná transakcí (externě), což jsou metody deklarované jako `public` a `external`.

### 4.5.2 Stavové modifikace

Dalším specifikem jsou stavové modifikace, metoda může měnit stav sítě, číst z něj a nebo s ním vůbec nepracovat. První možnost je implicitní, pokud metoda ze stavu pouze čte používá se modifikátor `view`. Pokud metoda se stavem nepracuje použije se modifikátor `pure`. Význam tohoto rozdělení spočívá, kromě možných optimalizací využití paliva, zejména v možnosti práce s návratovou hodnotou. V případě modifikující metody musí dojít ke

konsenzu sítě nad změnou stavu a nutně se musí volat transakcí a nelze tak využít návratové hodnoty volané metody. Pokud je metoda označena jako `view` nebo `pure`, pak její volání může vyřídit konkrétní uzel a transakce se vůbec nevytváří. Takovéto volání tedy nestojí žádné palivo.

### 4.5.3 Funkční modifikátory

Solidity podporuje tzv. modifikátory na které lze nahlížet jako na prostředek ke znovupoužití kódu. Modifikátor je jednoduše řečeno obalovač funkce, ukázka definice modifikátoru a jeho aplikace je vidět níže.

```
modifier ownable() {
    require(msg.sender == owner, "Volající není vlastník.");
    _;
}

function setOwner(address _owner)
public ownable {
    owner = _owner
}
```

Modifikátor je uvozen klíčovým slovem `modifier`, následuje pojmenování modifikátoru s argumenty v závorce (v ukázce bez argumentů). V následném bloku je obecný zdrojový kód, speciální je zde příkaz „podtržítko“, na jehož místo se na úrovni kompileru doslova vloží modifikovaná funkce. Modifikátor je aplikován na funkci jeho uvedením v hlavičce obdobně jako modifikátor přístupu nebo stavu.

Uvedený příklad do kontraktu vkládá koncept vlastnictví – některé metody může volat pouze vlastník definovaný členskou proměnnou `owner`. Klíčovým slovem `require` je zde konkrétně řečeno, že pokud odesilatel transakce (`msg.sender`) není vlastníkem kontraktu, pak se má ukončit provádění kódu (*revert*) se zprávou „Volající není vlastník“. Voláním metody `setOwner` pak vlastník může předat vlastnictví jiné adrese, kterou předá jako argument.

### 4.5.4 Události

Nemožnost využití návratové hodnoty při volání metody transakcí lze považovat za problematické. Jako náhradu návratových hodnot se využívají tzv. události `events`. Událost si lze jednoduše představit jako strukturu, kterou lze indexovat podle až tří jejích členů. Při emisi je událost uložena do záznamů `log storage`, což je postranní datová struktura (nejsou přímou

součástí blockchainu). Klient pak může na události reagovat v reálném čase nebo je prohlížet a filtrovat[3] zpětně.

Využití události jako prostředku pro realizaci návratové hodnoty však plně nevyčerpává možnosti využití událostí. Velmi zajímavou vlastností událostí, je řádově nižší cena za uložení dat oproti *storage*. V případě, kdy je potřeba ukládat větší množství dat, se kterými kontrakt dále nepracuje (je pouze jejich producentem), můžeme využitím událostí značně ušetřit. Podmínka na kontrakt v roli producenta je nutná, protože kontrakt k emitované události ztrácí přístup (nemůže ji dále upravovat, ani číst).

Dalším využitím událostí je realizace asynchronního spouštěče – klient může reagovat na emise události, které sám nezpůsobil.

### 4.5.5 Knihovny

Knihovna je množina funkcí, kterou lze aplikovat na proměnnou. Nad touto proměnnou pak lze volat funkce knihovny, samotná proměnná se pak přenáší jako první argument. Například lze vytvořit knihovnu s funkcí `increment(uint storage a, uint b)`, která inkrementuje argument `a` o hodnotu argumentu `b`. Tuto knihovnu pak lze aplikovat na proměnnou `uint value = 1` a volat `value.increment(20)`, hodnota `value` bude `21`.

Knihovna se nasadí pouze jednou a ke kontraktům, které ji vyžadují se pouze linkuje. Toto je zjevná výhoda, čím méně kódu nasazujeme, tím méně platíme.

[3]Filtrovat lze podle idexovaných členů

# 5 InterPlanetary File System

Zásadním nedostatkem blockchainu, který bude nutné při návrhu protokolu adresovat, je praktická nemožnost ukládání „velkých“ dat. V případě sítě Ethereum za velká data můžeme označit již jednotky kilobytů. Tuto limitaci lze obejít využitím meziplanetárního souborového systému IPFS[1]. Jednoduše lze IPFS popsat jako distribuovaný systém pro čtení a zápis dat. V následujících řádcích bude tento systém představen.

Vlastností, která dělá IPFS dobrým doplňkem k blockchainu, je adresování na základě obsahu souboru – v kontrastu s adresováním na bázi lokality jako například v unixových systémech nebo na webu. Adresa, zde označena jako content id (CID), je hashem daného souboru. Žádoucím důsledkem této vlastnosti je vynucená neměnnost souborů, pokud by byl soubor modifikován mění se jeho CID a tedy se vlastně jedná o jiný soubor dostupný pod jinou adresou. Implementačně je IPFS založeno na Merkelově orientovaném acyklickém grafu (*Merkle DAG*)[1].

Z pohledu vývoje na blockchainu stačí uložit pouze CID, které pak bude vždy odkazovat na tentýž soubor, pakliže nebude smazán. Tím je docíleno uložení dat na „drahý“ blockchain s minimálními náklady.

[1]Orientovaný acyklický graf, jehož hrany jsou realizovány hashem potomka – obdobně jako spoje v blockchainu. Vlastně by se dalo o Merklově DAG uvažovat jako o zobecnění blockchainu (každý blockchain je zároveň Merklův DAG).

# 6 Analýza protokolů

Cílem této kapitoly je diskuse možností jak navrhnout komunikační protokol. Tato diskuse sestává ze dvou částí, v první je řešeno jak ověřit totožnost uživatele a vlastnictví adresy na komunikačním kanálu. V části druhé budou diskutovány mechanismy pro bezpečné předávání zpráv mezi autentikovanými uživateli.

Blockchain přináší možnost transparentní práce s daty, této výhody lze využít při implementaci datových struktur potřebných pro autorizaci účastníků komunikace. Už jen distribuce veřejného klíče lze pomocí blockchainu implementovat bezpečněji, než klasické vystavení na webu, kdy si člověk často nemůže být jist, zdali klíč skutečně patří požadovanémů adresátovi. Ve chvíli, kdy se na blockchain podaří nahrát veřejný klíč spárovaný s identitou jeho vlastníka, je za předpokladu korekní implementace chytrého kontraktu nemožné ho změnit. Pokud je zdrojový kód kontraktu veřejný, lze ověřit že na adrese kontraktu skutečně běží. V případě, kdy zdrojový kód dostupný není, lze na nahraný binární program použít disassembler a funkčnost auditovat tímto způsobem. Toto je velká výhoda oproti klient-server architektuře, kdy i přes to, že softwarová výbava serveru může být open source, vlastně nevíme co na serveru běží, ani to vědět nemůžeme. Ethereum pracuje s myšlenkou „kód je zákon“, pravidla hry jsou dohodnuty při nasazení kontraktu a dále se dají těžko měnit. Tento fakt zásadně zesložiťuje implementaci náhlých požadavků na cenzuru uživatelů, které by mohly přicházet ať už od provozovatele služby nebo státu.

Je zřejmé, že blockchain přináší do oblasti zabezpečení komunikace vlastnosti, které by byly v konvenční klient-server architektuře těžko dosažitelné. Navrhovaný protokol se bude snažit tyto vlastnosti využít v maximální možné míře.

## 6.1 Ověřovaný kanál a adresy

Pro účel návrhu a analýzy komunikačního protokolu je nutné zadefinovat zde několik pojmů:

**Ověřovaný kanál**
: Jedná se o kanál, skrze který má uživatel úmysl komunikovat, ale zatím není známa jeho adresa na tomto kanálu, resp. není prokázáno její vlastnictví. Ověřovaný kanál je uvažován co nejobecnější – asynchronní

a nezabezpečený. Jako příklad bude zmíněna elektronická pošta nebo telefonní služby, principielně se může jednat i o přepravní službu, jako například Česká pošta.

**Adresa na ověřovaném kanálu (ověřovaná adresa)**
Adresou je myšlen unikátní identifikátor na daném ověřovaném kanálu s jehož znalostí je možné příjemci doručit zprávu. U zmiňované elektronické pošty by to byla „mailová“ adresa, u telefonních služeb telefonní číslo a pro tradiční poštu adresa nemovitosti. Zde si lze všimnout, že adresa nemovitosti figuruje jako adresa na vícero nezávislých kanálech – dopis může do schránky hodit zaměstnanec ČP, nebo i jiné přepravní společnosti, takže lze přepravní služby obecně označit jako jeden kanál.

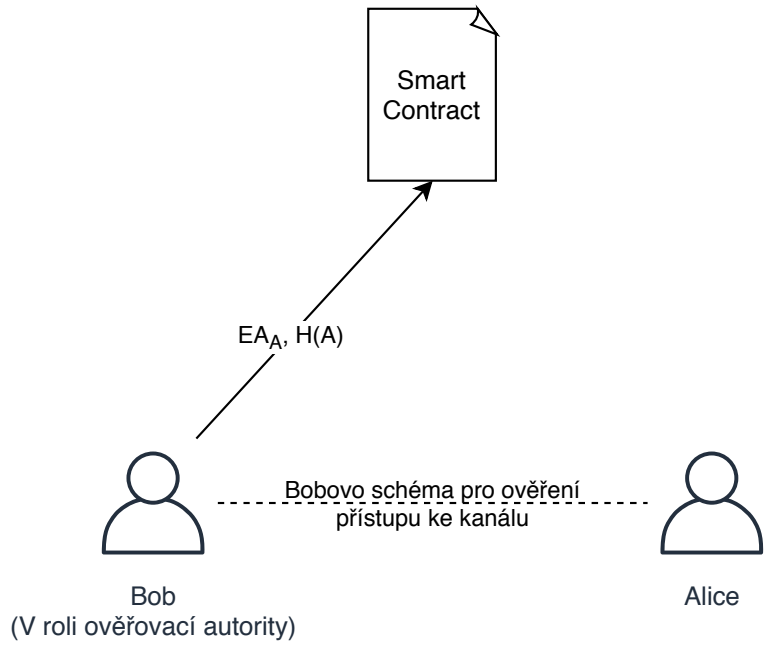
**Identita**
Identita vzniká ověřením adresy na ověřovném kanálu a patří konkrétnímu uživateli. Uživatel může mít více identit a s každou se pojí množina časových razítek, které reprezentují provedená ověření.

## 6.2 Autentikační protokoly

Vytvoření nové identity je podmíněno souhlasem verifikační autority (VA). Tu si lze představit podobně jako certifikační autoritu v TLS, rozdíl je však v tom, že zde není vystavován certifikát. Funguje také analogie se sítí důvěry, VA se zde chová jako uživatel podepisující veřejný klíč jinému uživateli, rozdíl je v tom, že žádný klíč podepisován není. VA pouze říká „Ručím za pravdivost této identity“, neodmítnutelnost tohoto výroku zajišťuje blockchain principem své funkce. VA samozřejmě musí vlastnictví nějak ověřit, způsob jakým tak provede je zcela v její kompetenci, zde budou však uvažovány protokoly fungující za základě posílání autorizačního tokenu. Tento způsob je zvolen protože funguje zcela obecně, ať už se ověřuje emailová adresa, telefonní číslo, nebo adresa vlastněné nemovitosti. Není však proti ničemu, aby konkrétní VA před vydáním tokenu využila jiné, nadstandardní, prostředky a tím dosáhla vyšší důvěryhodnosti.

### 6.2.1 Síť důvěry

Implementace sítě důvěry v rámci chytrého kontraktu je jednou z možností jak řešit problém ověření přístupu na kanál. Síť důvěry pak může být uložena jako grafová datová struktu přímo na blockchainu. Uživatel je identifikován svojí adresou v Ethereu a seznamem adres na ověřovaných kanálech.




Obrázek 6.1: Ověření přístupu ke kanálu v blockchainové síti důvěry.

Každá z techto adres je svázána s množinou potvrzení od ověřovacích autorit – v tomto případě je každý uživatel ověřovací autorita. Algoritmy pro vyhodnocení platnosti klíčů budou implementovány lokálně.

Validační protokol je z implementačního pohledu velmi jednoduchý, vizte obrázek 6.1, vlastně jde o jedinou transakci obsahující Alicinu Ethereovou adresu $EA_A$ a hash adresy na ověřovaném kanálu $H(A)$, kterou zde Bob potvrzuje, že Alice vlastní adresu $A$. Tuto skutečnost však nejprve Bob ručně ověří, jak toto ověření provede je zcela v jeho režii – těžko lze algoritmicky vynucovat, aby se uživatelé mimo blockchain chovali specifickým způsobem.

Oproti síti důvěry v PGP má blockchainové řešení výhodu pomyslné centralizace identit na blockchain namísto použití distribučních serverů. Z toho plyne možnost garance dostupnosti, ať už ve smyslu konektivity, nebo absence cenzury identit. Obecné nevýhody klasické sítě důvěry však zůstávají (vizte sekce 3.1.3) a proto se práce tímto směrem ubírat nebude.

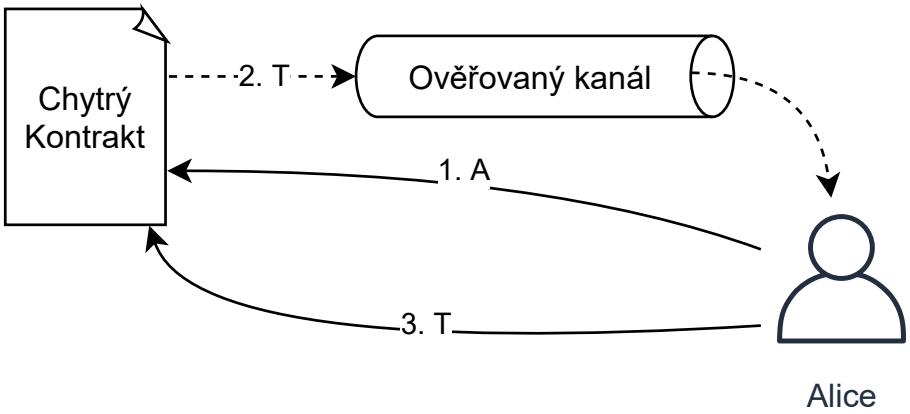
### 6.2.2 Naivní přístup

Z implementačního hlediska by se nabízelo použít „ověřovací zprávu“, jak jí známe z různých registračních formulářů na webu. Zpráva je rozesílána přímo z chytrého kontraktu na adresu $A$ a obsahuje autorizační token $T$, kterým uživatel následně prokáže vlastnictví adresy na ověřovaném kanálu. Tento naivní protokol je vyobrazen na diagramu 6.2, má však dva zásadní

nedostatky.

- Autorizační token $T$ je generován „on-chain“. EVM funguje deterministicky a neposkytuje žádný bezpečný zdroj entropie, proto není možné bezpečně generovat $T$[11]. V případě, že by se $T$ dalo predikovat však přenos přes ověřovaný kanál pozbývá významu, protože znalost $T$ nelze brát jako důkaz vlastnictví adresy.
- Komunikace okolního světa s EVM může probíhat pouze transakcemi, tato vlastnost je opět dána požadavkem na deterministočnost. Tato skutečnost samozřejmě úplně vylučuje odeslání $T$ přes ověřovaný kanál.

Uvedené nedostatky činí naivní protokol v síti Ethereum nerealizovatelný, lze ho však použít jako odrazový můstek k následujícím návrhům.




Obrázek 6.2: Naivní protokol na způsob ověřovací zprávy, chytrý kontrakt zde plní roli ověřovací autority. Plná čára představuje transakci.

### 6.2.3 3t protokol

Naivní přístup selhal na nemožnosti integrovat VA do chytrého kontraktu, při separaci VA lze dojít k tří-transakčnímu protokolu ($\mathcal{3}t$), který je naznačen na diagramu 6.3. Tento protokol vede k důkazu přístupu na kanál pěti zprávami, z nichž tři jsou transakce.

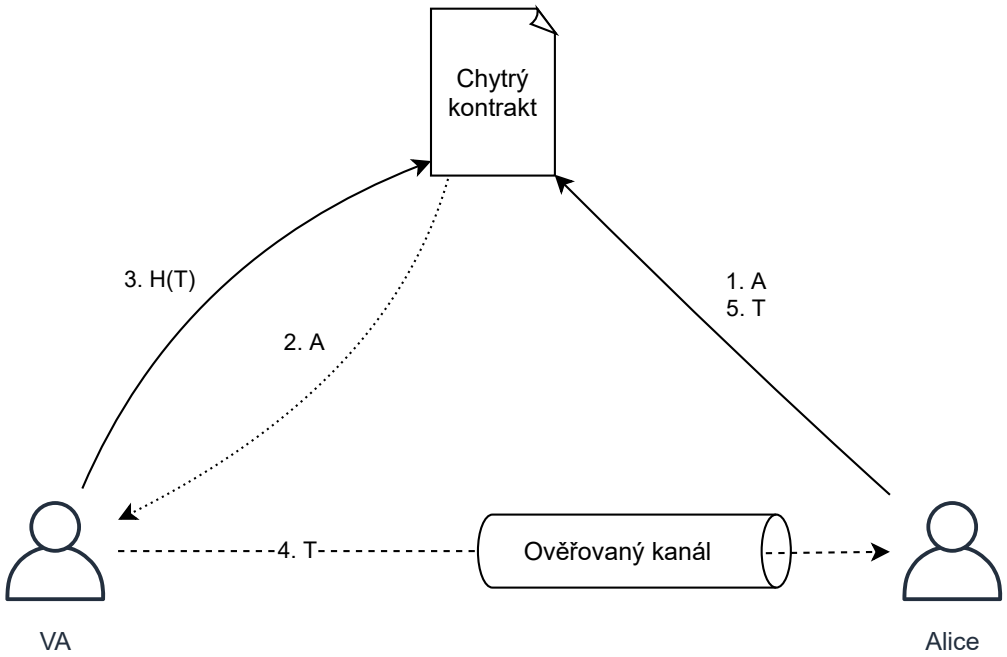
1. Alice posílá transakci na adresu kontraktu obsahující adresu na ověřovaném kanálu $A$, jejíž vlastnictví chce prokázat.
2. Kontrakt vyvolá událost obsahující ověřovanou adresu $A$, VA ji odchytí.
3. VA vygeneruje náhodný token $T$ a jeho hash $H(T)$ pošle transakcí na adresu kontraktu.
4. VA odešle Alici token $T$ skrze ověřovaný kanál.
5. Alice transakcí předá kontraktu obdržený token $T$.
6. Kontrakt provede vyhodnocení – pokud Alicí odeslaný token $T$ odpovídá hashi $H(T)$ vlastnictví je prokázáno.

Mohlo by se na první pohled zdát, že je zbytečné aby autorita posílala kontraktu hash tokenu $H(T)$, stačilo by odeslat $T$. Zde je nutné si uvědomit, že data transakcí jsou veřejná a posíláním tokenu v otevřeném textu by Alici umožnilo tento token z transakce vyčíst a v důsledku předložit důkaz vlastnictví adresy, kterou ve skutečnosti nevlastní.

O tomto protokolu již lze říci, že je realizovatelný, má však jednu potenciálně nežádoucí vlastnost. Zde je třeba zvážit v jakém prostředí bude protokol implementován, v případě privátního blockchainu, například v rámci firmy, nemusí být na závadu poslat adresu v rámci transakce. Obecně však není příliš vhodné vystavovat adresy veřejně a čelit tak potenciálnímu spamu, to platí zvlášť v případě blockchainu, kde není možné jednou publikovanou informaci odstranit. Navíc jsou zde prováděny tři synchronní transakce, to znamená, že běh protokolu zabere v nejlepším případě minimálně 3 bloky. I vzhledem k ceně transakcí by mělo být snahou jich provádět, pokud možno, co nejméně.

Jako zajímavou vlastnost tohoto protokolu lze uvést teoretickou možnost randomizace výběru VA, a tímto způsobem omezit vliv záškodných VA[1], efektivita tohoto přístupu bude diskutována v sekci 7.3.1. Při náhodném výběru autorit je třeba čelit již zmíněnému problému s omezenými

[1]VA zneužívající svého statutu k rozesílání verifikačních tokenů postranním kanálem




Obrázek 6.3: Protokol pro autoritativní ověření s použitím chytrého kontraktu. Transakce jsou značeny plnou a události tečkovanou čarou.

zdroji entropie, problematika bezpečnosti generátorů náhodných čísel bude nastíněna v sekci 7.1

### 6.2.4 2t protokol

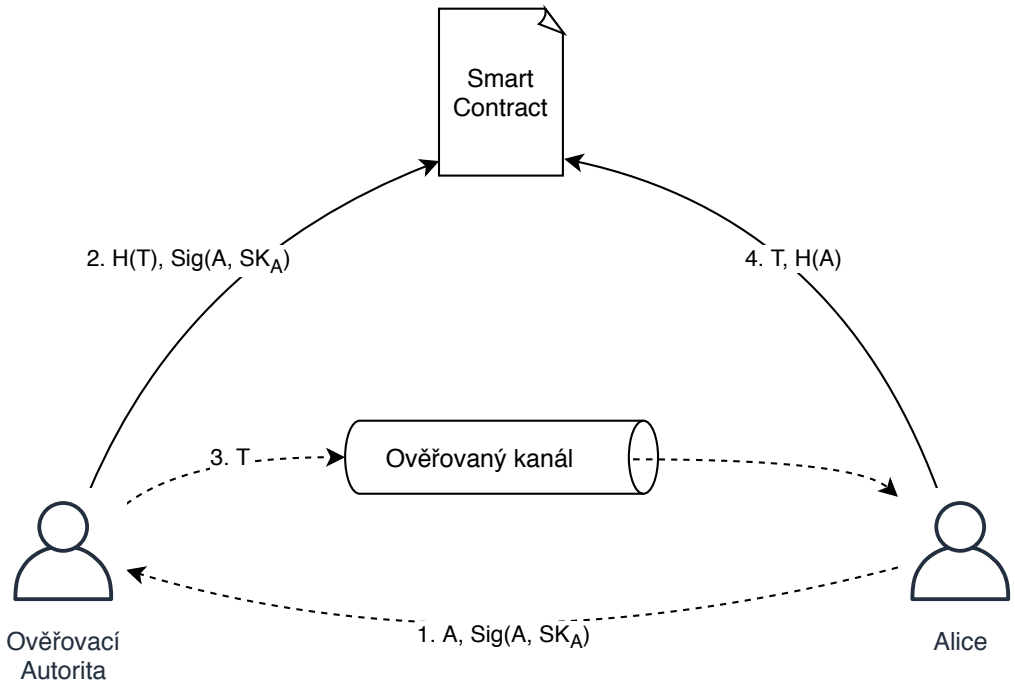
Problémy předchozího protokolu lze řešit jeho modifikací, která je vyobrazena na obrázku 6.4. Chytrý kontrakt již neplní roli zprostředkovatele při ověřování. Uživatel kontaktuje přímo ověřovací autoritu, tímto se ušetří jedna transakce a událost. Protokol pracuje v následujících krocích.

1. Alice pošle zabezpečeným kanálem ověřovanou adresu $A$ a její podpis $Sig(A, SK_A)$.

2. VA vygeneruje náhodný token $T$ a transakcí předá kontraktu $H(T)$ a $Sig(A, SK_A)$, kontrakt si je uloží do fronty.

3. VA zašle ověřovaným kanálem Alici $T$.

4. Alice transakcí předá kontraktu $T$ spolu s hashem ověřované adresy $H(A)$. Kontrakt napočte $H(T)$ a z fronty vyřadí odpovídající podpis. Podpis je ověřen vůči $H(A)$, pokud souhlasí, vlastnictví je prokázáno.




Obrázek 6.4: Protokol pro autoritativní ověření s použitím chytrého kontraktu.

V protokolu figuruje hash adresy $H(A)$, ukládání pouze této hodnoty namísto adresy jako otevřeného řetězce má své výhody. Jak již bylo zmíněno čitelnost adres by mohla vystavit jejich vlastníka spamu, hashováním adresy je tento problém vyloučen. Další výhodou může být nezávislost ceny transakce na délce adresy – hash má pevně danou velikost, adresa tedy může být prakticky neomezeně dlouhá.

Protokol by bylo navíc vhodné doplnit o synchronizační událost, která uživateli oznámí, že transakce od autority byla zpracována. Protokol bude samozřejmě fungovat i bez této události, zabrání se jí však teoretické situaci, kdy uživatel získá token dříve, než je veden na blockchainu. V takové chvíli nemůže transakce skončit jinak, než jejím selháním a uživatel tudíž zbytečně platí transakční poplatek. Stejně tak lze využít událost pro notifikaci uživatele o výsledku validace.

## 6.3 Revokace ověření

Ve světě TLS a certifikačních autorit občas dochází k situaci, kdy držitel certifikátu ztratí důvěru a je nutné jeho certifikát revokovat nadřazenou autoritou. Tato situace samozřejmě může nastat i u navrhované ověřovací

autority a samotných uživatelů. Řešení této situace je nasnadě.

Chytrý kontrakt si musí pro korektní funkci držet množinu ethereových adres na kterých sídlí VA, v opačném případě by se mohl za VA vydávat každý. Tato množina zároveň umožňuje triviální implementaci revokace všech vydaných ověření danou VA, stačí jí z množiny odstranit. Klienti pak při usuzování jak velkou důvěru v identitu jiného uživatele mají mohou vzít v potaz fakt, že VA, která mu vydala ověření, byla odstraněna z množiny verifikačních autorit.

Správce verifikačních autorit může být v nejjednodušší variantě majitel kontraktu, případně jich může být více a o manipulacích s množinami verifikačních autorit a jejich správců hlasovat. Zde je opět volnost ve volbě typu nadpoloviční většiny – prostá, nebo absolutní, jiné poměry zde nedávají příliš smysl. Na odevzdání hlasu mají správci omezený čas. Časové okno by mělo být dostatečně dlouhé na to, aby se správci stihli vyjádřit, zároveň však musí být v případě volby prostou většinou hlasování odbaveno rychle vzhledem k možné urgenci hlasované operace.

V případě odebrání verifikace koncovému uživateli není problém pro verifikační autoritu vydané ověření stáhnout s okamžitou platností.

## 6.4 Zahájení relace a úložiště klíčů

Předtím než Alice kontaktuje Boba, je nutné vyjednat společné tajemství, k tomuto účelu bude uvažována adaptace protokolu XD3H. Při implementaci X3DH serveru v rámci chytrého kontraktu je opět potřeba vzít v potaz, že data jednou uložená na blockchain jsou po dobu jeho existence veřejná. I v případě, že se jedná o proměnnou chytrého kontraktu, jejíž hodnota je později změněna, se stále lze dobrat původní hodnoty – stačí přehrát historii transakcí a sledovat, jak se proměnná mění. To je však zásadní rozdíl oproti typickému serveru, kde se dá předpokládat, že data na něm uložená lze držet v tajnosti a i v případě bezpečnostního incidentu lze s jistotou říci, že data, která již na disku fyzicky neexistují, nemohou být vyzrazena.

Tato vlastnost má praktický následek, že pokud Eve odposlechne zprávu mezi Alicí a Bobem, dostat se k Bobovým veřejným klíčům, které sloužily v derivaci společného tajemství nepředstavuje značný problém. Eve pak musí od Alice získat soukromou část identitního $IK_A$ a dočasného $EK_A$ klíče, který je nutné prolomit. Za předpokladu silného kryptosystému je lámání klíčů těžko realizovatelná operace, proto vystavení veřejných klíčů na blockchain nelze považovat za bezpečností riziko pokud Alice bezpečně pracuje se svými klíči. Kritické je zejména důsledně mazat $EK_A$ ihned po

použití, aby se zamezilo případnému úniku.

V typické klient-server architektuře není problém velmi často nahrazovat střednědobý klíč $SPK$, v Ethereu tato operace vyžaduje transakci. Transakce stojí gas, a není tedy možné vynutit na uživateli aby často měnil $SPK$, proto je nutné počítat i se scénářem, kdy se $SPK$ prakticky nemění. Také je nutné zvážit latenci, která je u serveru typicky v desítkách milisekund, u kontraktu se pohybujeme v řádu desítek až stovek vteřin – je nutné vytěžit blok, aby se $SPK$ propsal na blockchain. V důsledku této zvýšené latence mohou nastávat situace kdy Bob změní svůj $SPK$, avšak Alice použije neaktuální verzi. Tato situace lze řešit lokálním smazáním $SPK$ až ve chvíli, kdy je nahrazen podruhé – tj. Bob drží aktuální a předchozí $SPK$. Případné lepší řešení by bylo smazat starý $SPK$ po vytěžení bloku, který vypropaguje nový klíč na blockchain. Zmíněné skutečnosti prodlužují časový slot ve kterém se Eve může zmocnit Bobova střednědobého privátního klíče. Pokud by se Eve podařilo získat také identitní klíč $IK_B$, pak ji již nic nedělí od dopočtení sdíleného klíče v případě, že jednorázový klíč $OPK$ nebyl použit. Z uvedených důvodů bude z protokolu vyjmuta možnost nevyužívat jednorázové klíče.

### 6.4.1 Přímé ukládání do storage

Odejmutí možnosti nevyužít $OPK$ ovšem zvýrazní problém vyčerpání klíčů, který povede k nemožnosti kontaktovat protějšek. Tato skutečnost lze zneužít pro útok na konkrétního uživatele. Pokud Eve chce zamezit Alici kontaktovat Boba, může Eve opakovaně žádat server o klíče, dokud je nevyčerpá, až bude chtít komunikovat Alice, tak se na ní klíč nedostane.

V klasické klient-server architektuře se nabízí tři triviální opatření, jak tento vektor útoku minimalizovat. Těmi jsou časová omezení na rezervace klíčů, dostatečná zásoba klíčů a včasné občerstvení spotřebovaných klíčů. Problém těchto omezení je však jejich paměťová náročnost, při skladování sta veřejných klíčů o velikosti 32B se jedná o 3,2KB paměti. Při ceně storage 625 gas za byte se zde pohybujeme v řádu desetin Etheru, takto vysoká částka za skladování klíčů samozřejmě není přijatelná. Nehledě na to, že přepis paměti je stále zpoplatněn čtvrtinou alokačního poplatku, takže ani časté občerstvování menšího počtu klíčů není cestou.

Časová omezení na výběr klíčů jsou do jisté míry realizovatelná, avšak vzhledem k nízkému množství klíčů, které lze prakticky na blockchain uložit, by musela být až příliš restriktivní. Z uvedených důvodů se skladování klíčů ve storage nejeví jako šťastné řešení.

### 6.4.2 Zpoplatněné klíče

Alternativním přístupem je nesnažit se snížit cenu za uložení klíčů, nýbrž počet klíčů které je nutné na blockchain nasadit. Pokud by byl počet klíčů omezen příliš, reálně hrozí problém jejich vyčerpání. To lze do jisté míry řešit zpoplatněním rezervací klíčů, tyto poplatky by zároveň sloužily jako kompenzace pro vlastníka klíčů. Ve výsledku tedy cenu za paměť platí odesílající strana, což je z hlediska férovosti lepší variantou, než nahrátí desítek klíčů na své náklady jen proto, aby je zlomyslný uživatel vyčerpal. I přes férovost tohoto přístupu však setrvává problém ceny za úložiště, kterou tento postup vlastně příliš neřeší, pouze jí částečně upozadí.

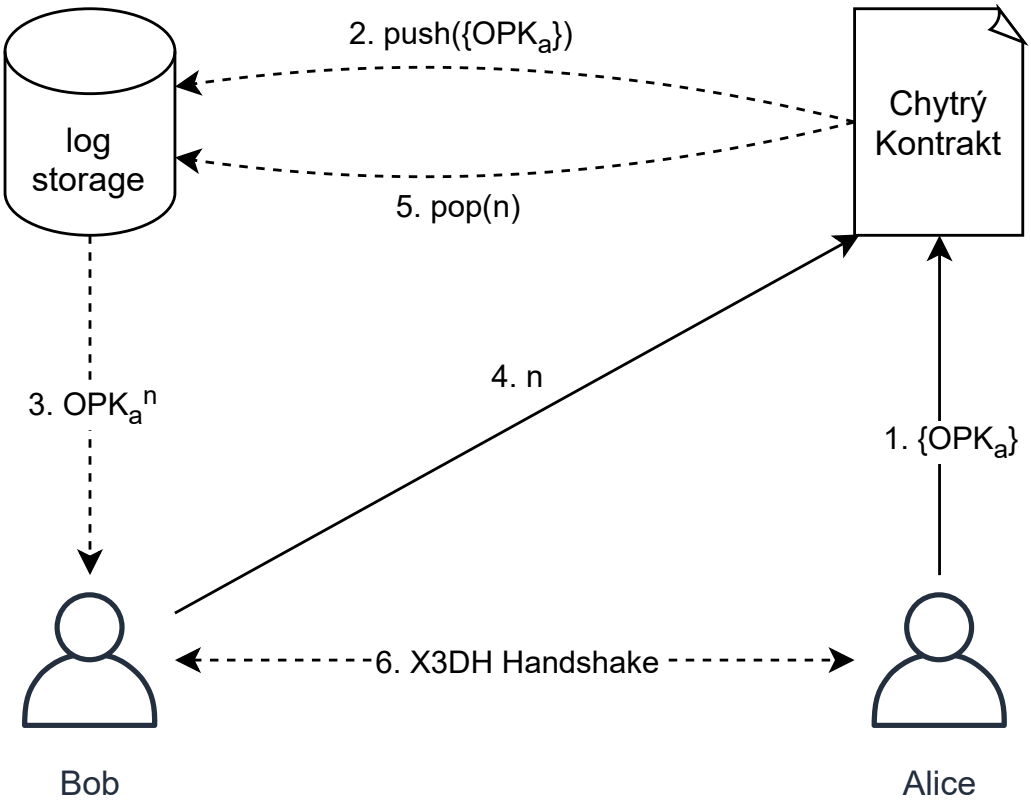
### 6.4.3 Logovaný zásobník

Možným řešením problému je nevyužívat pro skladování klíčů *storage*, ale *logy* (sekce 4.5.4), které jsou na ukládání větších datových objemů výrazně levnější. Implementace takového řešení by tedy spočívala ve vytvoření pomyslného zásobníku a kontrakt by pomocí událostí ohlašoval operace *push* a *pop*. Klientský software by pak lokálně zrekonstruoval zásobník do aktuálního stavu a oznámil kontraktu, který klíč si rezervuje. Zda-li tento požadavek dává smysl, tedy klíč existuje a je volný, zjistí až příjemce iniciální zprávy (vlastník klíče). Takovýto protokol by mohl fungovat následovně:

1. Alice předá kontraktu množinu svých jednorázových klíčů $\{OPK_A\}$.
2. Kontrakt emituje událost (*push*) obsahující množinu klíčů $\{OPK_A\}$.
3. Bob lokálně přehraje historii událostí a vybere si dostupný klíč $OPK_A$ na $n$-tém indexu.
4. Bob kontraktu oznámí index klíče, který si vybral.
5. Kontrakt emituje událost (*pop*) invalidující zvolený klíč.
6. Bob použije $OPK_A$ k navázání komunikace s Alicí.

Již při prvním pohledu se však naskytne otázka, co se stane v případě souběhu nad čtvrtým krokem, tedy výběrem klíče. Nutno podotknout, že souběh je zde velmi reálný, časové kvantum je zde celý blok, tedy až vyšší desítky vteřin. V takovém případě bude oběma žadatelům o tentýž klíč vyhověno, kontrakt klíč invaliduje dvojnásobně a komunikaci s Alicí naváže ten „rychlejší“. Opozdilec Alici kontaktuje s již použitým klíčem a tudíž mu

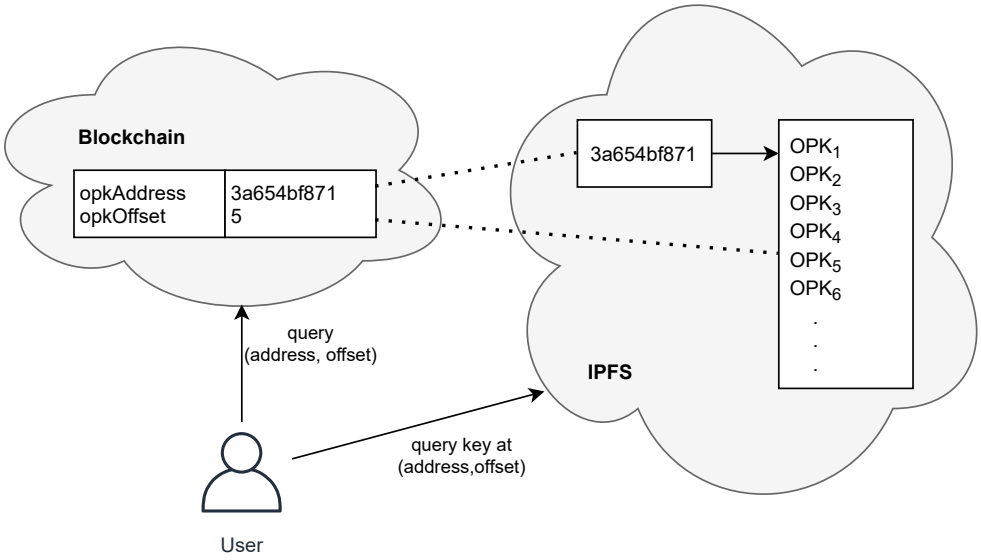
nemůže být vyhověno. K řešení této situace by ovšem byl nutný monitoring zásobníku chytrým kontraktem a ten nelze zajistit, protože logy jsou pro kontrakt *write-only* paměť. Grafické znázornění algoritmu je vidět na obrázku 6.5.




Obrázek 6.5: Posloupnost volání při *logování* klíčů.

### 6.4.4 IPFS a relativní adresování

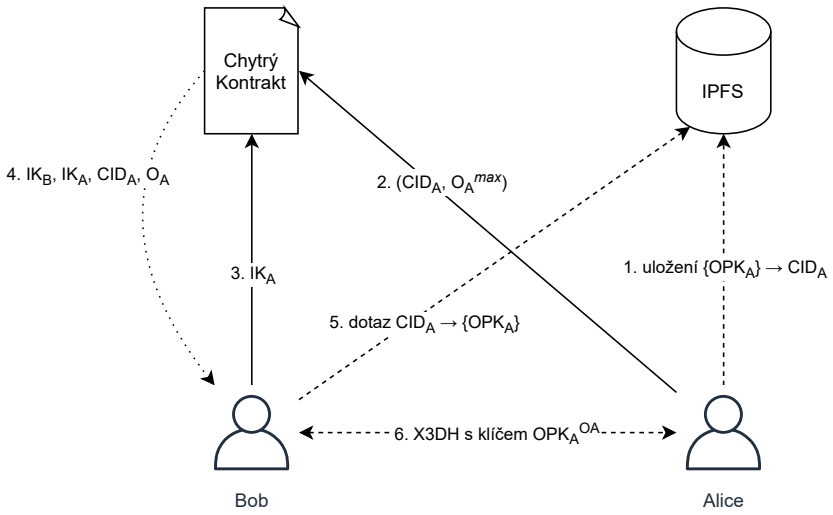
Elegantním řešením je popužití IPFS pro skladování velkého množství klíčů, na blockchain bude uložena pouze adresa souboru s klíči a offset (obrázek 6.6). Vzhledem k charakteristikám IPFS se touto strategií lze triviálně vyhnout oběma problémům, tj. vysoké ceně za úložiště i vyčerpání klíčů.




Obrázek 6.6: Adresování jednorázových klíčů s offsetem na IPFS.

V případě, že Bob chce kontaktovat Alici lze postupovat následujícím způsobem (diagram na Obrázku 6.7):

1. Alice na IPFS uloží soubor s množinou klíčů $\{OPK_A\}$, ta bude dostupná pod *contentId* $CID_A$.
2. Alice kontraktu předá $CID_A$ a maximální offset $O_A^{max}$. Kontrakt tyto hodnoty uloží, vynuluje offset $O_A$.
3. Bob zarezervuje jednorázový klíč předáním identitního klíče $IK_A$.
4. Kontrakt vyvolá událost obsahující čtveřici ($IK_B$, $IK_A$, $CID_A$, $O_A$), na kterou Bob naslouchá. Po této operaci je $O_A$ interně inkrementován. Tato událost bude později zároveň sloužit Alici jako důkaz, že Bob má daný klíč zamluven.
5. Bob odchytil událost a z IPFS si stáhne soubor s klíči *contentId* $CID_A$.
6. Bob použije klíč na offsetu $O_A$ ke kontaktování Alice.




Obrázek 6.7: Protokol pro ukládání a rezervaci jednorázových klíčů.

Tento protokol je imunní vůči souběhům. Ve chvíli, kdy jsou emitovány události už je patřičný blok finalizován a každému žadateli je tudíž přidělen unikátní offset. Nemůže se tedy stát, že by dva a více uživatelů zarezervovalo tentýž klíč.

Co se týče útoku vyčerpáním klíčů, zde lze nasadit již zmíněné časové restrikce. Každý uživatel má uloženo časové razítko bloku, ve kterém byla vyřízena jeho poslední rezervace. Kontrakt pak bude validovat, zdali uběhl dostatečný čas. Tato restrikce bude globální, při ukládání párů razítka a adresáta by paměťová náročnost mohla být příliš vysoká. Časový limit zde nemusí být ani příliš vysoký, klíčů je možno na IPFS uložit libovolné množství. Prakticky se tento limit zaokrouhlí nahoru na čas do vytěžení dalšího bloku, proto deset vteřin je více než dostačující. Případně je možné nechat uživatele specifikovat tuto hodnotu pro jeho vlastní klíče.

## 6.5 Key-management (Double Ratchet)

Po dohodě společného tajemství pomocí upraveného X3DH protokolu je nutné nasadit protokol využívající společné tajemství k šifrování uživatelských konverzací. Tomuto účelu vyhovuje již zmíněný Double Ratchet algoritmus, který má velmi dobré kryptografické vlastnosti jak z pohledu bezpečnosti, tak z pohledu integrace do uvažovaného systému. Ve skutečnosti lze tento algoritmus do systému integrovat tak jak je popsán ve své specifikaci.

# 7 Úvahy o bezpečnosti

## 7.1 Problematika náhodných čísel

V analýze protokolů bylo konstatováno, že náhodná čísla nelze generovat. V této části budou osvětleny důvody pro toto tvrzení. Pomyslný seznam způsobů jak si na blockchainu opatřit entropii, který tu bude představen, nelze však brát jako vyčerpávající.

### 7.1.1 Čistě blockchainový generátor

Generování náhodných čísel je v Ethereu reálný problém, který je dán už samotným principem funkce této sítě. Pokud by bylo možné generovat skutečně náhodná čísla, pak by se těžko hledal konsenzus nad stavem blockchainu mezi jednotlivými těžaři – každý by kalkuloval s jinými daty. Tento fakt ovšem nemění nic na tom, že náhodná čísla je potřeba na blockchainu čas od času generovat.

Pro vytvoření generátoru náhodných čísel je potřeba získat zdroj entropie. Nejhorší možnou volbou jsou vlastnosti bloku jako adresa těžare, složitost, limit paliva, výška a časové razítko. Problém s těmito hodnotami je, že je nastavuje těžař, ten si tedy při znalosti algoritmu může nastavit tyto hodnoty tak, aby mu padlo číslo, které si přeje.

Jako další možnost by se nabízel hash předchozího bloku, ani to však není zcela bezpečné. Pokud vytvořím kontrakt implementující identický generátor, tak budu dostávat identická náhodná čísla. To lze zneužít například k obalení kontraktu na který chci zaútočit a následně budu v každém bloku volat funkci obalovacího kontraktu. Tato funkce revertuje transakci, pokud se vygeneruje nezajímavé číslo a přepošle ji dál v opačném případě. Jako dobrou analogii tohoto útoku je hraní Člověče nezlob se, ale kostkou házím „nanečisto“, pouze ve chvíli, kdy hodím šestku prohlásím, že tento pokus byl ve skutečnosti platný.

Alternativně je možné využít hash budoucího bloku, i to má své problémy – prvně je nutné proces rozdělit do dvou fází, musí se čekat, než se budoucí blok vytěží, zároveň to také znamená dvě transakce místo jedné. Pak je třeba ošetřit, aby ve chvíli, kdy je náhodné číslo vygenerováno, nebyl budoucí blok již více, než 256 bloků starý. V tom případě EVM vrací nulový hash a náhodné číslo nebude náhodné. Obecně však platí, že hash bloku je také ovlivnitelný těžaři – pokud vytěžený blok negeneruje „výherní“ číslo, tak

ho zahodí a těží dál. Tento útok je na prostředky už poněkud náročný, ale pokud by útočník chtěl podvést blockchainové kasíno, kde je možné vyhrát 1000 Etheru, tak necelou setinku Etheru, kterou by dostal za vytěžený blok, jistě rád oželí.

Na závěr je nutno konstatovat, že generování náhodných čísel přímo na blockchainu nelze považovat za bezpečné.

### 7.1.2 Orákulum

Pokud nelze generovat náhodná čísla na blockchainu, pak musí přijít z vnějšku. K tomu slouží takzvané orákulum, což je ve své podstatě chytrý kontrakt, který dokáže zodpovědět dotazy na data, která se nachází mimo blockchain. Toho je dosaženo skrze události a naslouchajícího klienta, který transakcí posílá kontraktu zpět data, která si vyžádal vyvolanou událostí. Problém je, že orákulum se bude z principu chovat centralizovaně, tzn. je nutné věřit, že náhodná čísla, která generuje jsou opravdu náhodná, což se dá těžko dokázat.

Problém s centralizací Orákula řeší projekt Chainlink, který generování náhodných čísel podporuje a je v tuto chvíli celkem populární[1]. Je to cesta jak bezpečně implementovat *3t* protokol s náhodným přiřazováním autorit. Chainlink je však mimo rozsah této práce, proto se jím zde dále zabývat nebudu.

## 7.2 Odposlech verifikačního tokenu

Samotné odposlechnutí verifikačního tokenu v *2t* protokolu neotevírá útočníkovi dveře ke krádeži identity – pokud by se jím chtěl prokázat, tak nebude souhlasit podpis $Sig(A, SK_A)$ předaný kontraktu žadatelem skrze ověřovací autoritu. Pokud by se *3t* protokol upravil tak, že s adresou je předána i její signatura, tak by ani zde útočník nebenefitoval ze samotné znalosti tokenu.

Problém by nastal ve chvíli, kdy by sám útočník inicioval požadavek na ověření a následně odposlechl verifikační token. Dalo by se argumentovat, že pokud má útočník možnost odposlechnout verifikační token, tak je férové ho považovat za vlastníka proklamované adresy. U adres na alespoň nějakým způsobem zabezpečených kanálech by tento argument dával smysl (majiteli uniklo heslo nebo klíče a nyní má adresa „majitelů“ více), problematické by však mohlo být například u ověřování vlastnictví nemovitosti podle adresy,

[1]Ke konci dubna 2021 je Chainlink se svým nativním tokenem LINK dvanáctým největším krypto-projektem dle tržní kapitalizace[4].

kde jako komunikační kanál by figurovala Česká pošta. Pokud záškodník nenápadně vybere poštovní schránku před domem, nebo má své lidi uvnitř pošty, tak se k tokenu dostane a težko bude odhalen. V tomto případě nelze tak úplně hodit vinu na majitele dané nemovitosti, protože, i kdyby chtěl, obrana proti takovému útoku je mimo jeho možnosti.

Z tohoto důvodu by dávalo smysl tento typ kanálů ověřovat komplexněji a nespoléhat se pouze na verifikační token. Různé verifikační autority k tomuto mohou přistupovat různě a specificky pro rozdílné typy komunikačních kanálů, čímž mezi autoritami vzniká konkurenční prostředí s možností širší diverzifikace ověřovacích služeb.

## 7.3 Významnost záškodných autorit

Jako záškodnou autoritu uvažujme verifikační autoritu, která vědomě nedodržuje ověřovací protokol způsobem, který vede ke krádeži identity. Například autorita, která vydává ověřovací tokeny postranním kanálem nezávisle na vlastnictví adresy na ověřovaném kanále.

Výskyt záškodné autority může do značné míry snížit důvěru v celý systém. Bohužel to není něco, čemu lze se stoprocentní jistotou zabránit. Lze však podniknout kroky, které riziko sníží. Zde se můžeme ubírat dvěma směry – minimalizovat dopad výskytu, nebo minimalizovat výskyt.

### 7.3.1 Minimalizace dopadu

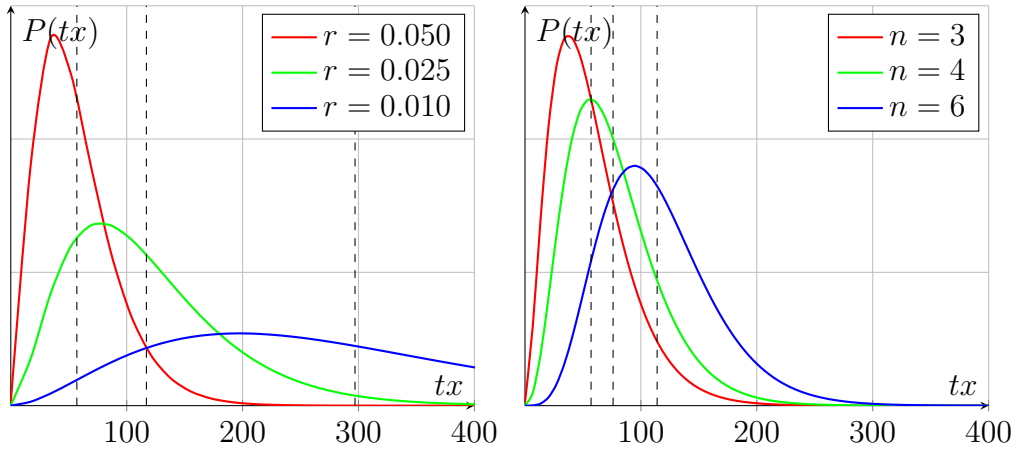
Jedním ze způsobů minimalizace dopadu výskytu záškodné autority je nasazení *3t* protokolu a snažit se maximalizovat počet autorit v systému. Kontrakt, který páruje uživatele s autoritou, musí vybírat autority co nejnáhodněji a nevybírá ty, kterými už byl uživatel ověřen. Míra důvěry v uživatelem proklamovanou identitu je pak závislá na počtu verifikací.

Počet, z pohledu útočníka, zbytečných transakcí (těch, které povedou k přidělení nezáškodné autority), které útočník musí provést, než se mu podaří získat ověření od $n$ záškodných autorit lze aproximovat negativním binomickým rozdělením. Tato aproximace bude počet transakcí spíše podhodnocovat (úspěšná verifikace je výběr bez vracení, což model zanedbává), přesnost se bude zvyšovat s celkovým počtem registrovaných autorit. Na obrázku 7.1a jsou vidět tři konfigurace útoku, cílem útočníka je získat tři falešná ověření. Konfigurace se liší pouze relativním počtem záškodných autorit v systému, varianty jsou 5%, $2,5$% a 1%.

Střední hodnoty jsou 57, 117, 297, z těchto výsledků lze pozorovat, že počet zbytečných transakcí je nepřímo úměrný procentu záškodných autorit.

Zdvojnásobím-li v systému počet autorit, procento nastrčených záškodných autorit se sníží zhruba o polovinu a útok se v průměru dvojnásobně prodraží. Tato strategie dává smysl pokud jsou v systému jednotky nebo desítky autorit, s rostoucím počtem začne razantně klesat výnosnost dalšího navyšování.

Alternativně se mohou bránit sami uživatelé zvýšením nároku na počet ověření, což lze vidět na obrázku 7.1b. V tomto případě roste očekávaný počet transakcí zhruba lineárně s vyžadovaným počtem ověření.




(a) $n = 3$, $r$ variabilní

(b) $n$ variabilní, $r = 5\%$

Obrázek 7.1: Pravděpodobnost odeslání $tx$ zbytečných transakcí před získáním $n$ ověření od záškodných autorit, jejichž relativní počet je $r$.

Z uvedených poznatků plyne, že *3t* protokol nelze řádně zabezpečit. Uvedené strategie útok v lepším případě zásadně prodraží a znepříjemní. V případě, že je nasazen *2t* protokol a autority tudíž nejsou přiřazovány, lze odpovědnost hodit na uživatele. Uživatel bude mít seznam autorit, kterým důvěřuje (protože je provozuje on sám nebo pro něj důvěryhodná osoba) a bude vyžadovat po účastnících konverzace, aby se u nich ověřili. V tomto případě si pak může být téměř jist, že ověření proběhlo korektně. Navíc je tento proces možné automatizovat na straně klienta např. přiložením automaticky generované patičky ke zprávě, která bude vyzývat k ověření u odesilatelem navolené autority, pokud tak adresát ještě neučinil.

### 7.3.2 Minimalizace výskytu

Efektivní obranou proti krádežím identit by bylo v systému záškodné autority vůbec nemít, čemuž se lze přiblížit co nejpřísnějším registračním procesem, případně hlasováním na bazí navrhovaného senátu. Tímto zde však může vzniknout úzké hrdlo – je snaha mít co nejvíc autorit, aby fungovala

náhoda, složitou registrací je však jejich počet tlačen dolu což jde proti výhodám *3t* protokolu. Nezávisle na entropii použitého generátoru náhodných čísel není výběr jedné z deseti zdaleka tak náhodný, jako výběr jedné ze sta.

V případě použití *2t* protokolu se senát jeví jako vhodné řešení, které sice není stoprocentní, ale určitě může představovat výraznou bariéru pro útočníka, který by chtěl do systému nasadit záškodnou autoritu.

## 7.4 Srovnání s protokolem Signal

Signal v tomto ohledu není příliš obecný a uživatelské identity váže na telefonní číslo, důkaz vlastnictví je proveden odesláním SMS zprávy s verifikačním tokenem. Dá se předpokládat, že subjekt, který tyto tokeny rozesílá je ve vlastnictví společnosti Signal a lze považovat za centralizovaný. Zde navrhovaný model je v tomto smyslu liberálnější, identita může být v podstatě cokoliv, na základě čeho lze s dotyčným komunikovat. Verifikační autority mohou mít obecně různé provozovatele a může jich být prakticky neomezené množství, decentralizace je zabudována přímo do protokolu (velmi dobře to je vidět na *3t* protokolu, ale platí to i pro *2t* protokol).

Další rozdíl je, že serverová část Signalu je opět centralizovaná a navíc i samotné konverzace jdou skrze tento server. Riziko DOS útoku je nutno brát na zřetel. Navíc sám server může cenzurovat své uživatele pokud se rozhodne, že jejich zprávy přestane doručovat. Je zde také riziko uplatnění *CLOUD Act*[2] pokud se na Signal vztahuje jurisdikce Spojených Států Amerických. Samozřejmě komunikace je koncově šifrována, takže prozrazení konverzací tímto nehrozí, samotná metadata však mohou prozradit mnoho.

Navržený model se těmto problémům docela dobře vyhýbá už jen tím, že nevyužívá centralizovaný komunikační kanál. Pokud tedy telefonní operátor účastníky konverzace odstřihne od sítě, pak pro ně není problém přesunou svou konverzaci na elektronickou poštu.

[2]Federální zákon umožňující úřadům vyžádat si data amerických společností nezávisle na tom, kde jsou fyzicky uložena.

# 8 Realizace

## 8.1 Zvolená protokolová sada

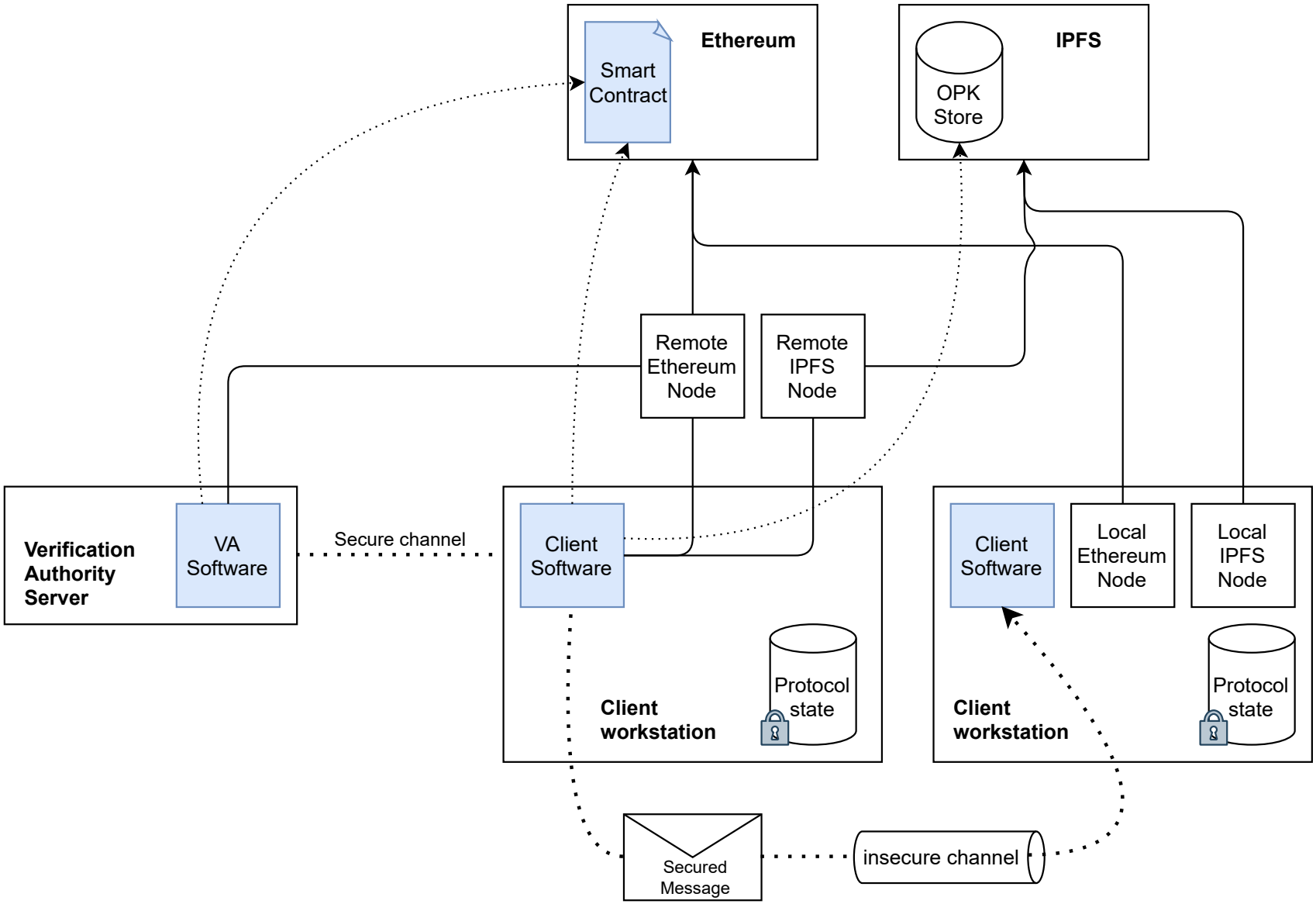
Algoritmy jsou voleny na základě analýzy v předchozí kapitole. Autentikace bude řešena použitím verifikačních autorit a představeného dvoutransakčního protokolu se skrytými identitami. Samotná komunikace bude probíhat na bázi protokolu Signal. Klíče pro X3DH handshake jsou uloženy na blockchainu s vyjímkou jednorázových, ty budou uloženy na IPFS. Chytrý kontrakt v tomto případě pracuje s relativní adresací klíčů (adresa, offset).

## 8.2 Architektura

Systém se bude skládat ze tří programových celků, těmi jsou klientská aplikace, chytrý kontrakt a server verifikační autority. Graficky je systém znázorněn na obrázku 8.1, kde je také naznačena jistá míra modularity co se týče umístění Ethereum a IPFS uzlů. V ideálním případě, pokud prioritizujeme bezpečnost, uživatel bude mít na své stanici uzel se synchronizovaným blockchainem a nepřetržitě dostupný IPFS uzel. Udržení synchronizovaného blockchainu je však náročné na prostředky, zejména na persistentní paměť, ale v případě iniciální synchronizace i na procesorový čas. Stejně tak lokální IPFS uzel nebude vždy dostupný, uživatel může být mobilní a není vždy připojen k internetu, případně svůj přístoj vypne, tato nedostupnost však může mít za následek nedostupnost jeho jednorázových klíčů pro uživatele, kteří by ho chtěli kontaktovat, pakliže je jiný uzel nemá v cache. V takovém případě se vytrácí vyžadovaná asynchronnost protokolu.

Z uvedených důvodů je praktičnější využít vzdálené uzly, které garantují dostupnost a synchronicitu s blockchainem. I když tento přístup může přinášet jistou úroveň centralizace do jinak decentralizovaného systému (více uživatelů sdílí jeden uzel), tak uživatel má stále možnost volby uzlu kterému věří, případně může provozovat vlastní vzdálené uzly.

Stavové proměnné, jako jsou nastavení západek a privátní klíče jsou vždy uloženy lokálně na klientské stanici a zabezpečeny heslem. Tento fakt může působit problematicky v případě, kdy uživatel vyžaduje použití více zařízení se stejnou konfigurací, řešení tohoto problému však přesahuje rozsah této práce.




Obrázek 8.1: Architektura systému s dvěmi variantami klientské instalace. Nalevo klient připojený ke vzdáleným uzlům, napravo klient s lokálními uzly.

## 8.3 Blockchain – Signal server

Chytré kontrakty jsou realizovány v jazyce Solidity ve verzi 0.8.1. Program se skládá ze tří volně provázaných celků – management identit, management uživatelů a senát. Kompletní objektový návrh je vidět na obrázku 8.2.

### 8.3.1 Návrhové vzory specifické pro chytré kontrakty

Psaní chytrých kontraktů je svým způsobem velmi specifiká činnost, která naráží na problémy se kterými se běžně v objektovém programování nesetkáváme. S několika takovými problémy jsem se při implementaci programu setkal a řešil je s použitím návrhových vzorů, které budou čtenáři pravděpodobně neznámy. Proto je zde ve stručnosti představím.

**Access Restriction (omezení přístupu)**

Nejedná se ani tak o návrhový vzor, jako obecný koncept. Každý smysluplný kontrakt má nějaké externí rozhraní, se kterým můžou ostatní kontrakty a uživatelé komunikovat pomocí transakcí. Problém nastává ve chvíli, kdy by některé metody tohoto volání neměl volat kdokoliv a kdykoliv – metodu lze spouštět pouze za určitých podmínek. Řešení spočívá ve využití klíčového slova `require`, případně v kombinaci s funkčními modifikátory (vizte sekce 4.5.3).

**Checks Effects Interaction (kontrola-změna-volání)**

Pokud kontrakt A volá metodu kontraktu B a kontrakt reaguje voláním „nazpět“ vzniká reentrantní smyčka. Tato situace je obecně nežádoucí, v lepším případě skončí vyčerpáním paliva, v horším případě lze takto vykrást kontrakt. Minimální implementace takového útoku s komentářem je uvedena na příkladu kódu 1. Útoku uvedeném na příkladu by se dalo předejít důslednou kontrolou ověření (má žadatel dostatečný zůstatek?), změnou stavu (nastavení zůstatku na 0), až následně přesunem prostředků na cizí adresu. Adresátův pokus o útok po této úpravě selže na kontrole omezení. Implementovaný kontrakt s peněžními prostředky nepracuje, návrhový vzor kontrola-změna-volání je zde využit jako obrana proti DOS útoku vyčerpáním klíčů, který by šel provést právě reentrantní smyčkou.

**Mutex (vzájemné vyloučení)**

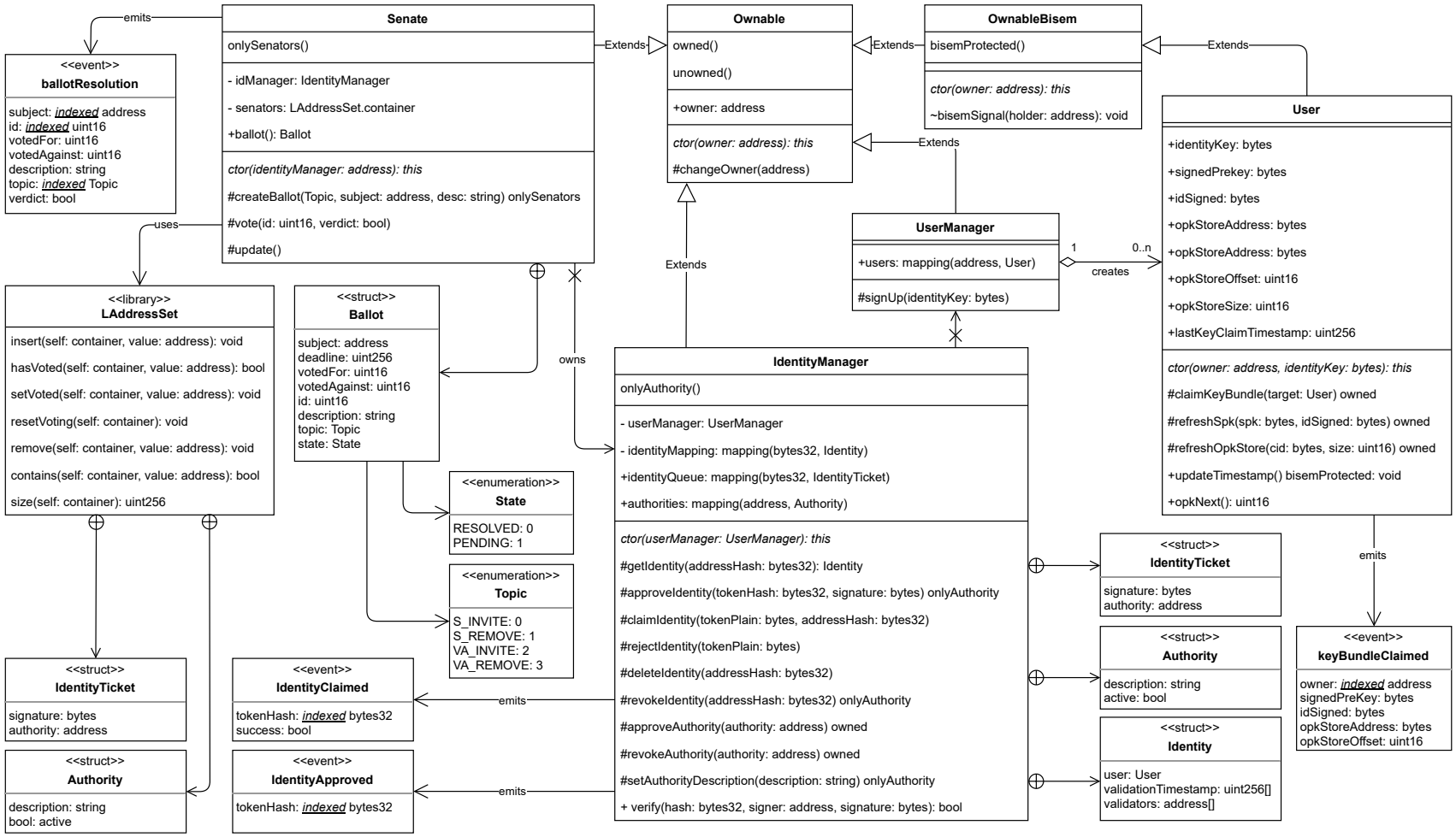
Mutex je návrhový vzor inspirovaný stejnojmenným synchronizačním primitivem. Využít lze dvěma způsoby – obrana před reentrantním útokem (těžkopádnější, avšak flexibilnější alternativa vzoru

kontrola-změna-volání, lze adresovat reentranci napříč vícero funkcemi). V tomto případě kontrakt A uzamyká svojí množinu funkcí pro kontrakt B (A je držitelem mutexu), v důsledku toho B nemůže volat „nazpět“. Druhou možností využití mutexu je předání pověření – kontrakt A má množinu funkcí vyžadující mutex, A předá kontraktu B mutex a následně mu předá i kontrolu volání některé jeho funkce. B pak může volat metody kontraktu A, které jsou uzamčené. Pokud se chceme v tomto případě bránit před reentrancí zamčených funkcí, je vhodné doplnit mutex o počítadlo vstupů a při překročení počtu vstupů mutex odebrat. Tato varianta použití mutexu je uvedena na diagramu 8.3 a je implementována kontraktem `OwnableBisem`.

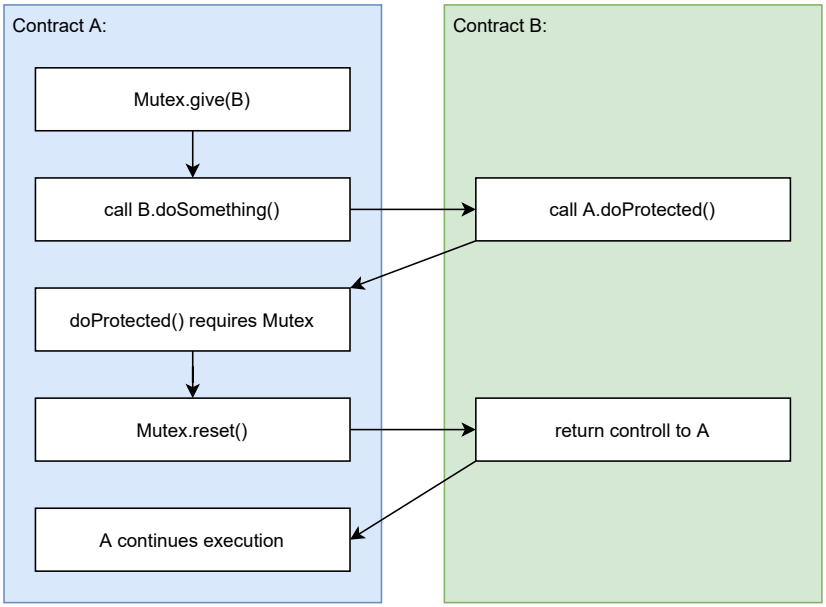
**Commit and Reveal Scheme (předlož a odhal)**

Z principu jsou data na blockchainu veřejná, pokud potřebujeme data utajit, nastává problém. Jako jednoduchý příklad uvedu, každému jistě známou, hru – kámen, nůžky, papír. Hraje-li spolu Bob a Alice, Alice by mohla švindlovat tak, že počká na Bobovu transakci a přečte si jeho tah např. „kámen“. Následně pošle svojí transakci obsahující „papír“, čímž podvodně vyhraje. Řešení je nasnadě, hru rozdělíme do dvou fází, první fáze – *commit*, zde hráči odevzdají hash kombinace jejich tahu a náhodné nonce[1]. Když všichni odevzdali své tahy, nastane druhá fáze – *reveal*, hráči odevzdají svůj tah a nonce, na blockchainu se reprodukuje hash a porovná se s odevzdaným. Pokud vypočtený hash s odevzdaným souhlasí, tak hráč provedl platný tah. Poté co všichni odevzdají své tahy kontrakt ohlásí vítěze. Stejně tak lze použít commit a reveal na hry typu „Na které číslo myslím?“, což se svým způsobem případ navrhovaného *2t protokolu*.

[1]Bez nonce by byly pouze tři platné hashe pro volby kámen, nůžky, papír – problém by se nevyřešil.




Obrázek 8.2: Objektový uml diagram, modifikátory přístupu jsou značeny +public, #external, ~internal, -private.




Obrázek 8.3: Příklad toku kontroly při použití návrhového vzoru mutex

### 8.3.2 Koncept vlastnictví kontraktu

Z diagramu 8.2 je patrné, že všechny kontrakty, ať už přímo nebo nepřímo, dědí od kontraktu `Ownable`. Tento kontrakt svým potomkům poskytuje modifikátor `owned()`, který aplikovaný na metodu revertuje transakci, pokud je odeslána někým jiným, než vlastníkem. Adresa vlastníka je předána parametrem konstruktoru a lze jí změnit metodou `changeOwner()`.

### 8.3.3 Správa uživatelů

Uživatelé jsou vytvářeni kontraktem `UserManager` pomocí tovární metody `signUp()`, vytvořená instance má nastaveného vlastníka na původního odesilatele transakce (předá se jako parametr kontruktoru). Uživatelem je zde myšlena instance kontraktu `User`, kterou si každý uživatel jednotlivě spravuje sám přes její externí rozhraní. `UserManager` adresu registrátora na nově vytvořenou instanci uživatele, vzniká tak jakýsi „telefonní seznam“ – podle adresy entity, která vydala požadavek na založení uživatelského účtu se lze dobrat k instanciovanému kontraktu. K obsluze kontraktu `User` jsou vystaveny tři metody které, jak ze signatury vyplývá, mohou být volány pouze vlastníkem (modifikátor `owned()`):

**`refreshSpk(spk, idSigned) owned`**

Aktualizuje střednědobý klíč `spk` a podpis identitního klíče `idSigned`.

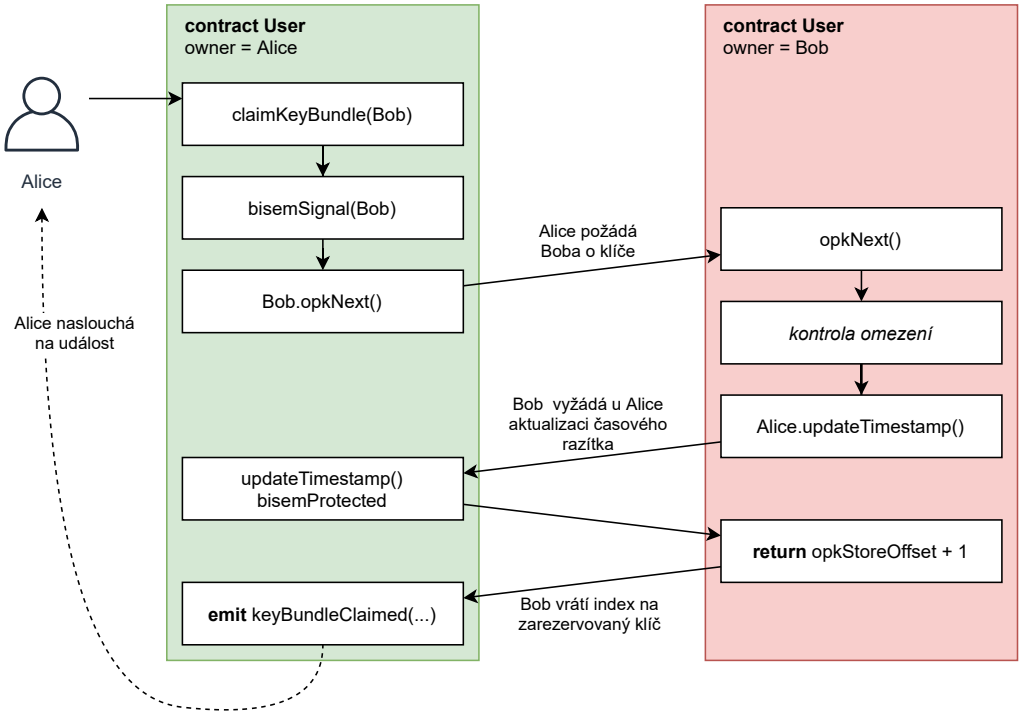
```
contract A~{
    // zůstatky uživatelů
    mapping(address => uint256) balance;

    // uživatel si může vybrat svůj zůstatek
    function withdraw() public {
        if (msg.sender.call.value(balance[msg.sender])()) {
            // v~tuto chvíli už se provádí kód útočníka
            // balance[msg.sender] má stále původní hodnotu
            balance[msg.sender] = 0;
            return;
        }
        revert();
    }
}

contract B {
    function withdraw(address _contract) public {
        // přetypuj adresu na A~a volej withdraw
        A(_contract).withdraw()
    }

    // fallback payable funkce se provede při odeslání
    // etheru na adresu tohoto kontraktu
    function() public payable {
        // pokud B má zůstatek nižší než 100 eth
        // pokračuj v~útoku
        if (address(this).balance < 100 ether) {
            // msg.sender bude v~tomto případě
A~withdraw(msg.sender)
        }
    }
}
```

Kód 1: Příklad útoku reentrantní smyčkou, útok je spuštěn pokud `B` zavolá `A.withdraw()`




Obrázek 8.4: Tok kontroly při rezervaci sady klíčů.

**`refreshOpkStore(cid, size) owned`**

Aktualizuje úložiště krátkodobých klíčů, `cid` je adresa úložiště na IPFS, `size` představuje kapacitu úložiště.

**`claimKeyBundle(target) owned`**

Zarezervuje balíček klíčů jiné instance kontraktu `User`. Samotný proces rezervace využívá návrhového vzoru mutex, který je implementován v rámci kontraktu `OwnableBisem`, od kterého `User` dědí. Metoda chráněná mutexem `updateTimestamp()` má modifikátor `bisemProtected()`. Zamluvení klíče je provedeno metodou `opkNext()`, ta nejprve ověří, zdali je možné klíč vydat (Nejsou klíče vyčerpány? Uběhla dostatečná časová rezerva od poslední rezervace žadatelem?) Pokud je možné pokračovat, zavolá se nazpět `updateTimestamp()`, tím adresát žadateli o klíč aktualizuje časové razítko. Adresát navrátí index rezervovaného klíče a žadatel emituje událost v níž je obsažena sada klíčů. Graficky je tento proces zobrazen na obrázku 8.4.

### 8.3.4 Správa identit

Správa identit je implementována v rámci kontraktu `IdentityManager`, ten interně kromě vlastníka zavádí další roli *autorita*. Hlavním cílem tohoto kontraktu je implementace navrhovaného *2t* protokolu a následné mapování verifikovaných identit na instance uživatelských kontraktů.

Na základě omezení přístupu lze externí rozhraní rozdělit na tři množiny metod. Prvně uveďme metody volatelné vlastníkem s modifikátorem `owned()`, jako vlastník tohoto kontraktu je předpokládán jiný kontrakt – `Senate`, který bude představen v další části textu. Tato skutečnost je zde explicitně zmíněna jako ilustrativní příklad toho, že v Ethereu platí zaměnitelnost externě vlastněných (uživatelských) a interních adres. Další podmnožinou jsou metody volatelné autoritou s modifikátorem `onlyAuthority()`. Zbytek metod bez modifikátoru je určen uživatelům spravujícím své identity. Následuje výčet metod externího rozhraní.

**`approveAuthority(address) owned`**
: Přidá adresu na interní seznam platných autorit, takto přidaná adresa splňuje restrikce definované modifikátorem `onlyAuthority()`

**`revokeAuthority(address) owned`**
: Interně označí adresu jako jako revokovanou autoritu, tato adresa již nesplňuje restrikce definované modifikátorem `onlyAuthority()`

**`approveIdentity(tokenHash, signature) onlyAuthority`**
: Autorita prohlašuje, že pokud uživatel, který vytvořil podpis adresy na ověřovaném kanále `signature`, tuto adresu skutečně vlastní, pak bude znát verifikační token, jehož hash je `tokenHash`. Volání této metody představuje druhou zprávu *2t* protokolu, zároveň lze také popsat jako *commit* fáze v *commit and reveal* schématu. Důsledkem tohoto volání je vytvořen `IdentityTicket`, obsahující přijatou signaturu a adresu autority, tento ticket je vložen do fronty neověřených identit `identityQueue` indexován jako `tokenHash`. Nakonec je emitována událost `IdentityApproved`.

**`revokeIdentity(addressHash onlyAuthority)`**
: Autorita zpochybňuje již v minulosti ověřenou identitu. Důsledkem je odejmutí validace volající autority pro danou identitu.

**`setAuthorityDescription(description)`**
: Nastavi volající autoritě `description` jako popisek. Tento popisek není povinný a implicitně je nenastaven. Autorita si zde může například uložit své doménové jméno, ale jedná se o obecnou řetězcovou položku.

**`claimIdentity(tokenPlain, addressHash)`**

Uživatel si přivlastňuje adresu skrze verifikační token `tokenPlain`, předává také hash adresy `addressHash` na ověřovaném kanále, který se ověří oproti podpisu který byl obdržen od autority. Toto volání představuje čtvrtou zprávu *2t* protokolu a *reveal* fázi *commit and reveal* schématu. Důsledkem tohoto volání je zavedení identity, pokud neexistuje, a přídání časového razítka s adresou oveřovací autority na seznam ověření dané identity. Časové razítko je aktuální k času této transakce, nikoliv k času transakce provedené autoritou (volání `approveIdentity()`). Nakonec je emitována událost `IdentityClaimed`, hodnota `success` je nastavena podle výsledku ověření podpisu.

**`rejectIdentity(tokenPlain)`**

Uživatel odepře validaci s využitím znalosti verifikačního tokenu `tokenPlain`.

**`deleteIdentity(addressHash)`**

Kompletně odstraní identitu na základě hashe adresy `addressHash`. Adresa volající tuto metodu musí být vlastníkem identity, resp. kontraktu `User` na který je identita vázána.

### 8.3.5 Senát

Senát je místo, kde se rozhoduje o zavádění nových a vyloučení již existujících verifikačních autorit, implementován je kontraktem `Senate`. Po nasazení kontraktu je v senátu veden pouze jeden senátor – adresa, ze které byl kontrakt nasazen. O přidávání dalších senátorů se hlasuje, vítězí absolutní nadpoloviční většina. Pouze v případě, kdy vyprší čas na rozhodnutí (jeden den od vyhlášení hlasování) vítězí nadpoloviční většina odevzdaných hlasů.

Hlasování je definováno strukturou `Ballot` a aktivní je vždy pouze jedno definováno veřejnou členskou proměnnou `ballot`. Hlasování se může nacházet ve stavu zahájeno (`State.PENDING`), nebo vyřízeno (`State.RESOLVED`). Nové hlasování lze vyvolat pouze pokud je existující hlasování vyřízeno.

**`createBallot(Topic, subject, desc) onlySenators`**

Senátor vyvolá nové hlasování, `Topic` říká o čem se bude hlasovat (přizvání, nebo vyloučení senátora, nebo autority), `subject` je pak adresa, které se hlasování týká (adresa potenciálního senátora, nebo autority). Poslední argument `desc` je poznámka pro ostatní senátory, kterou lze využít k stručnému dovysvětlení účelu hlasování.

`vote(id, verdict) onlySenators`

Senátor odevzdá svůj hlas – booleovská hodnota `verdict`, `true` pokud je „pro“ návrh. Zároveň předává pořadové číslo hlasování `id`, které slouží jako pojistka pro případ, že by se odhlasovalo a vyvolalo nové hlasování těsně před transakcí. Senátor by v této situaci hlasoval pro návrh, který už je odhlasován a jeho hlas by se propsal do návrhu nového pro který by třeba hlasoval opačně, či se hlasování zdržel úplně. Takto transakce v případě nesouhlasejícího pořadového čísla selže a k omylu tudíž nedojde.

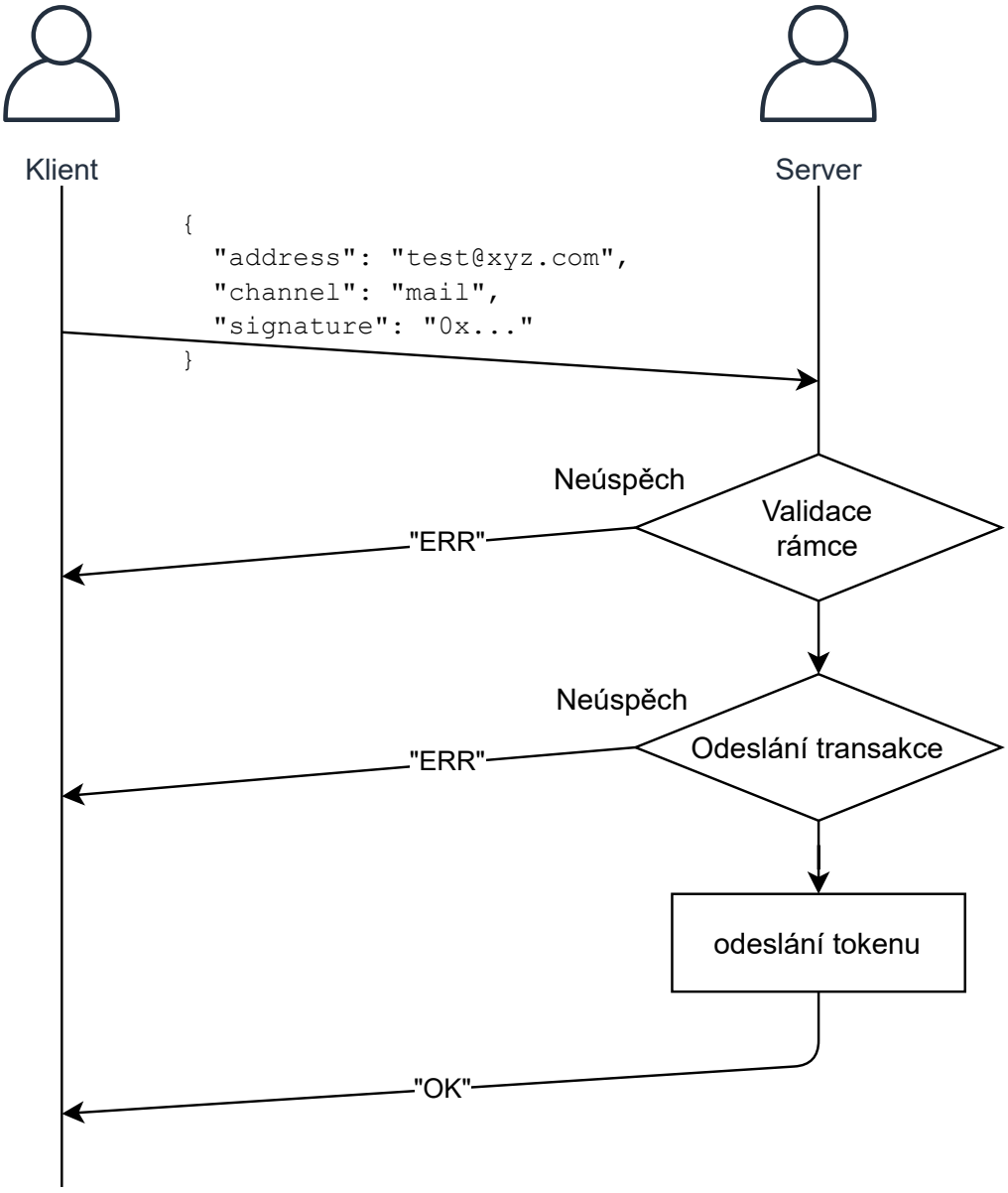
`update()`

Pokud návrh expiroval, provede vyhodnocení a uzavře hlasování, v opačném případě transakce selže. Tato metoda je zde nutná, protože kontrakt nedokáže reagovat na události (expirace návrhu), v takovém případě musí někdo rozhodovací proces „popostrčit“.

## 8.4 Server

Součástí řešení je serverová aplikace pro verifikační autoritu napsaná v jazyce JavaScript na platformě Node.js. Nejedná se o plnohodnotné řešení – odesílání verifikačního tokenu přes ověřovaný kanál implementováno nebylo, namísto toho se vypisuje na konzoli. Tato cesta byla zvolena z důvodu zachování obecnosti řešení, v případě reálného nasazení by pro autoritu neměl být problém doimplementovat odesílání tokenu přes několik málo kanálů které podporuje. Řešit zde tento problém ve vší obecnosti se nezdálo žádným způsobem výhodné a významným způsobem by se zkomplikovala implementace a následné testování. Proto je tento kus softwaru nutné brát spíše jako základ pro další vývoj než finální produkt.

Sám o sobě je program velmi jednoduchý skript komunikující s klienty přes websocket pomocí textového protokolu znázorněného na obrázku 8.5. Komunikaci zahájí klient odesláním adresy, jejího podpisu a typu kanálu ve formátu json. Server odpovídá řetězcem `OK` pokud se mu podařilo vygenerovat token a vypropagovat ho na blockchain. V opačném případě server odpovídá `ERR`.




Obrázek 8.5: Protokol pro domluvení verifikace

## 8.5 Klient

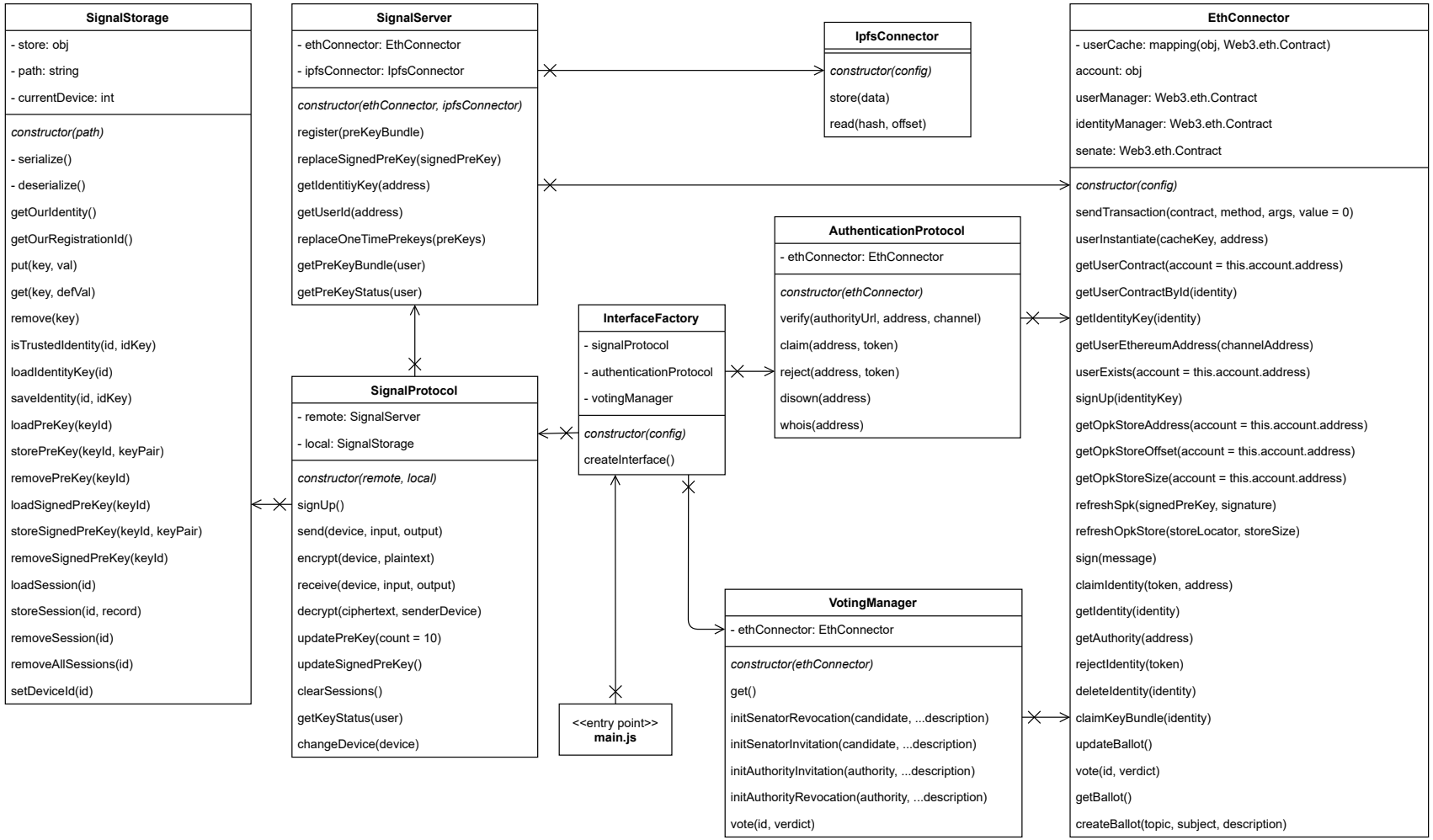
### 8.5.1 Konektory

Komunikace s IPFS a Ethereum uzlem jsou implementovány třídami `IpfsConnector` a `EthConnector`. V případě IPFS konektoru je spojení realizováno knihovnou `ipfs-http-client`. Klíče jsou na IPFS ukládány jako hexadecimální řetězce oddělené koncem řádky. `IpfsConnector` nabízí dvě metody pro přístup k úložišti jednorázových klíčů.

**`async store(data)`**
: Uloží již serializované klíče a vrátí content id souboru vytvořeného na IPFS.

**`async read(hash, offset)`**
: Přečte řádek ze souboru na IPFS. Content id je určeno parametrem `hash` a řádek je indexován parametrem `offset`.

EthConnector pro komunikaci s Ethereum uzlem využívá knihovnu `Web3.js`. Třída plní funkci adaptéru na externí rozhranní nasazených kontraktů (vizte 8.3). Metody jsou asynchronní a návratovou hodnotou je transakční hash u metod, které mění stav, případně obecná data u `view` metod.




Obrázek 8.6: Objektový diagram klientské aplikace

### 8.5.2 Modul Senate

Senát je implementován třídou `VotingManager`, která pouze obaluje `EthConnector` a přidává validaci uživatelských vstupů a formátování výstupu. Rozhraní je následující:

**`async get()`**
: Vrátí objekt obsahující probíhající hlasování, časová razítka a výčty jsou převedeny do čitelného formátu.

**`async init...(candidate, ...description)`**
: Čtyři metody pro vyvolání hlasování, za tečky lze doplnit SenatorRevocation, SenatorInvitation, AuthorityRevocation a AuthorityInvitation. Validuje vstupy a volá `Senate.createBallot()`. Návratovou hodnotou je transakční hash a id založeného hlasování.

**`async vote(id, verdict)`**
: Validuje vstup a volá `Senate.vote(id, verdict)`.

### 8.5.3 Modul Signal

Modul signal řeší samotnou komunikaci, použita je implementace Signal protokolu z knihovny `libsignal`. Tato knihovna vyžaduje implementaci tzv. Signal storage, které je realizováno třídou `SignalStorage`. Smyslem Signal storage je (nějakým způsobem) přechovávat interní datové struktury signal protokolu, tedy relace a klíče. `SignalStorage` serializuje tato data na disk, cesta k souboru je nastavena parametrem konstruktoru. Je nutné podotknout, že privátní klíče zde nejsou žádným způsobem zabezpečeny, na disk se ukládají v otevřeném textu.

Další částí modulu je `SignalServer`, který jak název napovídá, nahrazuje konvenční vzdálený server Signal protokolu. Implementovány jsou zde operace, které manipulují se vzdáleným úložištěm, ať už jde o blockchain či IPFS. Těmito operacemi jsou registrace uživatelů, aktualizace klíčů, rezervace klíčů, zároveň se lze na některá užitečná data dotazovat.

Jako rozhraní tohoto modulu slouží třída `SignalProtocol`, která propojuje funkcionalitu lokálního úložiště `SignalStorage` a vzdáleného úložiště `SignalServer` za účelem poskytnutí služeb Signal protokolu. Tyto služby jsou na nejvyšší úrovni odeslání šifrované zprávy – metoda `send()`, dále příjem takové zprávy – metoda `receive()`, registrace uživatele, pokud registrován ještě není – metoda `signUp()`, a aktualizace klíčů – metody `updatePreKey()` a `updateSignedPreKey()`.

Zajímavým implementačním detailem je řešení adresace uživatelů, která není s původní specifikací zcela kompatibilní. Signal, resp. Sesame, uživatele adresuje dvousložkovou adresou, která se skládá z `UserID` – telefonní číslo a `DeviceID` – identifikátor přístroje, který je unikátní pro uživatele, nikoliv však globálně. Zde je jako `DeviceID` brán 32bitový identifikátor generovaný z hashe identity (např. mailové adresy). Zároveň lze říci, že identita je unikátní (dáno implementací chytrého kontraktu), proto lze docela dobře počítat i s unikátností tohoto identifikátoru. Díky této vlastnosti může uživatel požadovat odeslání zprávy na adresu *x@y.com* aniž by znal ethereovou adresu majitele identity. Interně se ovšem dosazuje i `UserID` pro případ, že by vygenerované `DeviceID` bylo kolizní, což může nastat s významě vyšší, avšak stále poněkud mizivou, pravděpodobností, než kolize keccak512 hashe, kterým jsou adresy reprezentovány na blockchainu. Jako `UserID` je dosazena ethereová adresa majitele identity.

### 8.5.4 Modul Identity

Modul *Identity* sestává z třídy `AuthenticationProtocol`. Smyslem tohoto modulu je správa identit, zejména tedy implementace *2t* protokolu. *2t* protokol je implementován pomocí dvou metod `verify()` a `claim()`. První z nich jako argument příjmá URL verifikační autority, adresu na ověřenovaném kanálu a typ kanálu. Autoritě je odeslán požadavek na ověření této adresy, tento proces se řídí protokolem zmíněným v sekci 8.4. Verifikaci lze dokončit voláním `claim()`, argumenty jsou ověřovaná adresa a verifikační token obdržený od autority.

### 8.5.5 Uživatelské rozhraní

Třída `InterfaceFactory` z načtené konfigurace metodou `createInterface()` vytvoří příkazový strom. V souboru `main.js`, který je zároveň vstupním bodem programu, se na bázi tohoto stromu spouštějí příkazy načítané z konzole. Načtená řádka z konzole se nejprve rozdělí na slova, tato slova by měla tvořit cestu k listu stromu. Při nalezení této cesty se volá funkce referovaná jejím koncovým uzlem, pokud bylo dosaženo listu stromu před vyčerpáním všech slov, jsou zbylá slova předána dále jako argumenty volané funkce. Struktura zmíněného příkazového stromu je definována v souboru `Commands.js`.

Konzole funguje v režimu REPL[2]. Podrobnější údaje z hlediska použití a konfigurace lze nalézt v uživatelské příručce, příloha C.

[2]Read-eval-print loop – příkazy se vyhodnocují atomicky v pořadí v jakém byly zadány.

# 9 Oveření funkcionality

## 9.1 Integrační testy

Vývojové prostředí Truffle, které bylo použito pro vývoj chytrých kontraktů disponuje podporou testování na bázi testových kontraktů, tyto testy jsou psány v Solidity. Dále je možné psát testy v jazyce JavaScript, které fungují na bázi transakcí. Zvolena byla varianta javascriptových testů, která je spíše vhodná pro psaní integračních než jednotkových testů. Díky absenci podpory mockování se čistě jednotkové testy příliš dobře nepíší ani v Solidity – těžko lze izolovat jeden kontrakt a testovat ho nezávisle na zbytku. Výstup z analyzátoru pokrytí je vidět v tabulce 9.1, ve sloupcích jsou zobrazeny různé metriky pokrytí (pokrytí příkazů, větví, funkcí a řádek).

| File | % Stmts | % Branch | % Funcs | % Lines |
|---|---|---|---|---|
| contracts/ | 80.49 | 61.25 | 82.98 | 77.86 |
| IdentityManager.sol | 74.29 | 62.5 | 66.67 | 69.77 |
| LAddressSet.sol | 78.57 | 100 | 85.71 | 73.33 |
| Ownable.sol | 75 | 50 | 75 | 66.67 |
| OwnableBisem.sol | 100 | 50 | 100 | 100 |
| Senate.sol | 77.08 | 63.04 | 83.33 | 76.74 |
| User.sol | 100 | 60 | 100 | 100 |
| UserManager.sol | 100 | 50 | 100 | 100 |
| All files | 80.49 | 61.25 | 82.98 | 77.86 |

Tabulka 9.1: Pokrytí kódu chytrých kontraktů vyjádřeno v procentech s využitím různých metrik: pokrytí *Stmts* – příkazů, *Branch* – větví, *Funcs* – metod, *Lines* – řádků.

## 9.2 Testové scénáře

Funkčnost systému jako celku je ověřena pomocí předpřipravených scénářů. Definice scénářů a záznam konzole lze nalézt v příloze D. První scénář slouží k přípravě prostředí, následující scénáře počítají, že první scénář byl úspěšně proveden.

# 10 Zhodnocení

Bakalářská práce představila vlastnosti vyžadované po systému zprostředkujícím zabezpečenou komunikaci a seznámila čtenáře s principy existujících řešení PGP, SSL/TLS a Signal. Dále byl naznačen princip fungování blockchainu, sítě Ethereum, chytrých kontraktů a sítě IPFS.

Jako hlavní cíl této práce bylo využít nabytých znalostí k návrhu systému pro bezpečnou komunikaci, který jako důsledek využití blockchainu nabude dalších výhodných vlastností, které současné systémy nenabízejí a z principu ani nemohou. Jako základ implementovaného systému byl vybrán protokol Signal. Navrženy byly dva automatizované protokoly pro verifikaci vlastnictví adresy na obecném komunikačním kanále, které byly z pohledu bezpečnosti kriticky zhodnoceny. Jako komunikační protokol je využita kombinace X3DH a Double Ratchet algoritmu, které jsou součástí specifikace Signalu.

Následně byly vytvořeny tři demonstrační aplikace – Signal server, nasazen jako chytrý kontrakt na blockchain, server verifikační autority a klientská aplikace. Případné další kroky pro potenciální ostré nasazení by spočívaly v implementaci systému pro destrukci, či aktualizaci kontraktů a provedení bezpečnostního auditu. V případě serveru verifikační autority je nutno doimplementovat rozesílání tokenů po verifikovaných kanálech. Konečně v klientské aplikaci je nutné zabezpečit privátní klíče protokolu uložené na disku a místo načítání Ethereum účtu přes privátní klíč implementovat podporu peněženek. Protože se ve vytvořeném systému střetává nabídka s poptávkou (autority nabízejí ověření jako službu, uživatelé nabízí své klíče), dalším krokem by bylo vytvoření ekonomiky nad vlastním ERC-20 tokenem[1].

[1]ERC-20 je standardizované rozhraní chytrého kontraktu umožňující implementaci obchodovatelných tokenů – velmi zjednodušeně řečeno vlastní kryptoměny uvnitř sítě Ethereum.

# Seznam zkratek

**AD** Asymmetric decryption. 10

**AE** Asymmetric encryption. 10

**AES** Advanced Encryption Standard. 14

**CBC** Cipher block chaining. 14

**CID** content id. 25, 72

**DOS** denial of service. 43, 46

**ECDH** Elliptic Curve Diffie-Hellman. 14, 15

**ECDSA** Elliptic Curve Digital Signature Algorithm. 20

**EVM** Ethereum virtual machine. 20, 21, 29, 39

**GPG** Gnu Privacy Guard. 8

**IK** Identity key. 15

**IPFS** Interplanetary File System. 25, 36, 37, 38, 44, 51, 56, 58, 61, 71, 72, 77

**KDF** Key Derivation Function. 16, 17

**MAC** Message authentication code. 5

**OPK** One-time prekey. 15

**PGP** Pretty Good Privacy. 4, 8, 9, 11, 12, 28, 61

**RNG** Random number generator. 10

**SD** Symetric decryption. 10

**SE** Symetric encryption. 10

**SHA** Secure Hash Algorithm. 14

**SPK** Signed prekey. 15

**X3DH** Extended Triple Diffie-Hellman. 14, 38

# Literatura


[1] IPFS documentation. Dostupné z: https://docs.ipfs.io/.

[2] Validating other keys on your public keyring. Dostupné z: https://www.gnupg.org/gph/en/manual/x334.html.

[3] Borisov, N. – Goldberg, I. – Brewer, E. Off-the-record communication, or, why not to use PGP. *Proceedings of the 2004 ACM workshop on Privacy in the electronic society - WPES 04*. 2004. doi: 10.1145/1029179.1029200.

[4] CoinMarketCap. *Historical snapshot – 25 April 2021* [online]. 2021. Dostupné z: https://coinmarketcap.com/historical/20210425/.

[5] Finney, H. et al. OpenPGP Message Format. RFC 4880, November 2007. Dostupné z: https://rfc-editor.org/rfc/rfc4880.txt.

[6] Harn, L. et al. Fully Deniable Message Authentication Protocols Preserving Confidentiality. *The Computer Journal*. 2011, 54, 10, s. 1688–1699. doi: 10.1093/comjnl/bxr081.

[7] Marlinspike, M. The Double Ratchet Algorithm. Technical report, November 2016. Dostupné z: https://www.signal.org/docs/specifications/doubleratchet/doubleratchet.pdf.

[8] Marlinspike, M. The Sesame Algorithm: Session Management for Asynchronous Message Encryption. Technical report, April 2017. Dostupné z: https://www.signal.org/docs/specifications/sesame/sesame.pdf.

[9] Marlinspike, M. The X3DH Key Agreement Protocol. Technical report, November 2016. Dostupné z: https://www.signal.org/docs/specifications/x3dh/x3dh.pdf.

[10] Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. Dostupné z: https://bitcoin.org/bitcoin.pdf.

[11] Reutov, A. Predicting Random Numbers in Ethereum Smart Contracts, May 2018. Dostupné z: https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620.

[12] Wood, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*. 2020, 3e2c089. Dostupné z: https://github.com/ethereum/yellowpaper.

[13] Zimmermann, P. – Atkins, D. – Stallings, W. PGP Message Exchange Formats. RFC 1991, August 1996. Dostupné z: https://rfc-editor.org/rfc/rfc1991.txt.

# Přílohy

# A Manuál: chytré kontrakty

## A.1 Kompilace a migrace

Kód je psán v výhradně v jazycích Javascript na platformě Nodejs (integrační testy) a Solidity (chytré kontrakty). Chytré kontrakty jsou vyvíjeny ve vývojovém prostředí *Truffle* a lokálním blockchainu *Ganache*. Oba zmíněné programy lze získat v aktuální verzi přes *npm* (node package manager):

```
$ npm i -g truffle ganache-cli
```

Před nasazením je nutné spustit ganache příkazem:

```
$ ganache-cli -dp 8545 -h 127.0.0.1
```

Příznak `-d` říká, že blockchain bude deterministický, v tomto případě bude nutné dělat pouze minimální úpravy v předpřipravené konfiguraci klientské aplikace a serveru. Příznaky `-p` a `-h` specifikují port a host na kterém bude ganache naslouchat, zvolené hodnoty jsou nastavené v konfiguraci Truffle. Případně je lze změnit v souboru `blockchain/truffle-config.js`, sekce `networks.development`. V témž souboru lze nakonfigurovat i nasazení na *mainnet*[1], ale to je mimo rozsah tohoto počinu. Když Ganache běží, lze nasadit kontrakty, to je z adresáře `blockchain` provedeno příkazem:

```
$ truffle migrate
```

Před samotnou migrací se automaticky provede kompilace, která může napoprvé chvíli trvat – Truffle si musí stáhnout kompiler. V konzoli si lze všimnout jednotlivých migračních transakcí, důležité zde jsou zejména adresy kontraktů, na ty bude klient a server posílat své transakce.

## A.2 Spuštění testů a pokrytí

Testy lze spustit z adresáře `blockchain` na běžícím Ganache blockchainu příkazem:

[1] Oficiální Ethereum blockchain

```
$ truffle test
```

Praktičtější je však využít vývojovou konzoli Truffle, která má zabudovaný jednorázový blockchain. Do konzole se lze dostat a následně spustit testy příkazy:

```
$ truffle devel
> test
```

Analyzátor pokrytí si vytvoří vlastní blockchain na portu 9545 (musí být volný), takže Ganache běžet nemusí. Spustit ho lze příkazem:

```
$ truffle run coverage
```

Grafický výstup analyzátoru pokrytí se uloží do adresáře `blockchain/coverage` ve formátu html (soubor `index.html`).

# B Manuál: server

## B.1 Spuštění

Program není nutné kompilovat, před prvním spuštěním je pouze nutné stáhnout závislosti. To lze provést přes *npm*:

```
$ cd server
$ npm i
```

Pro spuštění je nutný Nodejs ve verzi 15. Z kořenového adresáře projektu lze server spustit v systémovém shellu příkazem:

```
$ node server/src/main.js <konfigurace>
```

Argument `<konfigurace>` je nutno nahradit cestou ke konfiguračnímu souboru. Více informací o konfiguraci naleznete v sekci B.3 Alternativně lze použít spouštěcí skript `run.sh`:

```
$ ./run.sh server <konfigurace>
```

## B.2 Výstup

Po spuštění server čeká na příchozí požadavky. Po obdržení požadavku je vygenerován verifikační token a provedena transakce. Po vyřízení transakce se na konzoli vypíše hash transakce a verifikační token s identifikací požadavku. Příklad takového výstupu je vidět níže.

```
0xe8734b0ccc65d067...
Request test@x.y, mail resolved into token:
83d901c283c7c473136c3b03a8f8da1f
```

## B.3 Konfigurace

Program vyžaduje cestu ke konfiguračnímu souboru jako argument příkazové řádky, který je povinný. Předpřipravenou konfiguraci lze získat ze

souboru `server/src/default_config.json`. Soubor musí být striktně ve formátu json. Možnosti konfigurace jsou následující:

**`server.port`** Port na kterém server bude naslouchat.

**`ethereum.url`** Url Ethereum uzlu ke kterému se server připojí.

**`ethereum.privkey`** Hexadecimálně zapsaný privátní klíč, který bude použit k podepisování transakcí. Platí, že tento klíč musí patřit k adrese, která je vedena jako adresa autority, jinak ověřování nebude fungovat.

**`ethereum.address`** Adresa nasazeného kontraktu `IdentityManager`.

# C Manuál: klientská aplikace

Spuštěním programu se uživatel dostane to příkazového rozhraní s podporou doplňování tabulátorem, odejít lze příkazem `quit`. Aplikace se skládá ze tří základním modulů *signal*, *identity* a *senate*. Obecně má každý požadavek tvar `modul příkaz argumenty`.

Pro popis argumentů příkazů v jednotlivých modulech bude použita jednoduchá notace. Ostré závorky `<>` definují povinný argument, hranaté závorky `[]` definují nepovinný argument, tečky (`...`) značí víceslovný argument. Absence závorek značí výčet, jednotlivé možnosti jsou odděleny svislítkem, např. `a|b`.

## C.1 Spuštění

Program není nutné kompilovat, před prvním spuštěním je, stejně jako u serveru, nutné stáhnout závislosti. To lze provést přes *npm*:

```
$ cd client
$ npm i
```

Z kořenového adresáře projektu lze program spustit příkazem (vyžadován je Nodejs ve verzi 15 a vyšší):

```
$ node client/src/main.js <konfigurace> 2>/dev/null
```

Místo `<konfigurace>` uveďte cestu ke konfiguračnímu souboru. Program při nestandardních situacích vypisuje do chybového výstupu stacktrace, pokud je toto chování nežádoucí je nutné chybový výstup přesměrovat. Alternativně lze použít spouštěcí skript `run.sh`:

```
$ ./run.sh client <konfigurace>
```

Více o konfiguračních souborech a možnostech konfigurace naleznete v sekci sekci C.6.

## C.2 Modul signal

Tento modul má na starost registraci uživatelů, správu klíčů a uživatelskou komunikaci nad protokolem Signal. V tabulce C.1 je vidět kompletní výčet příkazů tohoto modulu v četně jejich argumentů, podrobněji jsou popsány v následném textu.

Signal modul pracuje s *identitami* a *uživateli*, jejich správa je součástí modulu *identity* (vizte C.3). Část operací tohoto modulu se váže na uživatelskou identitu, kterou lze nastavit příkazem *setdev*, těmito operacemi jsou zejména příjem a odesílání zpráv. Každá identita má pak svou nezávislou množinu relací které slouží ke komunikaci s identitami jiných uživatelů. Relace mezi uživatelem a identitou je 1:n, operace jako registrace a obnovy klíčů se váže na uživatele nikoliv konkrétní identity.

| příkaz | argumenty |
|---|---|
| `register` | |
| `send` | `<identita> <vstup> [výstup]` |
| `receive` | `<identita> <vstup> [výstup]` |
| `status` | |
| `clear` | |
| `setdev` | `<identita>` |
| `update opk` | `[počet]` |
| `update spk` | |

Tabulka C.1: Přehled příkazů modulu signal

**`register`**

Zaregistruje uživatele a nahraje na blockchain jeho privátní klíče. Registrace je vázána na konkrétní účet (privátní klíč) v síti Ethereum a je nezvratná. Operace je jednorázová a provede tři transakce. Výstupem jsou hashe provedených transakcí.

**`send <identita> <vstup> [výstup]`**

Vytvoří šifrovanou zprávu adresovanou uživateli s konkrétní *identitou* (vizte sekce C.3). Zpráva v otevřeném textu je načtena ze souboru určeném argumentem vstup a zabezpečená zpráva je uložena do souboru dle argumentu výstup, který je nepovinný. Při navazování relace se provádí transakce související s X3DH výměnou klíčů, taktéž je nutný přístup k IPFS uzlu (alespoň pro čtení). Po navázání relace (zpracování první odpovědi adresáta) se již transakce neprovádí. Výstupem

příkazu je hash provedené transakce (pokud byla provedena) a zabezpečená zpráva určená k odeslání adresátovi.

**`receive <identita> <vstup> [výstup]`**
Přijme zprávu od uživatele určeného jeho *identitou*, uloženou v souboru *vstup*. Zpráva v otevřeném textu se uloží do souboru *výstup*. Tato operace nevytváří transakce, pouze volání (nelze provést offline). Výstupem je oteřený text obsažený v předané zprávě.

**`status`**
Vypíše status jednorázových klíčů, tj. jejich celkový počet, počet využitých, zbývající a CID úložiště klíčů. Tento příkaz nevytváří transakce.

**`clear`**
Tento příkaz fyzicky smaže všechny relace pro aktivovanou identitu. Tímto se zprávy přijaté na základě těchto relací stávají nečitelné a při komunikaci opačným směrem je opět potřeba provést výměnu klíčů. Tato operace negeneruje transakce.

**`setdev`**
Aktivuje identitu, veškerá následující komunikace bude probíhat pod touto identitou. Operace negeneruje transakce.

**`update opk [počet]`**
Vygeneruje zadaný počet jednorázových klíčů, v případě že počet není stanoven, použije se implicitní počet nastavený v konfiguraci. Vygenerované klíče se uloží na IPFS a prostřednictvím CID také na blockchain, současné úložiště klíčů je touto operací nahrazeno. Tento příkaz generuje transakci a vyžaduje přístup k IPFS uzlu. Výstupem je hash provedené transakce.

**`update spk`**
Vygeneruje nový střednědobý klíč a nahraje ho na blockchain. Operace generuje transakci. Výstupem je hash provedené transakce.

## C.3 Modul identity

Modul *identity* slouží ke správě uživatelských identit. Přehled dostupných příkazů je k vidění v tabulce C.2 a podrobněji jsou popsané v zbytku této příručky.

Slovem *identita* je zde označena ověřená adresa na libovolném komunikačním kanále. Identitu vlastní konkrétní uživatel a je z principu fungování systému unikátní, tedy existující identita identifikuje konkrétního uživatele.

| příkaz | argumenty |
|---|---|
| `verify` | `<url autority> <adresa na kanálu> <typ kanálu>` |
| `claim` | `<adresa na kanálu> <verifikační token>` |
| `reject` | `<adresa na kanálu> <verifikační token>` |
| `disown` | `<adresa na kanálu>` |
| `whois` | `<adresa na kanálu>` |

Tabulka C.2: Přehled příkazů modulu identity

**`verify <url autority> <adresa na kanálu> <typ kanálu>`**
Odešle ověřovací požadavek verifikační autoritě. Autorita bude reagovat odesláním verifikačního tokenu na požadovanou adresu na specifikovaném kanálu. Přítomnost podpory konkrétních přenosových kanálů je v kompetenci dané autority, každá autorita si definuje množinu kanálů, které podporuje. Při zadávání tohoto příkazu je typ kanálu určen klíčovým slovem, např. *mail* pro elektronickou poštu. Tato akce nevytváří transakce.

**`claim <adresa na kanálu> <verifikační token>`**
Tímto příkazem se dokončí verifikace proti tokenu obdrženém od autority jako adresu kanálu je nutno uvést adresu, na kterou token dorazil. Tato operace vytváří transakci, výstupem je hash této transakce.

**`reject <adresa na kanálu> <verifikační token>`**
Obdoba *claim*, avšak místo potvrzení dojde k invalidaci požadavku. Efekt je ve výsledku stejný jako v případě volání claim s nevalidním tokenem, avšak poplatek za transakci bude nižší. Výstupem je hash odeslané transakce.

**`disown <adresa na kanálu>`**
Odstraní již existující identitu. Výstupem je hash odeslané transakce.

**`whois <adresa na kanálu>`**
Odešle dotaz na konkrétní adresu. Výstupem je objekt obsahující položku *Ethereum* – adresa uživatele v síti Ethereum a objekt identity *Identity* obsahující položku *user* – adresa uživatelského kontraktu a

položku *validated* – pole validací, jehož prvek se skládá z adresy autority na blockchainu a časového razítka dané verifikace. Tato operace nevytváří transakce.

```
> identity whois pepa@novak.cz
{
    userId: '0x90F8bf6A479f320ead074411a4B0e7944Ea8c9C1',
    Identity: {
        user: '0x79183957Be84C0F4dA451E534d5bA5BA3FB9c696',
        validated: [
                '0x1dF62f...: Sat Apr 17 2021 15:20:03...',
                '0x95cED9...: Sat Apr 18 2021 12:30:10...',
                '0x1dF62f...: Sat Apr 18 2021 19:20:32...'
        ]
    }
}
```

Výstup C.1: Ukázka možného výstupu příkazu *whois*, obsah pole *validated* je zkrácen, chybějící text je nahrazen tečkami

## C.4 Modul senate

Modul *senate* umožňuje interakci se senátem na blockchainu, hlasovat a vytvářet nová hlasování však mohou pouze senátorské adresy. Pro běžného uživatele je zde zajímavý pouze příkaz *print*, kterým lze vypsat stav aktuálního hlasování. Výčet dostupných příkazů si lze prohlédnout v tabulce C.3

| příkaz | argumenty |
|---|---|
| `init` | `invite\|revoke authority\|senator <adresa> [...poznámka]` |
| `vote` | `<id> accept\|reject` |
| `print` | |

Tabulka C.3: Přehled příkazů modulu senate

**`init invite|revoke authority|senator <adresa> [...poznámka]`**
: Založí nové hlasování, jeho účel je dán výběrem klíčových slov *invite*, nebo *revoke*, tedy pozvat, nebo odvolat. K jaké pozici se pozvánka, nebo případné odvolání je specifikováno slovy *authority*, nebo *senator*, tedy verifikační autorita, nebo senátor. Hlasování se vždy týká konkrétní ethereové adresy a je doplněno nepovinnou poznámkou, která je tvořena několika slovy oddělenými mezerou, které se propíšou na blockchain. Iniciací hlasování senátor nedává najevo svůj verdikt, k tomu slouží příkaz *vote*. Tato operace generuje transakci, její hash se propíše na výstup spolu s *id* založeného hlasování.

**`vote <id> accept|reject`**
: Hlasuje pro nebo proti (*accept*, nebo *reject*) aktuálnímu návrhu. Také je nutné předložit *id* návrhu. Operace generuje transakci, její hash se propíše na výstup.

**`print`**
: Vypíše podrobné informace o aktuálním hlasování včetně jeho identifikátoru a počtu hlasů, ukázka na obr. C.2. Tato operace nevytváří transakci.

```
> senate print
Result {
    subject: '0x1dF62f291b2E969fB0849d99D9Ce41e2F137006e',
    deadline: 2021-04-18T13:19:06.000Z,
    votedFor: '1',
    votedAgainst: '0',
    id: '1',
    description: 'Pozvánka pro Pepíka',
    topic: 'Authority invitation',
    state: 'Resolved'
}
```

Výstup C.2: Ukázka výstupu příkazu *print*

## C.5 Modul help a vestavěná nápověda

Stručnou referenci k jednotlivým příkazům je monžné vypsat přímo v textovém rozhraní uvedením otazníku (`?`) jako jediného argumentu. Například nápověda k příkazu *verify* z modulu *identity* lze vypsat požadavkem `identity verify ?`. Stromovou strukturu dostupných volání lze vypsat pomocí příkazu `help tree`, příkazem `help info` se vypíše stručná nápověda.

## C.6 Konfigurace

Program si při spuštění načítá konfiguraci ze souboru, který je uveden jako povinný parametr. Konfigurační soubor se musí striktně držet formátu json, v opačném případě nebude konfigurace načtena a program se ukončí. Šablonu konfiguračního souboru lze vykopírovat z `client/src/default_config.json`.

**`ethereum.node.protocol`** Protokol přes který se klient připojí k Ethereum uzlu, možné je použít websocket (ws), http, případně jejich zabezpečené varianty. Při použití http může být problém s naslouchánímm událostí, proto je lepší použít websocket.

**`ethereum.node.host`** IP adresa, nebo doménové jméno Ethereum uzlu

**`ethereum.node.port`** Port na kterém naslouchá Ethereum uzel

**`ethereum.privkey`** Privátní klíč, který bude podepisovat transakce

**`ethereum.contracts.Senate`** Adresa nasazeného kontraktu Senate.sol

**ethereum.contracts.UserManager** Adresa nasazeného kontraktu UserManager.sol

**ethereum.contracts.IdentityManager** Adresa nasazeného kontraktu IdentityManager.sol

**ipfs.node.protocol** Protokol pro připojení k IPFS uzlu

**ipfs.node.host** IP adresa, nebo doménové jméno IPFS uzlu

**ipfs.node.port** Port na kterém IPFS uzel naslouchá

**signal.storagePath** Cesta k signal úložišti

**signal.defaultKeyBatchSize** Počet jednorázových klíčů, které se budou generovat při registraci uživatele nebo obnovení klíčů v případě, že argumentem není specifikováno jinak.

# D Testové scénáře

## D.1 Docker

Pro snazší přípravu prostředí je možné využít Docker. Sestavení image a spuštění kontejneru se z adresáře `Dockerfile` provede příkazy:

```
$ docker build . -t blockchain
$ docker run --rm -it --name bc blockchain
```

Nasazení kontraktů lze provést příkazem:

```
$ docker exec -t bc sh -c \
"cd blockchain && truffle migrate"
```

Spuštění serveru verifikační autority:

```
$ docker exec -it bc ./run.sh server config/server.json
```

Spuštění první a druhé instance klienta:

```
$ docker exec -it bc ./run.sh client config/c1.json
$ docker exec -it bc ./run.sh client config/c2.json
```

## D.2 Instrukce

Následují instrukce pro testování, u každého scénáře jsou v závorce odkazovány výpisy z terminálu. Kopírovatelné příkazy jednotlivých scénářů jsou v souboru `scenare.txt`.

1. Příprava prostředí (výstupy D.3, D.4, D.5)
   (a) Spusť instanci klienta (dále `C1`) ze senátorské adresy (konfigurace `config/c1.json`).
   (b) Spusť server (konfigurace `config/server.json`)
   (c) Z instance `C1` odhlasuj přijetí adresy serveru jako autority.
   (d) Z instance `C1` proveď registraci uživatele.

(e) Z instance `C1` ověř mailovou adresu test1@x.y.

(f) Spusť instanci klienta (dále `C2`) s jiným signal úložištěm a adresou (konfigurace `config/c2.json`)

(g) Z instance `C2` proveď registraci uživatele

(h) Z instance `C2` ověř mailovou adresu test2@x.y.

(i) Operace proběhly úspěšně a výpis identit test1@x.y a test2@x.y odpovídá očekávání

2. Navázání relace (výstupy D.6, D.7)

(a) Z instance `C1` nastav aktivní identitu na test1@x.y

(b) Z instance `C2` nastav aktivní identitu na test2@x.y

(c) Pokud mají klienti aktivní relace, smaž je.

(d) Z instance `C1` připrav šifrovanou zprávu pro test2@x.y

(e) Z instance `C2` dešifruj zprávu od test1@x.y

(f) Zpráva se dešifruje dle očekávání

3. Asynchronní komunikace (výstupy D.8, D.9)

(a) Z instance `C1` nastav aktivní identitu na test1@x.y.

(b) Z instance `C2` nastav aktivní identitu na test2@x.y.

(c) Pokud mají klienti aktivní relace, smaž je.

(d) Z instance `C1` připrav šifrovanou zprávu pro test2@x.y.

(e) Z instance `C2` tuto zprávu dešifruj.

(f) Z instance `C2` připrav několik šifrovaných zpráv.

(g) Z instance `C1` tyto zprávy dešifruj v opačném (nebo nahodilém) pořadí.

(h) Zprávy byly úspěšně dešifrovány.

4. Obnova klíčů (výstup D.10)

(a) Vytiskni status klíčů

(b) Z instance `C2` obnov jednorázové klíče.

(c) Status reflektuje nový stav po aktualizaci klíčů.

5. Přizvání senátora (výstupy D.11, D.12)

(a) Z instance `C1` odhlasuj přizvání `C2` v roli senátora.

(b) Z instance `C2` založ hlasování o přijetí dalšího senátora.

(c) Hlasování bylo úspěšně založeno.

6. Odmítnutí identity (výstupy D.13, D.14)

   (a) Z instance `C1` inicializuj ověření nové identity.

   (b) Po obdržení tokenu identitu odmítni.

   (c) Vypsání identity skončí chybou (identita neexistuje)

7. Zřeknutí se identity (výstup D.15, D.16)

   (a) Z instance `C1` ověř novou identitu.

   (b) Ověřené identity se zřekni.

   (c) Vypsání identity skončí chybou (identita neexistuje)

## D.3 Konzole

```
> senate print
> senate init invite authority 0x1dF6... pozvánka
0xea180f27890d7ee18fb5062c81535344f55094a188dd1b6b326cffbf4b5ac3dd
1
> senate vote 1 accept
0x9c7e61d2c063f5bd0b124ce046ef06fa6119ee9bb37c18517c54fc4a3db4a026
> senate print
Result {
    subject: '0x1dF62f291b2E969fB0849d99D9Ce41e2F137006e',
    deadline: 2021-04-29T10:46:08.000Z,
    votedFor: '1',
    votedAgainst: '0',
    id: '1',
    description: 'pozvánka',
    topic: 'Authority invitation',
    state: 'Resolved'
}
> signal register
0x2b63cf0a5646102756b698a0cccb3a4d6417f4ebf424aa40c7ee7b5b28e5bc27
0x56e7d06c5ed1c49dafc814d1a05c1adf8d2dbd37539d17bc53802f3df375b647
0x34f0d3d16568c4061074aa0937574feff7a6dcf8ef33e8d8b4589afc8beeddee
> identity verify ws://localhost:8080 test1@x.y mail
> identity claim test1@x.y 0a1cef6a46f37a3981d91a08b7061d88
0x336e45d891d2a9e677f23aa45ec286a325d602f312a9e468521cd02b133a31d2
> identity whois test1@x.y
{
    Ethereum: '0x90F8bf6A479f320ead074411a4B0e7944Ea8c9C1',
    Identity: {
        user: '0x79183957Be84C0F4dA451E534d5bA5BA3FB9c696',
        validated: [
            '0x1dF6...: Wed Apr...'
        ]
    }
}
```

Výstup D.3: Příprava prostředí, `C1`

```
Failed to read signal storage at storage2.json, creating a new one...
> signal register
0x557dedb8505a767dc02bd89c15f00a62f9aff66f8af59dc9db47c9c6e38c0e6f
0x680f2016e57bcbcc3ae928622be2b626c47e0937edc9eb61b5540da1b508d7b3
0xddebb6cd9e1401ece355ba527fa9cf8e0d91368e0d30b0d04970aeaa8e17996b
> identity verify ws://localhost:8080 test2@x.y mail
> identity claim test2@x.y f8797169175139e31f5f6682b1f4ef07
0x8f2d0cdbdc427417f6fe69ec44d0021d5facfc3f9cb0725bffde888527391aee
> identity whois test2@x.y
{
    Ethereum: '0xFFcf8FDEE72ac11b5c542428B35EEF5769C409f0',
    Identity: {
        user: '0x0d370B0974454D7b0E0E3b4512c0735A6489A71A',
        validated: [
            '0x1dF6...: Wed Apr...'
        ]
    }
}
```

Výstup D.4: Příprava prostředí, `C2`

```
0x6b4d19ffa90fa4814dadabcc80db9c873f90cc91b41abf09a3161b8b341b9f49
Request test1@x.y, mail resolved into token:
0a1cef6a46f37a3981d91a08b7061d88
0x43b0a7ed74b6041a2cb8577c6095f525015cd977b0766e89147355328e9d58fb
Request test2@x.y, mail resolved into token:
f8797169175139e31f5f6682b1f4ef07
```

Výstup D.5: Příprava prostředí, server

```
> signal setdev test1@x.y
> signal clear
> signal send test2@x.y plain.txt cipher.txt
0x37a1a1a84b2f227afe48901fcdd51e9c254b76e94a4cdefd9f7aa96682a8a493
{
    type: 3,
    body: 'MwgA...',
    registrationId: 2128706230656927700
}
```

Výstup D.6: Navázání relace, `C1`

```
> signal setdev test2@x.y
> signal receive test1@x.y cipher.txt
ahoj
```

Výstup D.7: Navázání relace, `C2`

```
> signal setdev test1@x.y
> signal clear
> signal send test2@x.y plain.txt cipher.txt
0xe8f65974396e11cae50c2b46399dab517c6fa9fa02fe23a44d7031595a34df94
{
    type: 3,
    body: 'MwgB...',
    registrationId: 2128706230656927700
}
> signal receive test2@x.y cipher3.txt
3

> signal receive test2@x.y cipher2.txt
2

> signal receive test2@x.y cipher1.txt
1
```

Výstup D.8: Asynchronní komunikace, `C1`

```
> signal setdev test2@x.y
> signal clear
> signal receive test1@x.y cipher.txt
ahoj

> signal send test1@x.y plain1.txt cipher1.txt
{
    type: 1,
    body: 'Mwoh...',
    registrationId: 93
}
> signal send test1@x.y plain2.txt cipher2.txt
{
    type: 1,
    body: 'Mwoh...',
    registrationId: 93
}
> signal send test1@x.y plain3.txt cipher3.txt
{
    type: 1,
    body: 'Mwoh...',
    registrationId: 93
}
```

Výstup D.9: Asynchronní komunikace, `C2`

```
> signal status
Storage cid:    QmQDWo2nfVHJJW6LZh2ptAT2NNCociiLpaZg62QtUyhs2W
Used keys:      2 out of 10
Remaining keys: 8
> signal update opk 50
0x3f4979106824b4b9b06551cad98efc503488e1d0b8628ef1f7a9150c1c069085
> signal status
Storage cid:    QmXwWjMt9GtkX1syY6M2kJ4eeR7FGndZsFScbH4QFQkQfk
Used keys:      0 out of 50
Remaining keys: 50
```

Výstup D.10: Obnova klíčů, `C2`

```
> senate init invite senator 0xFFcf8FDEE72ac11b5c542428B35EEF5769C409f0
0x60873ee5850dac8b38e250cf82fa2f4049ba49791c1a5c7f1af2719b1cd75b54
2
> senate vote 2 accept
0x91ab705474028339aea0f69d3bdea6826599d6c998b87e55ff31095f7f546f26
> senate print
Result {
    subject: '0xFFcf8FDEE72ac11b5c542428B35EEF5769C409f0',
    deadline: 2021-04-30T10:17:12.000Z,
    votedFor: '1',
    votedAgainst: '0',
    id: '2',
    description: '',
    topic: 'Senator invitation',
    state: 'Resolved'
}
```

Výstup D.11: Přizvání senátora, `C1`

```
> senate init invite senator 0x28a8746e75304c0780E011BEd21C72cD78cd535E
0x67648abb0fa5bb944d7c763458ed017b5a1f70bd750315b3142a1286baaf270c
3
> senate print
    Result {
    subject: '0x28a8746e75304c0780E011BEd21C72cD78cd535E',
    deadline: 2021-04-30T10:18:24.000Z,
    votedFor: '0',
    votedAgainst: '0',
    id: '3',
    description: '',
    topic: 'Senator invitation',
    state: 'Pending'
}
```

Výstup D.12: Přizvání senátora, `C2`

```
> identity verify ws://localhost:8080 pepa@novak.cz mail
> identity reject pepa@novak.cz 29ee975423364388f9eeb51acc932e19
0x1c3cdd22b00da5f4a6c64d434120cf6b5545322d894eae41f3e368c69c850685
> identity whois pepa@novak.cz
User not found
```

Výstup D.13: Odmítnutí identity, `C1`

```
0xf5cfc9cba31f8f4920a25d5b3abc19394ea71dd4d4fa425e3dc282b52b704c01
Request pepa@novak.cz, mail resolved into token:
29ee975423364388f9eeb51acc932e19
```

Výstup D.14: Odmítnutí identity, server

```
> identity verify ws://localhost:8080 franta@novak.cz mail
> identity claim franta@novak.cz e00ddf49320611f51787b858664cbc8e
0x920d699012469bb186dfff6af15f74dc4606e514c3f7dd0f61f707a0742b4b8a
> identity whois franta@novak.cz
{
    Ethereum: '0x90F8bf6A479f320ead074411a4B0e7944Ea8c9C1',
    Identity: {
        user: '0x79183957Be84C0F4dA451E534d5bA5BA3FB9c696',
        validated: [
            '0x1dF6...: Thu Apr...'
        ]
    }
}
> identity disown franta@novak.cz
0x9d9d23c13d434e6cf1debba949027d0bb1d200059bd5e70b3de4598d2a8e93ed
> identity whois franta@novak.cz
User not found
```

Výstup D.15: Zřeknutí se identity, `C1`

```
0xeff300499f0e36038ac1bc45ecefdbad1dcbcaeec9a09f5e9c0811e8f035ab4d
Request franta@novak.cz, mail resolved into token:
e00ddf49320611f51787b858664cbc8e
```

Výstup D.16: Zřeknutí se identity, server