VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

ORCHESTRACE MODULŮ MULTITENANTNÍCH SYSTÉMŮ

MODULE ORCHESTRATION OF MULTITENANT SYSTEMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. PETR FREYBURG**
AUTHOR

VEDOUCÍ PRÁCE **Ing. ALEŠ SMRČKA, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

142729

Ústav:	Ústav inteligentních systémů (UITS)
Student:	**Freyburg Petr, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Informační systémy a databáze
Název:	**Orchestrace modulů multitenantních systémů**
Kategorie:	Webové aplikace
Akademický rok:	2022/23

Zadání:

1. Nastudujte multitenantní systémy. Nastudujte možnosti použití kontejnerů při orchestraci informačních systémů.
2. Analyzujte požadavky na tvorbu multitenantních systémů z jednotlivých instancí informačních systémů. Navrhněte architekturu pro orchestraci informačního systému s vyčleněným sdíleným aplikačním modulem za účelem multitenance daného systému.
3. Implementujte navržené řešení pomocí systému Kubernetes.
4. Demonstrujte navržené orchestrační řešení.

Literatura:

- IEEE standard 1471-2000. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. doi: https://doi.org/10.1109/IEEESTD.2000.91944
- Oracle. "Introduction to the Multitenant Architecture." Dostupné na URL: https://docs.oracle.com/en/database/oracle/oracle-database/19/multi/introduction-to-the-multitenant-architecture.html

Při obhajobě semestrální části projektu je požadováno:
První dva body zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu:	Hanáček Petr, doc. Dr. Ing.
Datum zadání:	1.11.2022
Termín pro odevzdání:	17.5.2023
Datum schválení:	3.11.2022

Abstrakt

Tato práce se zabývá tvorbou multitenantních systémů a jejich orchestrací. Proces tvorby je založen na transformaci existujícího monolitického, avšak modulárního systému s cílem vyčlenění vybraného modulu. Vzniklé řešení zahrnuje infrastrukturu, která umožňuje zabezpečený přenos mezi informačním systémem a vyčleněným modulem. Tato infrastruktura izoluje jednotlivé tenanty do sebe. Jednotlivé moduly jsou kontejnerizovány v technologii Docker a jsou orchestrovány pomocí nástroje Kubernetes. Navržené řešení podporuje několik rozhraní mezi modulem a systémem. Podporovaná rozhraní zahrnují například standardní klient-server architekturu nebo standardní vstupně-výstupní umožňující jednorázové spouštění konzolových aplikací.

Abstract

This thesis deals with the creation of multitenant systems and their orchestration. The creation process is based on the transformation of an existing monolithic but modular system in order to extract a selected module. The resulting solution includes an infrastructure that enables secure transmission between the information system and the extracted module. This infrastructure isolates the individual tenants from each other. The individual modules are containerized in Docker technology and orchestrated using Kubernetes. The proposed solution supports several interfaces between the module and the system. Supported interfaces include, for example, a standard client-server architecture or a standard input/output to allow the single-running of console applications.

Klíčová slova

multitenantní systém, kontejnerizace, orchestrace, zabezpečený přenos, Docker, Kubernetes, cloud, proxy

Keywords

multitenant system, containerization, orchestration, secure data transport, Docker, Kubernetes, cloud, proxy

Citace

FREYBURG, Petr. *Orchestrace modulů multitenantních systémů*. Brno, 2023. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Orchestrace modulů multitenantních systémů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Petr Freyburg
16. května 2023

Poděkování

Děkuji vedoucímu práce panu Ing. Aleši Smrčkovi, Ph.D. za cenné rady a ochotu při vypracování této diplomové práce.

Obsah

Kapitola 1

Úvod

Cílem této diplomové práce je navrhnout a implementovat architekturu na orchestraci modulů multitenantních systémů pomocí nástroje Kubernetes. Multitenantní systémy jsou v dnešní době velmi populární přístup. Takový systém je používán více uživateli (tenanty), kteří mají iluzi, že jsou jedinými uživateli tohoto systému. Data jednotlivých tenantů jsou od sebe izolována. Tenanti typicky komunikují se systémem prostřednictvím sítě internet.

Tato diplomová práce navazuje na diplomovou práci Filipa Jeřábka [8], která navrhla proces transformace monolitických systémů na multitenantní systémy. Při přesunu části informačního systému do multitenantního prostředí je jako první třeba vymezit část systému, která bude oddělena a přesunuta. Dále je třeba tuto část zabalit do kontejneru a nasadit ji do zmíněného mulitenantního řešení.

Práce si klade za cíl vytvořit infrastrukturu, která umožní zabezpečené spojení mezi informačním systémem a vyčleněným modulem. Vzniklé řešení sestává z programů `client` a `server`. Program `client` je umístěn na straně informačního systému a přes zabezpečené SSL připojení přeposílá požadavky programu `server`, který je umístěn na straně multitenantního systému. Program `server` má na starosti zpřístupnění odpovídajího modulu danému tenantovi.

V rámci řešení dodaného touto prací lze v rámci multitenantního systému provozovat kontejnerizované moduly, které očekávají požadavky na některém vlastním portu prostřednictvím TCP připojení. Dále lze v rámci výsledné infrastruktury provozovat konzolové aplikace, které pracují na základě vstupu v argumentech příkazové řádky nebo které pracují se vstupem, který obdrží v rámci vlastního standardního vstupu.

V kapitole 2 je diskutován současný stav v oblasti Cloud computingu a multitenantních systémů. Dále jsou v rámci této kapitoly uvedeny technologické výzvy, které je třeba u tvorby takového multitenantního systému uvažovat. V kapitole 3 se hovoří o návrhu takového systému počínaje vyčleněním modulu, analýzou požadavků na tento systém a poté návrhem samotné infrasktruktury včetně návrhu jednotlivých komponent. Na závěr se v této kapitole podíváme i na účtování v takovém systému. V kapitole 4 se práce zaměřuje na implementační detaily vzniklé multitenantní infrastruktury. V kapitole 5 je popsáno, jak bylo výsledné řešení otestováno. Konečně v kapitole 6 si představíme některé příklady použití výsledné infrastruktury.

Kapitola 2

Současný stav

V rámci této kapitoly budeme diskutovat současný stav multitenantních systémů. V první části 2.1 se pobavíme o cloud computingu obecně. V další části 2.2 se zaměříme na samotné multitenantní systémy – definujeme si tento pojem a také se seznámíme s jejich dělením. Ve třetí části 2.3 se budeme zabývat technologickými aspekty, které je třeba při vývoji multitenantních systémů vzít v potaz.

2.1 Cloud Computing

Název **Cloud Computing** je odvozen od internetu. Obecně se hovoří, že se jedná o nějaký software, který je dostupný prostřednictvím sítě internet. V podstatě cloud computing je konstrukce, která umožňuje přístup uživatele k aplikaci, která se nachazí v jiné geografické lokalitě než je jeho zařízení. Nejčastěji se tato aplikace bude nacházet v *data centru* – kolekci serverů, kde je aplikace hostována. Tato celá podkapitola (včetně podsekcí) byla převzata z *Cloud computing: a practical approach* [19].

2.1.1 Služby

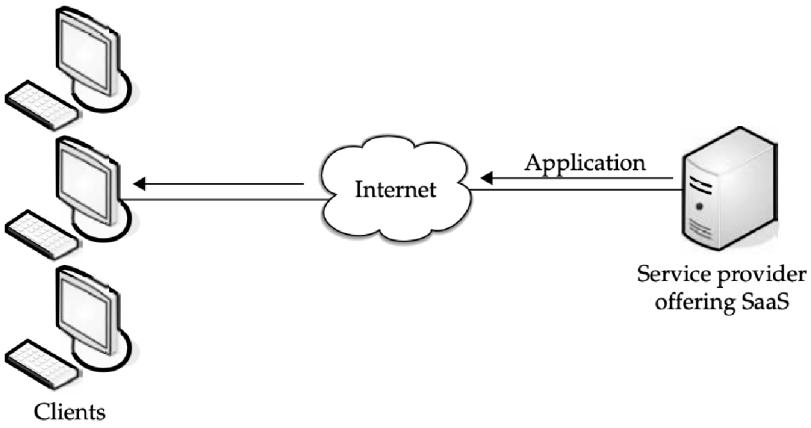
Označení služby (anglicky *services*) je v cloud computing koncept užívání znovupoužitelných komponent v rámci sítě poskytovatele cloudu. Klíčové vlastnosti služeb jsou:

- vysoká škálovatelnost,
- *multitenance*,
- nezávislost na zařízení,
- dostupnost (ve smyslu jednoduchého zapojení se do využívání těchto služeb).

Dále se seznámíme s různými typy služeb.

Software as Service

Software as Service (zkráceně SaaS) je model, kdy je aplikace hostována jako služba, která je dostupná uživatelům prostřednictvím internetu. Uživatel nemusí řešit údržbu této služby, na druhou stranu nemůže ovlivnit jakékoliv změny ve službě, o kterých rozhodne její provozovatel. Typicky se jedná o webovou aplikaci.



Obrázek 2.1: Software as Service [19].

Na obrázku 2.1 je ukázka, jak může vypadat služba poskytovaná jako Software as Service. Na pravé straně vidíme server, který poskytuje aplikaci klientům prostřednictvím sítě internet.

Platform as Service

V případě Platform as Servide (zkráceně PaaS) hovoříme o případu cloud computingu, který poskytuje veškeré zdroje potřebné k sestavení aplikací a služeb kompletně prostřednictvím sítě internet a to bez nutnosti stahovat či instalovat jakýkoliv software.

Hardware as Service

Hardware as a Service (HaaS) neposkytuje softwarová řešení jako takové, ale hardwarové prostředky. V rámci těchto služeb lze využívat např. úložiště, procesorového času atp. pro řešení vlastních úloh.

2.2 Multitenantní systémy

Multitenantní systém je architektura, ve které jedna instance software obsluhuje vícero uživatelů, kterým říkáme *tenanti*, ti jsou od sebe logicky odděleni [3]. Tenant je považován za majitele dané aplikace, který je zodpovědný za managment a údržbu aplikace, na druhou stranu se k aplikaci poskytované tenantem mohou připojovat běžní uživatelé [9]. Důležitou vlastností těchto systémů je, že tenanti sdílí výpočetní a jiné prostředky, ale jejich data a konfigurace jsou od sebe *izolovány*. Multitenanci lze uplatnit ve všech přístupech cloud computingu (SaaS, PaaS i HaaS), ale nejčastěji se však hovoří o multitenantních SaaS aplikacích [9].

2.2.1 Dělení multitenantních systémů dle nasazení

Existují čtyři přístupy [9] nasazení multitenantních systémů. Tyto čtyři přístupy se liší tím, které prostředky a jak budou sdíleny. Tyto přístupy si shrneme níže:

- Úplně izolovaný aplikační server – zde každý tenant užívá pro sebe vyhrazený server. Tato možnost není příliš efektivní, jelikož tenanti nesdílí žádné prostředky. Sice tenant dostane velmi dobrý výkon, ovšem toto řešení je velice drahé, ale správa tohoto řešení je nejjednodušší.
- Virtualizovaný aplikační server – v tomto případě každý tenant přistupuje k vyhrazené aplikaci, která běží na jí vyhrazeném virtuálním stroji. Toto řešení sice umožní obsluhu více než jednoho tenanta v rámci serveru, ale stále zde nedochází ke sdílení většiny prostředků. Oproti předchozí možnosti se snižuje cena, ale obvykle i výkon.
- Sdílený virtuální server – při tomto použití každý tenant přistupuje k aplikaci, která s vícero aplikacemi sdílí virtuální stroj. Dochází k většímu sdílení prostředků, k nárůstu kolizí mezi tenanty, kteří se snaží využít stejný zdroj. Dochází ovšem k nárůstu efektivity a poklesu ceny řešení.
- Sdílený aplikační server – zde tenanti sdílejí aplikační server a přistupují ke zdrojům aplikace prostřednictvím separátních sezení nebo vláken. Prostředky jsou sdíleny tak moc, jak je jen možné. Cena tohoto řešení je z uvedených přístupů nejlevnější, ale z pohledu implementace je toto řešení nejkomplikovanější, protože je nutné implementovat izolaci tenantů mezi sebou.

2.2.2 Typy databází multitenantních systémů

Při tvorbě multitetnatního systému je třeba zvážit, jak budou uložena data jednotlivých tenantů. Dále si představme tři přístupy [17].

Sdílená databáze

Jak název napovídá, v tomto multitenantním systému bude použita jedna databáze pro všechny tenanty. Jako příklad takové tabulky uvažme tabulku 2.1. Vidíme, že každý řádek má své ID (primární klíč řádku), také má své sloupce c_1 až c_n, na záver řádek obsahuje sloupec `Tenant_ID`, který značí tenanta, který je vlastníkem daného záznamu. Při hledání určitého záznamu (popř. určitých záznamů) pro nějakého tenanta T_X bude třeba při dotazování přidat do dotazu i klauzuli: `Tenant_ID` $= T_X$. Tato klauzule provede izolaci tenantů mezi sebou.

Nyní si shrňme výhody a nevýhody tohoto přístupu:

- Výhodou tohoto přístupu je zejména jeho jednoduchost – udržuje pouze jednu databázi a oproti ostatním přístup je snadné i přidat nového tenanta. Též přístup k datům je jednoduchý.
- Tento přístup má však i řadu nevýhod. Zásadní nevýhodou jsou bezpečnostní rizika, kdy se může stát, že tenantovi zpřístupníme data jiného tenanta (ať už jen pro čtení či dokonce i pro modifikaci). Jsou zde problémy s přístupností i výkonem – jeden tenant může vytížit databázi svým dotazem tak, že tím omezí přístup ostatním tenantům.

Jedno databázové schéma na tenanta

Pokud aplikujeme tento přístup, tak v rámci jedné databáze pro každého tenanta vytvoříme jedno databázové schéma. To zajistí jistou izolaci mezi tenanty. V tabulce 2.2 vidíme příklad

ID	c_1	c_2	...	c_n	Tenant_ID
1	$V1.1_{T1}$	$V2.1_{T1}$	...	$VN.1_{T1}$	T_1
2	$V1.2_{T1}$	$V2.2_{T1}$	...	$VN.2_{T1}$	T_1
3	$V1.3_{T2}$	$V2.3_{T2}$	...	$VN.3_{T2}$	T_2
4	$V1.4_{T2}$	$V2.4_{T2}$	...	$VN.4_{T2}$	T_2
...	...	...	...	...	...

Tabulka 2.1: Příklad databázové tabulky ve sdílené databázi.

ID	c_1	c_2	...	c_n
1	$V1.1_{T1}$	$V2.1_{T1}$	...	$VN.1_{T1}$
2	$V1.2_{T1}$	$V2.2_{T1}$	...	$VN.2_{T1}$
...	...	...	...	...

Tabulka 2.2: Příklad databázové tabulky ve schématu pro tenanta T1 .

databázové tabulky při použití tohoto přístupu. Na rozdíl od 2.1 vidíme, že tabulka obsahuje jen data pro právě prvního tenanta a žádné jiné (respektive neobsahuje data nikoho jiného). Není tedy třeba přidávat k dotazům žádnou klauzuli pro rozlišení tenantů, ale je třeba zvolit správné schéma pro příslušného tenanta.

Nyní si opět shrňme výhody a nevýhody tohoto přístupu:

- Tento přístup přináší následující výhody: data tenantů jsou od sebe lépe izolována než v případě předchozí přístupu, ale stále se jedná o stejnou databázi čili jistá rizika zde pořád jsou. Není zde již potřeba používat zvláštní klauzuli v dotazech pro rozlišení dat tenantů. Další výhodou je, že se nám zmenšují tabulky, protože každý tenant má vlastní.
- Nevýhodou tohoto přístupu je možnost použití špatného schématu a tím zpřístupnění tenantovi těch dat, které mu nepatří. Pořád je zde riziko přetížení databáze jiným tenantem. Stále zde nejsou možnosti pořádné škálovatelnosti.

Jedna databáze pro každého tenanta

Jak již název napovídá, v tomto případě každému tenantovi vytvoříme jeho vlastní, pouze jemu vyhrazenou databázi. Takže přidání nového tenanta znamená též i vytvoření nové databáze právě pro něj. Tabulka v databázi bude vypadat obdobně jako tabulka 2.2, akorát bude plně oddělená od ostatních tenantů, jelikož je izolována právě ve vlastní databázi daného tenanta. Samozřejmě zde opět není třeba přidávat do tabulek žádný identifikátor tenanta ani používat žádné zvláštní klauzule pro odlišení tenanta, jelikož prostor pro tenanta je úplně izolovaný od ostatních tenantů.

I tento přístup s sebou nese své výhody i nevýhody:

- Výhodou je ona izolovanost od ostatních tenantů, data jsou pro ostatní neviditelná. Tento přístup přináší i snadnou správu a přístupnost dat daného tenanta. Na rozdíl od prvního přístupu není třeba nijak upravovat dotazy na databází. Je minimalizován problém se znepřístupněním databáze činností jiného tenanta.
- Nevýhodou je rostoucí výpočetní náročnost na uložení a provoz vícero databází – rostoucí nároky na technické vybavení. Tento přístup přináší i redundanci dat, jelikož

různé konfigurace, co mohou být pro každého tenanta stejné, musíme uložit právě pro každého tenanta zvlášť. S tím přichází i složitější přidávání nového tenanta, kdy je třeba vytvořit a zprovoznit novou databázi. Může být složitější řízení přístupu tenantů k jejich databázi.

2.2.3 Typy multitenantních systémů dle přístupu k aplikaci

Dalším možným rozdělením multitenantních systémů je dělení podle počtu instancí aplikací a instancí databází v tomto systému. Existují tři typy při dělení podle tohoto pohledu [3].

Jedna aplikace, jedna databáze

První přístup je, kdy v multitenantním systému běží jedna aplikace a k ní jedna databáze – vše je sdíleno všemi tenanty. V případě databáze je třeba řešit izolaci dat jednotlivých tenantů, což lze řešit pomocí přístupů popsaných v podkapitole 2.2.2. Aplikace na tyto přístupy též musí navázat, aby tenantům prezentovala právě jejich data a žádná cizí.

Jedna aplikace, vícero databází

Tento přístup pracuje s jednou aplikací, která se napojuje na vícero databází – každou vyhrazenou pro tenanta. Data tenantů jsou od sebe izolována v jejich samostatných databázích. Na úrovni aplikace je třeba ošetřit zvolení správné databáze pro daného tenanta.

Vícero aplikací, vícero databází

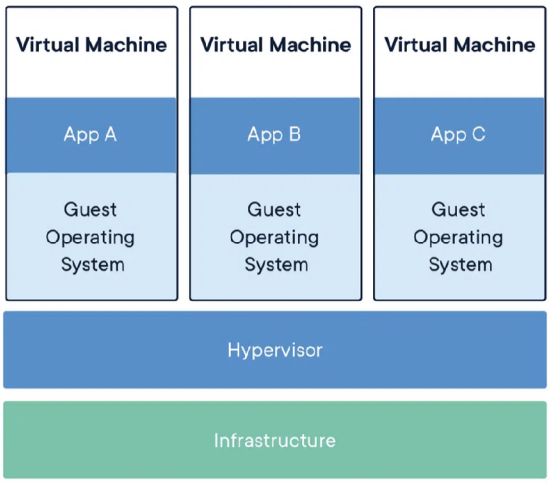
Poslední přístup pro každého tenanta vytváří jeho vlastní instanci aplikace a k ní i zvláštní databázi. Je třeba jen zabezpečit připojení tenanta na jeho instanci a žádnou jinou. Tento přístup je nejnáročnější, ale přináší též nejlepší zabezpečení ve smyslu oddělení jednotlivých tenantů od sebe.

2.3 Technologické aspekty tvorby multitenantního systému

Tvorba multitenantních systémů s sebou přináší několika výzev. Je třeba zabezpečit provoz vícero aplikací v rámci jednoho počítače. Dále je třeba zajistit bezpečný přenos dat po síti mezi multitenantním systémem a jeho uživatelem. Též je třeba se podívat na stávající informační systémy, jejich strukturu a diskutovat možnosti jejich multitenance. V závěru této podkapitoly se ještě podíváme na možnosti serializace dat, která je třeba jak pro přenos dat, tak i kvůli uložení konfigurací a na programovací jazyky, ve kterých můžeme naše výsledné řešení implementovat.

2.3.1 Nasazení software

V podkapitole 2.2 byla několikrát zmíněna možnost provozu více služeb v rámci jednoho počítače. K dosažení tohoto cíle je třeba aplikace buď virtualizovat nebo kontejnerizovat. Budeme také diskutovat tzv. tradiční nasazení, což byla dřívější možnost nasazení aplikací bez použití virtualizace nebo kontejnerizace.



Obrázek 2.2: Virtualizace [5].

Tradiční nasazení

Nejstarší tradiční možností nasazení je nasazování aplikací do operačního systému bez jakéhokoliv obalení (tj. bez virtualizace či kontejnerizace). Do operačního systému se nainstaluje celá aplikace, její závislosti atd. a spustí se. Tento přístup je sice jednoduchý, ale přináší s sebou různé problémy. Jedním problémem je izolace software od sebe, vznikají problémy s alokací prostředků a zejména nejsou určeny hranice kolik prostředků může, která aplikace využívat. Dalším problémem je, že aplikace nelze škálovat.

Převzato z *Traditional Deployment VS Virtualization VS Container* [1].

Virtualizace

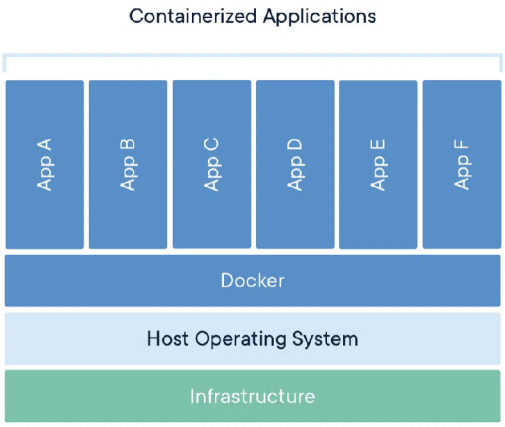
Virtualizace [7] umožňuje sdílení hardwarových prostředků počítače mezi více virtuálními počítači, zvanými *virtuální stroje* (anglicky *virtual machine*). Virtualizace pomocí software vytváří abstrakční vrstvu (tzv. `Hypervisor`, která umožňuje rozdělení hardwarových prostředků (paměť, procesor, atp.) mezi více virtuálních strojů. Každý tento virtuální stroj obsahuje svůj vlastní operační systém, svůj souborový systém a je izolován od ostatních virtuálních strojů běžících v rámci jednoho fyzického počítače.

Na obrázku 2.2 je znázorněno několik virtuálních strojů běžících nad jednou infrastrukturou. Mezi samotnými virtuálními stroji a onou infraskturou vidíme zmiňovanou vrstvu `Hypervisor`. Dále vidíme, že každý virtuální stroj obsahuje kromě své aplikace i svůj vlastní operační systém.

Kontejnerizace

Kontejner [5] je softwarová jednotka, která obsahuje zdrojový kód aplikace a jeho závislosti. Cílem kontejnerů je, aby software, který obsahuje fungoval všude stejně bez ohledu na infrasktrukturu a též se dal jednoduše spustit.

Na obrázku 2.3 je infrastruktura s několika kontejnery. Nad infraskturou běží operační systém, nad kterým dále běží vrstva provozující kontejnery (v tomto případě Docker). Konečně vidíme, že každý kontejner již obsahuje pouze svoji aplikaci bez operačního systému.



Obrázek 2.3: Kontejnerizace [5].

Když srovnáme obrázky 2.2 a 2.3 na první pohled je vidno, že kontejnery jsou úspornější, jelikož neobsahují svůj operační systém, ale využívají hostitelský operační systém, ke kterému přidávají závislosti aplikace (knihovny atp.), které jsou v kontejneru obsaženy.

Jednou z možností kontejnerizace je **Docker**[1]. Docker běží jako klient-server aplikace. Klientská část komunikuje se serverovou částí, zvanou `Docker daemon`, která zodpovídá za sestavování a provoz kontejnerů. Pro vytvoření kontejneru je nejdříve třeba vytvořit tzv. obraz, to se dělá pomocí předpisu instrukcí zvaných `Dockerfile`. Po sestavení obrazu je možné již rozjet samotný kontejner. Pro aplikace skládající se z více kontejnerů existuje nástroj `Docker Composer`.

Alternativou Dockeru je **Buildah**[2]. Rozdílem oproti Dockeru je, že zde neběží žádný démon. Obraz kontejneru lze vytvořit také pomocí `Dockerfile` nebo z existujícího obrazu, ale také lze vytvořit prázdný obraz a ten si dále přizpůsobit.

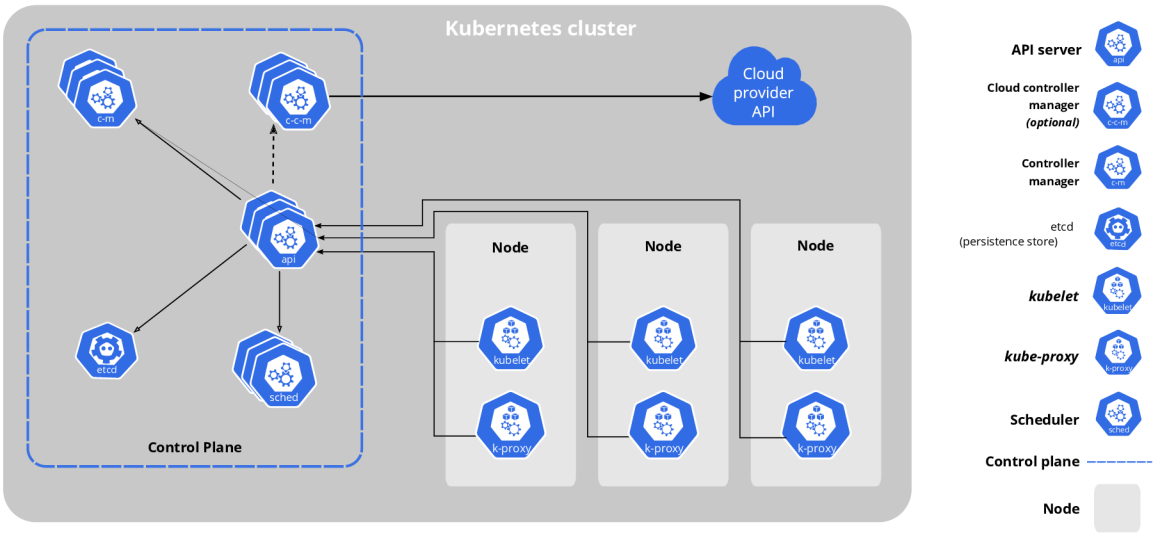
2.3.2 Orchestrace

Orchestrace kontejnerů [16] automatizuje nasazení, správu, škálování a propojení kontejnerů. Orchestrace může být použita v jakémkoliv prostředí, kde lze použít kontejnery. Další důležitou vlastností je, že orchestrace též zabezpečuje bezpečné propojení mezi kontejnery. Také zajistí alokaci zdrojů pro kontejnery a monitoruje stav kontejnerů za běhu. Klíčovou vlastností je i vyrovnávání zátěže mezi kontejnery.

Kubernetes

Kubernetes[3] (nebo též známý jako K8s) je open-source orchestrační nástroj. Při sestavení a spuštění kontejnerů získáme tzv. *cluster*, ten se skládá z pracovních strojů (anglicky *worker machines*), které obsahují kontejnery. Takový pracovní stroj je v clusteru alespoň jeden. Na obrázku 2.4 jsou znázorněny komponenty [12], které jsou v Kubernetes clusteru. V části nazvané `Control Plane` se nachází části, které jsou odpovědné za řízení clusteru a také za detekci a odezvu na události v clusteru. Část `Control Plane` se skládá z:

[1] `https://www.docker.com`
[2] `https://buildah.io`
[3] `https://kubernetes.io`



Obrázek 2.4: Schéma Kubernetes clusteru [12].

- `kube-apiserver` je komponenta, která odhaluje Kubernetes API. Je navržen tak, aby horizontálně škálovala, tak že nasazuje více jejích instancí, tj. vyrovnává se zátěž mezi těmito více instancemi.
- `etcd` je konzistentní a vysoce dostupný záložní sklad dvojic klíč-hodnota, který ukládá všechna data clusteru. Pokud je tento sklad používán, je třeba mít vytvořen plán zálohování.
- `kube-scheduler` je komponenta, která sleduje nově vytvořené *pody* (viz dále), které nemají přiřazený pracovní stroj (tj. uzel) a tento uzel jim přiřadí.
- `kube-controller-manager` – komponenta, ve které probíhají řídící procesy (např. ovladač uzlů, ovladač činností atp.).
- Poslední komponenta `cloud-controller-manager` obsahuje tzv. `cloud-specific` logiku. Tato komponenta umožňuje provázat cluster s API poskytovatele cloudu a separovat komponenty, které intereagují s cloud platformou od komponent, které intereagují pouze s clusterem. Opět je zde možnost horizontálního škálování. Tato komponenta se v cluster nachází jen tehdy, pokud cluster běží v cloudu, tj. např. ne lokálně na počítači.

Pracovní stroj se též označuje jako uzel (anglicky *node*). Jak bylo výše zmíněno obsahuje právě samotný kontejner nebo kontejnery, ty jsou uvnitř tzv. Podu. Pracovní stroj se skládá z těchto částí:

- `kubelet` je agent, který ověřuje, že kontejnery běží v rámci Podu, tj. ověřuje, že Pody běží a také, že běží správně dle specifikovaných vlastností v popisu PodSpecs.
- `kube-proxy` je síťová Proxy, která spravuje pravidla sítě. Tato komponenta umožňuje komunikaci s Pody ze vnitř i ze vnějšku Clusteru.

Kubernetes cluster se konfiguruje pomocí konfiguračního souboru ve formátu YAML. Deklarativním stylem se definuje, ze kterých kontejnerů se mají Pody skládat atp. Dále je např. možné nastavit tzv. replikaci Podů, která umožňuje horizontální škálování.

V rámci Kubernetes lze nakonfigurovat tyto tři typy služeb [11]:

- `ClusterIP` je služba, která je vystavena na portu na vnitřní IP adrese, to znamená, že služba je vystavena dovnitř clusteru.
- `NodePort` vystaví službu na každé IP adrese uzlu. Aby byla služba k dispozici nastaví jí IP adresu obdobně jako u `ClusterIP`.
- `LoadBalancer` je služba, která je vystavena ven z clusteru.

Docker Swarm

Možnou alternativou ke Kubernetes je Docker Swarm, což je orchestrační nástroj, který je součástí přímo Dockeru. Opět se skládá z uzlů, na kterých tentokrát běží Docker Engine a provozují kontejnery. Stejně jako Kubernetes podporuje rozkládání zátěže. Problémem je úzká provázanost s Docker API, což limituje jeho funkcionalitu oproti Kubernetes. Další nevýhodou jsou menší možnosti přizpůsobení a rozšiřování.

Převzato z *Kubernetes VS Docker Swarm – What is the Difference?* [6].

2.3.3 Zabezpečený přenos

Multitenantní systém je aplikace komunikující po síti, typicky prostřednictvím sítě internet, která je veřejně dostupná. Z toho důvodu je třeba zvažovat zabezpečení tohoto systému, tak aby byli jasně identifikovány obě strany komunikace (tj, aby každá z nich věděla s kým nyní komunikuje) a tuto komunikaci nemohl číst ani modifikovat nikdo třetí.

Protokol Secure Sockets Layer

Secure Socket Layer [18] (zkráceně SSL) je protokol sloužící k zabezpečení komunikace mezi dvěma počítači pomocí šifrování. Zašifrování jednotlivých zpráv probíhá pomocí certifikátů. Typické použití je při klient-server komunikaci. Certifikát lze také použít jako potvrzení identity.

SSL certifikát je vydáván *certifikační autoritou* (zkráceně CA). Pro vydání certifikátu je třeba na počítači žadatele vygenerovat `CSR` (zkratka *Certificate Signing Request*), obsahující informace o žadateli a jeho doméně, a dvojici *privátního* a *veřejného* klíče. K certifikační autoritě je třeba odeslat vygenerované CSR a veřejný klíč. CA provede různá ověření (např. identity, vlastnictví uvedené domény atp.) a v případě úspěšného ověření CA odešle žadateli jeho nový vygenerovaný certifikát.

Na úrovni souborů se certifikát skládá z několika souborů:

- Soubor `.pfx` obsahuje certifikát a jeho privátní klíč. Tento soubor se v žádném případě nepředává žádné další straně, drží jej pouze jeho vlastník. Soubor se předává dané aplikaci, která jej použije k vytvoření SSL spojení.
- Soubor `.key` obsahuje pouze privátní klíč.
- Soubor `.crt` je soubor s samotným certifikátem. Tento soubor lze předat druhé straně, která s námi bude komunikovat.

2.3.4 Struktura informačního systému

Informační systémy lze dělit různými způsoby. V této práci se zaměříme na dělení dle architektury. Toto dělení obsahuje dva typy – monolitický informační systém a mikroslužby (anglicky *microservices*) [4].

Monolitický informační systém

Monolitický informační systém je velký a komplexní software, který zastává více různých úkolů – např. zpracování HTTP požadavků, aktualizace a prezentace dat z databáze, různé výpočty atp. typicky z více oblastí, které daný systém pokrývá. Z pohledu operačního systému se jedná o jeden proces.

Výhodou tohoto přístupu je obecně jednoduchost v nasazení, testování i provozu. Problém nastává při zvětšování se informačního systému – zhrošuje se jeho udržovatelnost, dělají se hůře různé aktualizace zdrojového kódu. Vzhledem k velké vzájemné závislosti jednotlivých částí mezi sebou, může změna v jedné části vést k neočekávanému chování v části druhé.

Mikroslužby

Informační systém implementovaný jako mikroslužby je soubor služeb, kde každá tato služba má svoji vlastní odpovědnost. Každá mikroslužba je autonomní a může běžet i samostatně. Mikroslužba obsahuje všechny své závislosti – typicky běží v kontejneru nebo ve virtuálním stroji.

Mikroslužba obsahuje API, pomocí kterého s mikroslužbou komunikuje zbytek systému. Ostatní logika je schována za tímto API. Mikroslužby typicky komunikují přes protokol HTTP, ale také mohou využívat i jiné protokoly.

Zásadní výhodou mikroslužeb je dekompozice velkého monolitu do menších celků, kde každý má svoji odpovědnost. Tím se usnadní např. údržba. V případě aktualizací systému, se aktualizuje pouze daná mikroslužba, zbytek systému nebývá třeba modifikovat (pokud se nezmění komunikační protokol dané mikroslužby).

2.3.5 Programovací jazyky

V rámci multitenantního systému mohou běžet různé aplikace, které mohou být implementovány v různých programovacích jazycích. Infrastruktura, která umožní běh multitenantního systému a její součásti musí být implementována v některém programovacím jazyce. V této části si představíme dva programovací jazyky – C# a Python.

Jazyk C#

Jazyk C# [14] je objektově-orientovaný programovací jazyk, který umožňuje vytvářet bezpečné a robustní aplikace. Tento programovací jazyk pochází z rodiny programovacích jazyků C (spolu s jazykem C či C++).

Ve výpisu 2.1 je jednoduchý program napsaný v jazyce C#, který na obrazovku vypíše zprávu „Hello, World!“. Program se skládá z jedné třídy `Hello`, která obsahuje metodu `Main`.

```
using System;

class Hello
{
   static void Main()
   {
      Console.WriteLine("Hello, World");
   }
}
```

Výpis 2.1: Příklad jednoduchého programu v jazyce C# [14].

Programy napsané v jazyce C# běží na virtuálním systém spouštění zvaném CLR (zkráceno *Common Language Runtime*) a sadě knihoven. Zdrojový kód v jazyce C# je zkompilován do jazyka střední úrovně (anglicky *intermediate language*). Tento kód je uložen typicky v souborech s příponou `.dll`. Při samotném spouštění jsou tyto soubory načteny do CLR, který provede kompilaci do nativních instrukcí daného systému. CLR při provádění těchto instrukcí poskytuje služby jako je automatický `garbage collector`, zachytávání výjimek atp.

Jazyk Python

Jazyk Python [15] je interpretovaný objektově-orientovaný programovací jazyk. Jedná se o *vysoce úrovňový jazyk*, který se například oproti jazyku C vyznačuje svou jednoduchostí, umožňuje psaní jednoduše čitelných a kompaktních programů.

```
x = int(input("Insert number: "))

if x > 0:
  print("x is greater than zero.")
else:
  print("is less than or equal to zero.")
```

Výpis 2.2: Příklad jednoduchého programu v jazyce Python.

Ve výpisu 2.2 je příklad jednoduchého programu v jazyce Python. Uživatel je nejdříve vyzván k zadání čísla. Dále je o tomto čísle vypsáno, jestli je větší než nula nebo menší nebo rovno nule.

Jak bylo zmíněno v prvním odstavci, Python je interpretovaný programovací jazyk. Interprety jazyka Python byly napsány v různých programovacích jazycích. Jako příklad zmiňme interpret `cpython` napsaný v jazyce C. Tento interpret čte zdrojový kód v jazyce Python, který převádí do mezikódu zvaného `bytecode`, který poté vykonává. Na rozdíl od jazyku C#, zde není třeba žádná kompilace, která by se prováděla před samotným spouštěním.

2.3.6 Serializace dat

Při tvorbě multitenantního systému je třeba brát v úvahu i serializaci dat. Jednak bude třeba serializovat některá dat pro přenos přes počítačovou síť, ale také bude třeba ukládat různé konfigurace tenantů a u nich by bylo vhodné, aby byli jak strojově, tak i lidsky čitelné.

Formát JSON

JSON (Javascript Object Notation) [10] je formát dat pro výměnu informací, který je založen na podmnožině programovacího jazyka JavaScript. JSON je založen na dvou strukturách:

- kolekce párů název/hodnota (nebo též klíč-hodnota) zvaná *objekt*,
- uspořádaný seznam hodnot.

```
{
  "name": "George Orwell",
  "books": ["Animal farm", "1984"]
}
```

Výpis 2.3: Příklad JSON dokumentu s jedním objektem.

Ve výpisu 2.3 vidíme příklad jednoduchého JSON dokumentu s jedním objektem. Tento objekt obsahuje dva klíče `name` a `books`. První klíč obsahuje hodnotu typu textový řetězec a druhý klíč obsahuje hodnotu typu seznam, která obsahuje dva textové řetězce.

Formát YAML

YAML (YAML Ain't Markup Language) [2] je serializační jazyk, který klade důraz na čitelnost lidmi a přenositelnost mezi různými programovacími jazyky.

YAML dokument se může skládat ze tří uzlů:

- **skalár** (anglicky *scalar*) – konkrétní hodnota, jako např. textový řetězec, číslo atp.,
- **sekvence** (anglicky *sequence*) – seznam hodnot,
- **mapování** (anglicky *mapping*) – neuspořádaná množina dvojic klíč-hodnota.

```
name: George Orwell
books:
- Animal Farm
- 1984
```

Výpis 2.4: Příklad YAML dokumentu s jedním objektem.

Ve výpisu 2.4 vidíme obdobný dokument jako ve výpisu 2.3. Opět zde máme dva klíče – `name` a `books`. Klíč `name` obsahuje textový řetězec a klíč `books` obsahuje seznam textových řetězců.

Kapitola 3

Návrh infrastruktury multitenantních systémů

V této kapitole se budeme zabývat návrhem samotné infrastruktury multitenantního systému. Jako první budeme diskutovat vyčlenění samotného modulu dle typu informačního systému. Dále se zaměříme na analýzu požadavků pro tento systém, tj. co by tento systém měl splňovat. Poté nás čeká návrh samotné architektury tohoto systému, kde budeme diskutovat možnosti bez použití orchestrace a s použitím orchestrace. Poté probereme hlouběji návrh jednotlivých komponent představené v rámci navržené architektury. Dále budeme diskutovat neméně důležitou část – konfiguraci tenantů, která zajistí správné přepínání jednotlivých tenantů. Na závěr této kapitoly navrhneme účtování v rámci navrhovaného systému, tak aby jej bylo možné analyzovat a dalšími způsoby využívat.

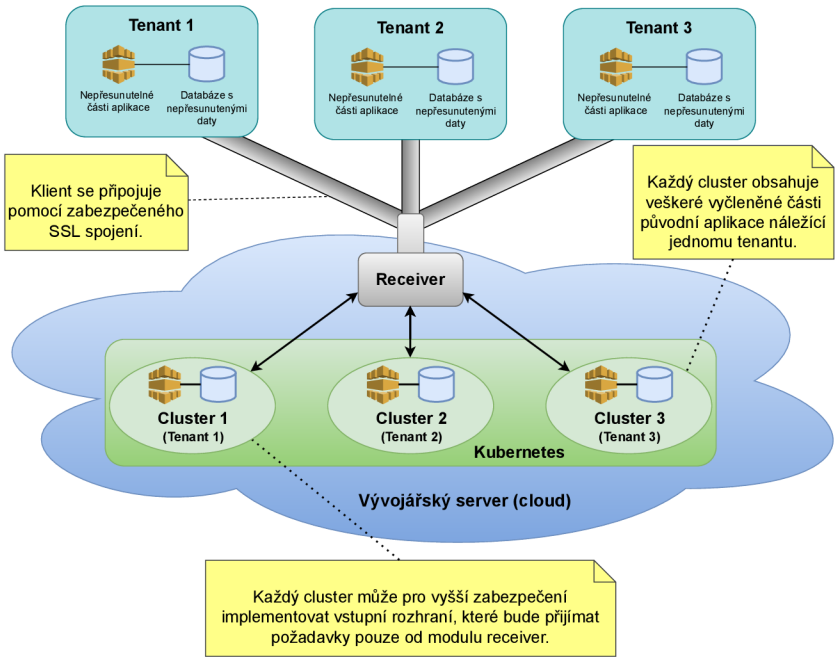
3.1 Vyčlenění modulu

Cílem této práce je vzít větší nebo menší část stávajícího informačního systému a tuto část nasadit do multitenantního prostředí [8]. Zbylá část, která zůstane běžet stávajícím způsobem, bude komunikovat s vyčleněnou částí, která již poběží v multitenantním systému. Zde jsou tři možnosti, které si nyní představíme.

Na obrázku 3.1 je znázorněna komunikace mezi vyčleněným modulem a informačním systémem. Ideální způsob takové komunikace je prostřednictvím nějakého otevřeného portu. Alternativně ale modul může být konzolovou aplikací a komunikace s ním bude probíhat přes standardní vstup a standardní výstup. Případně jako vstup mohou být použity argumenty konzole. Tato alternativní možnost je blíže diskutována v podkapitole 3.1.3. Eventuelně může informační systém komunikovat s více moduly, ne pouze s jedním, jak je uvedeno v našem obrázku.

3.1.1 Vnější část informačního systému

Vnější částí informačního systému je myšlena nějaká externí služba jako např. databáze (např. Redis, MySQL, atp.). Takové externí služby typicky komunikují s informačním systémem pomocí nějakého síťového protokolu. Tyto vnější části běžně fungují s informačním systémem v rámci jednoho počítače. Vyčlenění takové služby je velmi jednoduché, kdy stačí službu kontejnerizovat (pokud se tomu tak nestalo dříve) a umístit na server poskytující multitenantní řešení a tuto službu informačnímu systému zpřístupnit. V informačním systému je poté třeba pouze upravit navázání spojení s touto nově vyčleněnou službou.



Obrázek 3.1: Znázornění komunikace mezi informačním systémem a vyčleněným modulem. Na straně tenanta běží nevyčlenitelná část IS a na vývojářském serveru funguje jeho vyčleněná část, která je prostřednictvím programu `Receiver` zpřístupňována [8].

3.1.2 Informační systém jako mikroslužby

V případě informačního systému implementovaného jako mikroslužby je situace jednodušší než v monolitickém informačním systému. Zde typicky jednotlivé mikroslužby běží jako samostatné jednotky a komunikují mezi sebou např. pomocí protokolu HTTP. Další výhodou je, že mikroslužby typicky běží v kontejnerech. Zde stačí vybranou mikroslužbu, kterou chceme provozovat v multitenantním prostředí vzít a nasadit ji do tohoto prostředí. Poté je třeba pouze pozměnit komunikaci ve zbývajících částech informačního systému, které s touto mikroslužbou komunikovali tj. např. místo `localhost` napsat URL či IP adresu serveru, kde bude nově tato mikroslužba běžet.

3.1.3 Monolitický informační systém

Dalším případem je monolitický informační systém, zde je situace složitější. Zde je třeba oddělit určitou jeho část a vytvořit z ní samostatně pracující modul, který bude možné provozovat samostatně v multitenantním prostředí.

První nejjednodušší možností je vyčlenit určitou část zdrojového kódu informačního systému, přidat k ní HTTP komunikaci či obecně jinou síťovou komunikaci otevřenou na portu a touto cestou zpřístupnit komunikaci s tímto modulem.

Další možností je z tohoto vyčleněného zdrojového kódu vytvořit konzolovou aplikaci a síťovou komunikaci zajistí právě vytvořená infrasktuktura, která vloží této konzolové aplikaci na standardní vstup či jako argumenty této aplikace to, co dostane v těle požadavku a zpět poté pošle obsah standardního výstupu této konzolové aplikace.

```
class C:
  def __init__(self, i1, i2):
    # ...
  def m(self, p1, p2):
    # ...
    return y
```

Výpis 3.1: Příklad jednoduchého modulu, jehož rozhraní jsou jednotlivé metody a komunikace s ním probíhá právě přes parametry těchto funkcí.

Ve výpisu 3.1 vidíme příklad jednoduchého modulu v jazyce Python. Modul se skládá z jedné třídy `C` s konstruktorem a jednou metodou `m`. Rozhraním tohoto modulu je konstruktor a ona metoda `m`, komunikace probíhá přes parametry těchto funkcí. Výstupem tohoto modulu je návrat z metody `m`. Ve výpisu 3.2 je ukázka, jak tento modul bude vypadat po vyčlenění. Jako parametry se do konstruktoru a metody `m` předají parametry z příkazového řádku. Návrat z metody `m` se vypíše poté na standardní výstup.

```
from package import C
from sys import argv

c = C(argv[1], argv[2])
x = c.m(argv[3], argv[4])
print(x)
```

Výpis 3.2: Příklad vyčleněného modulu

Popsané vyčlenění bylo snadné, ovšem může problém může nastat při závislosti komponent. Pak může být vyčlenění velmi komplikované či dokonce nemožné. Jako závislost komponent si představme kooperaci více objektů různých tříd dohromady.

3.1.4 Informační systém pracující s konzolovými aplikacemi

Poslední možností je informační systém, který spouští konzolové aplikace a dále zpracovává (popř. ukládá) jejich výstupy. Obdobně jako v předchozí kapitole 3.1.3 takové aplikace mohou vstup obdržet prostřednictvím argumentů příkazové řádky nebo případně prostřednictvím standardního vstupu.

3.1.5 Typy modulů

Jak bylo nastíněno v předešlých podsekcích, budeme rozlišovat tři způsoby komunikace s moduly:

- prostřednictvím otevřeného portu, takový modul bude kontejnerizovaný a orchestrovaný pomocí Kubernetes,
- modul je konzolová aplikace, se kterou se komunikuje prostřednictvím argumentů příkazové řádky,
- module je konzolová aplikace, se kterou se komunikuje prostřednictvím standardního vstupu.

V případě konzolových aplikací nebudou moduly kontejnerizovány, ale poběží jich více v rámci jednoho kontejneru.

3.2 Analýza požadavků

Níže následuje soupis požadavků na výsledné řešení. V rámci tohoto řešení vzniknou dva programy – `secmux_client` a `secmux_server`. Tyto dva programy budou přes zabezpečenou komunikaci předávat data mezi informační systémem a vyčleněným modulem.

REQ1 Kontejnerizovaný modul je vyčleněn z původního IS.

REQ2 Klient je součástí nového IS (původní IS bez vyčleněných modulů).

REQ3 Server je součástí cloudového řešení.

REQ4 Klient a Server utvaří zabezpečenou obousměrnou komunikaci mezi IS a vyčleněným modulem.

REQ5 Server na základě portu a certifkátu zpřístupňuje daný vyčleněný modul a jeho část držiteli certifikátu.

REQ6 Klient má právě 1 certifikát.

REQ7 Server poskytuje účtovatelnost nad požadavky jednotlivých klientů.

REQ8 Vytvořený převáděcí nástroj umožní vyčlenění a vložení modulu do cloudu dle jasně popsaného postupu.

REQ9 Vytvořený převáděcí nástroj zajistí vygenerovaní certifikátů a nastavení přístupů k dané části modulu.

REQ10 Převáděcí nástroj je soubor skriptů a pokynů pro uživatele vykonávajícího převod.

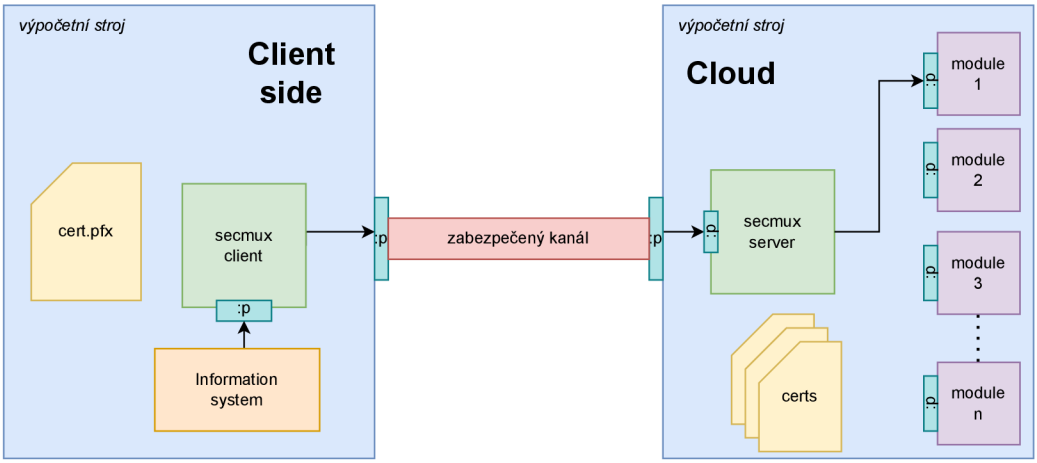
3.3 Návrh architektury

Z pohledu přístupu multitenantního systému k aplikaci (viz podkapitola 2.2.3) bude výsledný systém pracovat následovně: každý tenant dostane vlastní instanci aplikace, kterou potřebuje, tj. vlastní kontejner s danou aplikací. Z pohledu přístupu k databázi (viz podkapitola 2.2.2) každý tenant dostane i vlastní databázi, pokud ji bude potřebovat, tj. ve výsledném systému se nebude řešit sdílení jedné databáze vícero tenanty.

Výjimku tvoří konzolové aplikace, kdy může být sdílen kontejner více tenanty, ale při spuštění konzolového programu konkrétním tenantem se mu vytvoří jeho vlastní adresářový prostor, který se po odeslání výsledků ihned smaže.

Na obrázcích 3.2, 3.3 a 3.4 vidíme navrženou architekturu. Každý obrázek ukazuje jiný způsob komunikace mezi informačním systémem a vyčleněným modulem.

Nejdříve si popišme, co mají tyto tři obrázky společné. Na klientské straně se vždy nachází program `secmux_client` a na straně cloudu se nachází program `secmux_server`. Tyto dva programy mezi sebou navazují zabezpečenou komunikace přes SSL. Klientský program přijímá požadavky od informačního systému – buď přímo od něj nebo prostřednictvím prostředníka (viz dále). Tento program se autentizuje u serverového programu na základě svého certifikátu a tyto požadavky mu přeposílá. Program `secmux_server` na základě certifikátu a k němu přidružené autorizace zpřístupní příslušný modul, kterému tyto požadavky předá a veškeré odpovědi tohoto modulu poté pošle zpět klientské straně.



Obrázek 3.2: Komunikace informačního systému s vyčleněným modulem přes TCP prostřednictvím infrastruktury `secmux`. Značka `:p` značí nějaký otevřený port.

3.3.1 Komunikace s TCP modulem

Nejjednodušší možnost znázorňuje obrázek 3.2, kde při původní situaci informační systém a modul komunikovali prostřednictvím TCP připojení. Informační systém se přímo napojí na otevřený port programu `secmux_client` a ten veškeré požadavky přeposílá. Dále program `secmux_server` tyto požadavky přepošle na port příslušného modulu.

3.3.2 Komunikace s konzolovou aplikací, která očekává parametry příkazové řádky

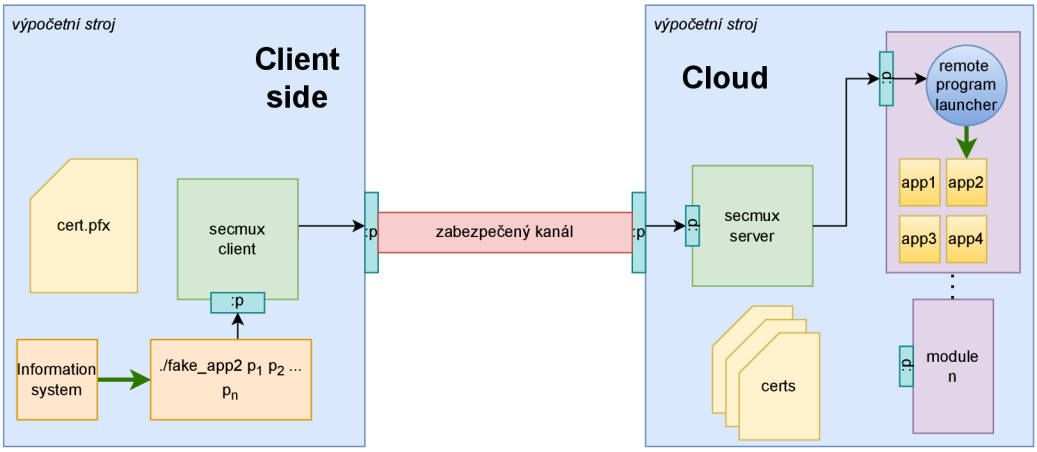
Další možnost je znázorněna na obrázku 3.3. Informační systém se nepřipojuje přímo na klientský program, ale pracuje s programem, který se tváří jako původní konzolový program, se kterým dříve tento informační systém pracoval. Tento program přijme parametry, popř. i soubory a přepošle je klientskému programu. Konzolový program zabalí veškeré parametry do JSONu (viz podkapitola 2.3.6) jako ve Výpisu 3.3. JSON objekt obsahuje dva klíče:

- `program` – název programu, který se má spustit,
- `args` – seznam argumentů, se kterými se má uvedený program spustit.

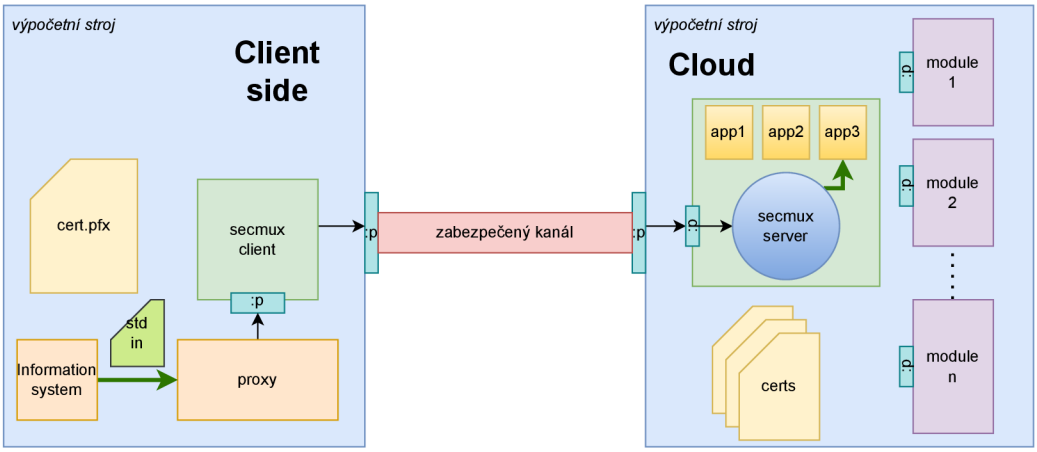
```
{
   "program": "app2",
   "args": ["p1", "p2", "p3"],
}
```

Výpis 3.3: Příklad zprávy spouštějící program `app2`.

Serverový program poté přepošle tuto zprávu přes TCP kontejneru `cli_worker`. Tento kontejner obsahuje REST API, které spustí uvedený program. Informační systém poté obdrží v odpovědi standardní výstup, standardní chybový výstup i návratový kód aplikace a popř. i vygenerované soubory.



Obrázek 3.3: Komunikace informačního systému s konzolovou aplikací přes její parametry spouštění prostřednictvím infrastruktury `secmux`. Značka `:p` značí nějaký otevřený port.



Obrázek 3.4: Komunikace informačního systému s konzolovou aplikací přes její standardní vstup prostřednictvím infrastruktury `secmux`. Značka `:p` značí nějaký otevřený port.

3.3.3 Komunikace s konzolovou aplikací, která čte standardní vstup

Poslední možností je Obrázek 3.4. V tomto případě Informační systém komunikuje s programem `proxy`, který čte svůj standardní vstup a veškerý jeho obsah přeposílá klientskému programu. Jakmile je vstup uzavřen odešle klientskému programu i symbol reprezentující EOF (`END OF FILE`). Serverový program na základě konfigurace tenantů spustí přímo ve svém kontejneru příslušný program, kterému předává veškerá data, která přijdou od klientské části na jeho standardní vstup. Jakmile obdrží symbol reprezentující konec vstupu, tak zavře standardní vstup příslušné aplikace. Poté přečte veškerý obsah standardního výstupu i standardního chybové výstupu a ty rozlišeně přepošle klientské straně.

Program `proxy` vytiskne vše co bylo na standardním výstupu aplikace na svůj standardní výstup a obdobně vytiskne i standardní chybový výstup na svůj standardní chybový výstup.

3.3.4 Správa kontejnerů

Existují dvě možnosti správy kontejnerů a výsledná infrastruktura obě tyto možnosti umožňuje. První možností je provoz kontejnerů bez jejich orchestrace, tedy kontejnery poběží na různých portech např. `localhost`. Druhou možností je orchestrování kontejnerů pomocí nástroje Kubernetes.

První případ je jednodušší, co se týče nastavení. Je třeba spustit kontejner a jeho port propagovat na `localhost` (či jiné doménové jméno). V konfiguraci ke každému certifkátu bude nejdůležitější právě onen port, na který bude program `server` přesměrovávat komunikaci. V této jednodušší implementaci lze nastavit `server` na správné přepínání komunikace, účtování atp. Tento přístup má jako výhodu pouze svou jednoduchost, ale je použitelný jenom pro pár kontejnerů. Tento přístup také neobsahuje žádnou detekci zda kontejnery korektně fungují.

Druhá možnost pracuje s orchestrováním kontejnerů. Kontejnery jsou v tomto případě orchestrovány pomocí nástroje Kubernetes. Pro každého tenanta bude existovat jeho vlastní instance jeho modulu (viz podkapitola 2.2.3). To je z důvodu, že Kubernetes umožní tyto kontejnery horizontálně škálovat, pro více vytížené kontejnery bude možné vytvářet jejich repliky pro rozložení zátěže. Další přínosem tohoto řešení je izolovanost jednotlivých tenantů, co se týče dat – nebude třeba nijak modifikovat schéma databáze či záznamy v databázi (viz podkapitola 2.2.2).

3.4 Návrh komponent

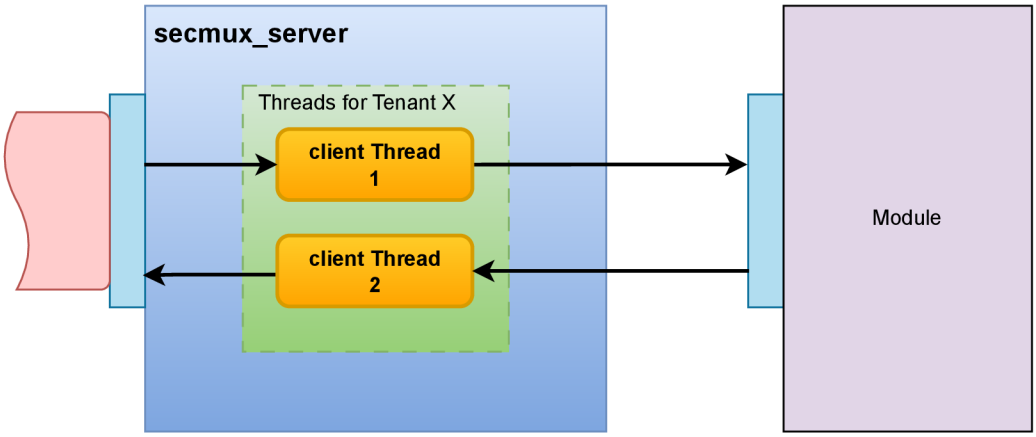
V předchozí podkapitole 3.3 jsme si představili navrhovanou architekturu a jednotlivé způsoby komunikace. V rámci této kapitoly si blíže popíšeme jednotlivé součásti (nebo též komponenty) architektury.

3.4.1 Serverový program

Program `secmux_server` musí přijímat zabezpečené SSL připojení od klientské strany. V rámci tohoto spojení je klientská strana identifikována certifikátem na jehož základě tento program zpřístupňuje daný modul. Pro každé připojení klientskou stranou vytváří program minimálně jedno nové vlákno, které bude dané připojení obsluhovat. Toto vlákno nazveme *klientské vlákno* (anglicky `client thread`). Serverový program stejně jako klientská strana pracuje s vlastním certifikátem, kterým se identifikuje v rámci SSl spojení.

Tento program musí také udržovat konfigurace jednotlivých tenantů a pravidelně si je obnovovat. Na základě těchto konfigurací bude právě zpřístupňovat určený modul danému tenantovi. O tomto dále viz podkapitola 3.5.

V případě modulu komunikujícího přes TCP připojení se nejdříve vytvoří klientské vlákno, které ustanoví komunikaci s daným modulem. Toto vlákno poté bude čekat na data přijatá z SSL připojení a rovnou je bude přeposílat danému modulu. Před vstupem do této smyčky, která bude zajišťovat odesílání od klientské části k modulu, se vytvoří ještě *klientské pomocné vlákno*, které bude čekat na příchozí data z modulu a bude je přeposílat přes SSL klientské části. Celá situace je znázorněna na obrázku 3.5. První klientské vlákno je na obrázku označeno jako `clientThread 1` a druhé pomocné klientské vlákno je zde označeno jako `clientThread 2`. Šipky v obrázku znázorňují směr komunikace, které dané vlákno zajišťuje. Obě klientská vlákna se uzavřou při uzavření spojení daným tenantem nebo popřípadě při uplynutí určitého časového intervalu.



Obrázek 3.5: Znázornění komunikace mezi programem `secmux_server` a TCP modulem.

Význam řídící sekvence znaků	Řídící sekvence znaků
EOF	<[secEOF>]
Obsah standardního výstupu o velikosti N	<[oN>]
Obsah standardního chybovéhovýstupu o velikosti N	<[eN>]
Návratový kód aplikace N	<[rN>]

Tabulka 3.1: Řídící sekvence, které se používají při komunikaci mezi klientskou stranou a aplikací, která zpracovává svůj standardní vstup.

Případ komunikace mezi informačním systémem a konzolovou aplikací, jejímž vstupem jsou pouze parametry příkazové řádky spadá z pohledu serverového programu do kategorie TCP spojení, jelikož tento program pouze zprostředuje spojení klientské straně se spouštěčem vzdálených programů.

Poslední možností je modul, který je aplikací, která pracuje se svým standardním vstupem. Po ustanovení spojení a spuštění dané aplikace klientské vlákno běží ve smyčce a veškerá přijatá data vkládá na standardní vstup dané aplikace. Takto činí do doby než obdrží symbol konce vstupu. Klientské vlákno na to opouští smyčku a zavírá standardní vstup dané aplikace. Dále přečte kompletní standardní výstup dané aplikace. Do SSL spojení odešle řídící sekvenci znaků, která značí, že bude odeslán obsah standardního výstup. Tato řídící sekvence obsahuje i velikost výstupu. Obdobně činí i pro standardní chybový výstup, který však předchází řídící sekvence znaků značící standardní chybový výstup. Poté ještě odešle v rámci řídící sekvence znaků i návratový kód aplikace. Na všechny řídí sekvence lze nahlédnout v tabulce 3.1.

Jako demonstrační příklad si uveďme aplikaci, která vše, co dostane na svůj standardní vstup zkopíruje na svůj standardní výstup i na svůj standardní chybový výstup. Ve výpisu 3.4 vidíme odeslání textové řetězce „Hello!“ se znakem nové řádku následované symbolem konce vstupu. Dále ve výpisu je 3.5 je odpověď, kterou obdrží klientská část. Nejdříve obdrží daný textový řetězec ze standardního výstupu, což je označeno na prvním řádku daného výpisu, dále obdrží tento textový řetězec i ze standardního výstupu. Nakonec obdrží návratový kód aplikace, v tomto případě číslo 0.

```
Hello!\n<[secEOF>]
```

Výpis 3.4: Příklad odeslaných dat při komunikaci s aplikací zpracovávající svůj standardní vstup.

```
<[o007>]
Hello!\n
<[e007>]
Hello!\n
<[r0>]
```

Výpis 3.5: Příklad odeslaných dat při komunikaci s aplikací zpracovávající svůj standardní vstup.

3.4.2 Klientský program

Klientský program běží na straně klienta. Při jeho spuštění se určí port, na kterém klient poslouchá požadavky od informačního systému, které přeposílá přes SSL serverovému programu. Tento port je jediným vstupem do klienta. Klientský program při spuštění dostane i certifikát, kterým se bude autentizovat a autorizovat u serverového programu.

Při spuštění klientský program provede jednoduchý test spojení se serverovou částí, kdy přes navázané spojení odešle serveru `HELLO` zprávu, na kterou serverový program při správné autentizaci a korektním běhu zareaguje stejnou zprávou. V případě, že klientský program neobdrží od serveru danou reakci v řádu několika sekund dojde k uzavření klientského programu, protože server s ním nenavázal spojení.

Klientský program pro každé nové spojení na daném portu otevírá nové vlákno, které otevírá nové SSL připojení. Toto vlákno budeme nazývat *vlákno spojení* (anglicky *connection thread*). Toto vlákno komunikuje ve směru od serverového programu k informačního systému. Odbobně jako serverového programu (znázorněno na obrázku 3.5) vzniká ještě pomocné vlákno, které přeposílá komunikaci od informačního systému do serverového programu .

3.4.3 Spouštěč vzdálených programů

Spouštěč vzdálených programů je kontejnerizovaný program běžící na serverové straně, který spouští programy, které jako vstup přijímají parametry přikazové řádky. Tento program je REST API o jednom bodě `/exec/`. Tento bod přijímá:

- JSON požadavek znázorněný ve výpisu 3.3 nebo
- vícero souborů, z nichž jeden obsahuje zmiňovaný JSON požadavek.

První možnost se uplatní ve chvíli, kdy program nepracuje se soubory a svou odpověď vytváří pouze na základě parametrů příkazové řádky. V tomto případě je spuštěn program s parametry z JSON požadavku a poté API vrací odpověď znázorněnou ve výpisu 3.6. V tomto JSON objektu se pod klíčem `stdout` vrací standardní výstup programu, pod klíčem `stderr` se vrací standardní chybový výstup programu a v rámci klíče `returncode` se vrací návratový kód programu.

```
{
   "stdout": STDOUT,
   "stderr": STDERR,
   "returncode": EXIT_CODE
}
```

Výpis 3.6: Odpověď API při použití programů bez souborů.

Uveďme si jednoduchý příklad takového spuštění. Mějme program `soucet`, který očekává dva argumenty – čísla, která sečte a výsledek tohoto součtu vytiskne na svůj standardní výstup. V rámci multitenantní infrastruktury chceme spustit:

```
soucet 2 3
```

Ve výpisu 3.7 vidíme JSON dokument, který bude odeslán v těle požadavku na spouštěč vzdálených programů. Jako spouštěný program uvedeme program `soucet` a do seznamu `args` vložíme čísla 2 a 3.

```
{
   "program": "soucet",
   "args": ["2", "3"],
}
```

Výpis 3.7: Příklad zprávy spuštějící program `soucet` s argumenty 2 a 3.

V následujícím výpisu 3.8 vidíme odpověď na spuštění programu `soucet`. Cílový program běžící v infrastruktuře vytiskl na svůj standardní výstup číslo 5 se znakem nového řádku. standardní chybový výstup programu zůstal prázdný. Program vrátil číslo 0 jako návratový kód.

```
{
   "stdout": "5\n",
   "stderr": "",
   "returncode": 0
}
```

Výpis 3.8: Odpověď API při spuštění programu `soucet` s argumenty 2 a 3.

Druhá možnost očekává, že bude na bod odesláno více souborů, z nichž jeden obsahuje JSON požadavek z výpisu 3.3, tento soubor se jmenuje `json`. V tomto případě vytvoří pracovní adresář, který se po vrácení odpovědi smaže. V tomto pracovním adresáři se vytvoří dva podadresáře – `input` a `output`. Všechny přijaté soubory kromě souboru `json` se uloží do adresáře `input`. Program se spustí ve vytvořeném pracovním adresáři. Po jeho skončení se v adresáři `output` vytvoří tři soubory:

- soubor `stdout` s obsahem standardního výstupu spouštěného programu,
- soubor `stderr` s obsahem standardního chybového výstupu,
- soubor `returncode` s návratovým kódem.

Předpokládá se, že program byl spuštěn tak, že pracoval se soubory, které byly uloženy do složky `input` a vytvořil veškerý výstup do složky `output`. Veškerý obsah podadresáře

`output` se zabalí do archívu (včetně vytvořených souborů s výstupy programu) a ten se odešle zpět v rámci odpovědi na požadavek.

Z důvodu bezpečnosti je v programu spouštěč vzdálených programů omezeno spouštění konzolových aplikací – nelze spusti libovolnou aplikaci, ale pouze aplikaci, která je povolena v souboru `allowed_programs.txt`. Uvedený soubor obsahuje na každém řádku:

- název programu, který lze spouštět nebo,
- název program a jeho argumenty příkazové řádky.

Před spuštěním daného programu se vždy provádí kontrola, zda daný program může být spuštěn. Pokud se název daného programu vůbec v souboru nevyskytuje, nebude vůbec spuštěn. Pokud jsou k danému souboru uvedeny i argumenty, se kterými může být spuštěn, pak jsou tyto argumenty zkontrolovány a na základě této kontroly je buď program spuštěn nebo není spuštěn.

```
someprogram1
someprogram2 p1 p2
```

Výpis 3.9: Příklad souboru `allowed_programs.txt`.

Ve výpisu 3.9 je příklad souboru `allowed_programs.txt`. V tomto případě obsahuje soubor dva povolené programy: `someprogram1` a `someprogram2`. První program může být spuštěn s libovolnými parametry. Druhý program musí mít při spuštění první argument `p1` a druhý argument `p2`. Další argumenty mohou, ale nemusí být zadány a mohou být libovolné.

3.4.4 Falešná aplikace

Falešná aplikace na obrázku 3.3 pojmenována `fake_app2` je program, který působí jako prostředník mezi informačním systémem a programem `secmux_client`. Tato aplikace se před informačním systémem tváří jako původní konzolový program, který byl vyčleněn do cloudu. Přijímá argumenty v totožném pořadí a vypisuje totožné výsledky jako původní program.

Falešná aplikace má pokaždé velmi podobný běh, ale typicky bude nutné před každým nasazením ke konkrétnímu tenantovi provést určité úpravy. Jako první je třeba ve falešné aplikaci nastavit parametry připojení na `cloud` a to zejména nastavit port, na kterém poslouchá `secmux_client`.

Tato aplikace bude odesílat JSON uvedený ve výpisu 3.3. První klíč `program` je možné nastavit napevno nebo popř. nastavit, aby se použil název spouštěného programu. Do klíče `args` se budou vkládat všechny argumenty příkazové řádky, které aplikace obdržela při spuštění.

Jak již bylo nastíněno v textu o spouštěči vzdálených programů, REST API očekává napojení na bod `/exec/` s tím, že JSON požadavek bude odeslán přímo v těle požadavku nebo mezi soubory v souboru `json`.

V případě, že JSON bude odeslán přímo v tělě požadavku a tedy nejsou odesílány žádné soubory, pak se očekává odpověď, která obsahuje výstupy cílové spouštěné aplikace – její standardní výstup, standardní chybový výstup a její návratový kód, tak jak je znázorněno ve výpisu 3.6. Výstupy, pak falešná aplikace vypíše na příslušné místo (standardní výstup či standardní chybový výstup) a vrátí daný návratový kód.

Pokud se pracuje se soubory, pak každý argument, který začíná řetězcem `input/`, se považuje za soubor. Takový soubor se otevře a odešle se na daný bod API. K těmto souborům se přidává ještě soubor `json`, který se naplní JSON dokumentem obsahující parametry spuštění (viz podkapitola 3.3). Všechny tyto soubory jsou odeslány v jednom požadavku. Aplikace očekává v odpovědi na požadavek archív s výsledky spuštění. Tento archív je rozbalen do složky `output`. Soubory `stdout`, `stdin` a `returncode` se použijí jako výstupy aplikace. Pokud archív obsahoval nějaké další výsledky ve formě souborů, pak jsou tyto soubory informačnímu systému (nebo popř. uživateli) k dispozici v adresáři `output`.

3.4.5 Proxy aplikace pro standardní vstup-výstup

Proxy aplikace pro standardní vstup-výstup působí jako prostředník mezi informačním systémem a klientským programem `secmux_client`. Stejně jako falešná aplikace i tento proxy program může před informačním systémem vystupovat jako nezměněná původní aplikace. Na rozdíl od ní však nepracuje s uživatelskými argumenty – ty musí být nastaveny na serverové straně. Cílem této aplikace je pouze přečíst vše ze standardního vstupu a předat to `secmux_server`, od kterého poté aplikace prezentuje výstupy spuštění daného STDIN programu.

Aplikace čte svůj standardní vstup a veškerý obsah odesílá klientskému programu. Jakmile dojde k uzavření standardního vstupu, tak tato aplikace odešle klientskému programu symbol EOF (viz tabulka 3.1). Aplikace poté vždy přečte řídící sekvenci a přečte data velikosti N. Číslo N je přečtené z dané řídící sekvence znaků. Takto činí dokud nepřijde řídící sekvence, značící návratový kód aplikace. Poté vypíše data na standardní výstup a standardní chybový výstup podle toho, jak bylo určeno řídící sekvencí.

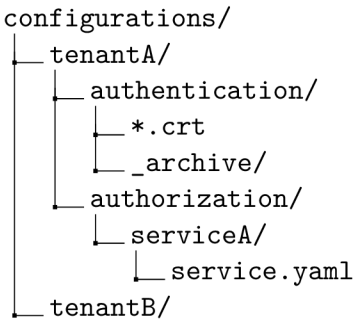
Příklady komunikace mezi multitenantní infrastrukturou a touto aplikací jsou ve výpisech 3.4 a 3.5. Všechny řídící sekvence jsou zdokumentovány v tabulce 3.1.

3.5 Konfigurace tenantů

V podkapitole o architektuře (viz podkapitola 3.3) byla zmíněna konfigurace tenantů na straně serveru, která obsahuje oprávnění a dodatečné informace k certifikátu každého tenanta.

Jako způsob uložení byla zvolena adresářová struktura, jelikož je třeba uložit samotné certifikáty a k nim i textové soubory obsahující konfigurace služeb. Výhodou adresářové struktury je, že je čitelná i lidmi. Dále je možné do této struktury doplnit i další soubory atp., co bude daný provozovatel multitetantního systému potřebovat. Kvůli lidské čitelnosti budou konfigurační soubory ve formátu YAML (viz podkapitola 2.3.6).

Na obrázku 3.6 je znázorněna adresářová struktura, která bude obsahovat konfiguraci certifikátů. Adresář `configurations` obsahuje adresář pro každého tenanta. Složka `tenantA` je složka konkrétního tenanta, která obsahuje dva podadresáře `authentication` a `authorization`. V rámci prvního zmíněného podadresáře najdeme všechny certifikáty (soubory s příponou `.crt`) platné pro daného tenanta, ve složce `_archive` najdeme zneplatněné certifikáty daného tenanta. Druhý podadresář `authorization` je členěn dle služeb (např. `serviceA`), které daný tenant používá. V rámci adresáře pro každou službu se nachází soubor `service.yaml`, který obsahuje konfiguraci dané služby. V tomto souboru se nachází informace pro program `server`, jak s touto službou komunikovat atp.



Obrázek 3.6: Uložení konfigurace tenantů na straně serveru.

3.5.1 Konfigurace služby v souboru service.yaml

Nyní si blíže popíšeme soubor `service.yaml` a co může obsahovat. Soubor je ve formátu `YAML` a musí obsahovat typ služby `type`, který obsahuje některou z následujících hodnot:

- `tcpService` – TCP služba nebo konzolová aplikace očekávající parametry,
- `cliService` – konzolová aplikace očekávající vstup na standardním vstupu.

Pokud se jedná o typ `tcpService`, pak musí povinně obsahovat klíč `bind_port`, který značí port, na který se má serverový program připojovat. Dále může také obsahovat klíč `service_hostname`, který značí adresu (`hostname`) dané služby. Pokud tento klíč v konfiguraci chybí, použije se vychozí název, např. `localhost` či jiný stanovený při spuštění serverového programu. Dále tato konfigurace může obsahovat klíče `container_name` (název kontejneru dané služby) a `container_port`, který značí vnitřní port služby v kontejneru. Tyto dva nakonec zmíněné klíče jsou použitelné ve skriptu startující kontejnery (v případě, že není použita orchestrace kontejnerů přes Kubernetes).

V případě konzolové aplikace, tedy typu `cliService` musí konfigurační soubor obsahovat povinně klíč `program`, který značí název spouštěného programu. Dále může obsahovat i klíč `args`, který obsahuje seznam argumentů pro daný program. Pokud konfigurace neobsahuje klíč `args`, pak se spustí daný program bez argumentů.

Konfigurace může dále obsahovat klíč `name`, který značí jméno daného tenanta. Dále může obsahovat jakýkoliv klíč libovolného názvu, který zde nebyl uveden – serverový program jej bude ignorovat. To je využitelné např. pro potřeby provozovatele infrastruktury, aby si v souborech konfigurace mohl evidovat jakékoliv další informace k tenantovi.

Uveďme si dva příklady, první příklad je ve Výpisu 3.10. Zde vidíme kontjner s Redis databází, který očekává komunikaci na portu 6379. Hostitelské jméno (`hostname`) této služby je `redis-demo-zakaznik1`. Serverový program poté při komunikaci otevře TCP spojení s adresou `redis-demo-zakaznik1:6379`.

```
---
name: redisForZakaznik1
type: tcpService
bind_port: 6379
container_name: redis
container_port: 6379
service_hostname: redis-demo-zakaznik1
```

Výpis 3.10: Příklad konfigurace tenanta komunikujícího s Redis databází

Ve výpisu 3.11 je konfigurace služby, která spouští konzolovou aplikaci, která očekává svůj vstup na standardním vstupu. V tomto případě serverový program spustí konzolový program `some_program` s parametry příkazové řádky `p1` a `p2`. A poté veškerá obdržená data od klientské strany vloží na jeho standardní vstup.

```
---
name: some_programForZakaznik2
type: cliService
program: some_program
args:
  - p1
  - p2
```

Výpis 3.11: Příklad konfigurace tenanta pracujícího s konzolovou aplikací komunikující přes standardní vstup.

Při otevření spojení na základě certifikátu s konfigurací z Výpisu 3.11 bude spuštěn program následujícím způsobem:

```
some_program p1 p2
```

3.5.2 Načítání konfigurace tenantů

Lze předpokládát, že program `server` bude dlouhodobě běžícím programem, který bude neustále obsluhovat nové i stávající připojení. Vzhledem k tomu, že lze minimálně občas očekávat změny v konfiguraci tenantů (např. přidání či odebrání nějakého certifikátu, vytvoření nového tenanta atp.), pak je třeba v rámci programu `server` zavést mechanismus načítání konfigurace tenantů, který bude probíhat nejenom při spuštění tohoto programu.

Opakované načítání konfigurace tenantů se bude periodicky opakovat po celou dobu běhu programu po čase `T`. Po načtení konfigurace tenantů bude každý stávající připojený klient znovu zkontrolován a všechna nová připojení již budou autorizována na základě oné nově načtené konfigurace. Tedy například pokud nějakému stávajícímu připojenému klientovi propadne certifikát, pak bude po načtení nové konfigurace odpojen.

3.6 Účtování

Jedním z požadavků uvedeným v podkapitole 3.2 je účtovatelnost. To znamená pro každého tenanta evidovat objem dat, které za určitý časový úsek prostřednictvím vytvořené infrastruktury přenesl. V rámci naší infrasktruktury musíme rozlišit dva směry komunikace:

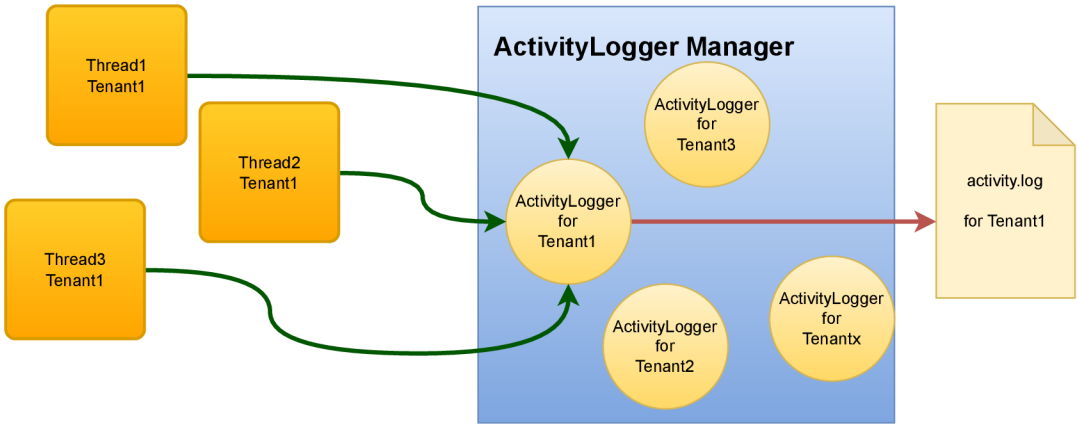
- od informačního systému do vyčleněného modulu,
- z vyčleněného modulu do informačního systému.

Pro každý směr komunikace budeme evidovat počet přenesených bajtů za časovou periodu `T` do určitého času. Účtovatelnost bude probíhat na straně serveru, jelikož zde bude evidence pro všechny klienty a provozovatel serveru může např. dle objemu dat zpoplatňovat poskytovanou službu nebo popř. tyto získané údaje používat k jejich vyhodnocování, např. pro statistické účely.

Účtování bude probíhat pro každý zmíněný směr komunikace zvlášť následujícím způsobem:

Čas záznamu	Velikost obdržených dat
2023-01-21 10:05:55	128B

Tabulka 3.2: Příklad záznamu účtování.



Obrázek 3.7: Ukázka jednotlivých vláken obsluhujících jednoho tenanta přistupujících k příslušnému ActivityLogger, který zapisuje do příslušného logu.

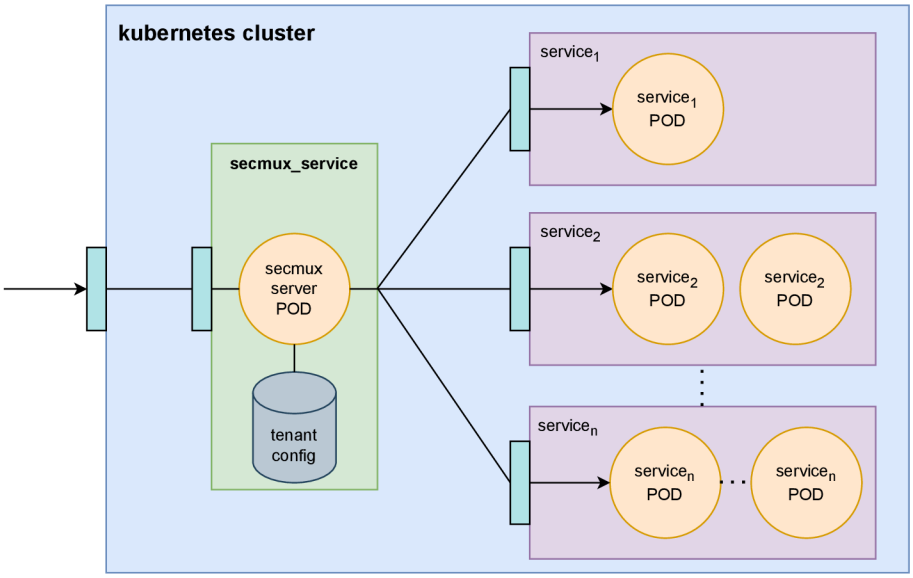
1. Do proměnné `acc` se přičítá velikost obdrženého vstupu,
2. Pokud uplynul čas `T`, proměnná `acc` se zapíše s aktuálním časem do určeného souboru na disk, proměnná `acc` se vynuluje a pokračuje se bodem 1.

V tabulce 3.2 je příklad záznamu účtování. Například pro `T` = 5 minut tento záznam značí, že v čase `2023-01-21 10:00:55` – `2023-01-21 10:05:55` bylo přeneseno právě 128 bajtů.

Uložení souborů obsahujících statistiky přenosu dat jednotlivých tenantů bude v rámci adresářové struktury konfigurace tenantů (viz obrázek 3.6).

O účtování se bude starat `ActivityLogger` – třída, která bude udržovat výše zmíněnou proměnnou `acc` a v určitých časových intervalech zapisovat na disk. To bude dělat pro 1 tenanta a 1 směr komunikace, to znamená, že každého tenanta a jeho službu budou obsluhovat dva objety třídy `ActivityLogger`. Jak již bylo nastíněno v podkapitole 3.4 pro každé nově otevřené spojení bude vznikat nové vlákno (resp. vlákna, ale z pohledu jednoho směru komunikace nyní hovořme o vlákně), což znamená, že pokud by pro každé vlákno vznikal nový `ActivityLogger` znamenalo by to i více záznamů v souboru než je nutné, proto je třeba navrhnout způsob sdílení objektu `ActivityLogger` více vlákny.

Na obrázku 3.7 je znázorněn `ActivityLoggerManager`, což je třída, která spravuje jednotlivé objekty třídy `ActivityLogger`. Mějme tři vlákna, která obsluhují připojení pro nějakého jednoho tenanta. Tyto tři vlákna přistupují k jednomu objektu třídy `ActivityLogger`, která po čase zapisuje do souboru `activity.log` pro daného tenanta.



Obrázek 3.8: Schéma cloudové aplikace orchestrované pomocí Kubernetes. Zelený box představuje službu `secmux_service`, jednotlivá kolečka představují jednotlivé pody, fialové obdelníky znázorňují jednotlivé služby pro tenanty, které obsahují moduly pro dané tenanty. Modul je typicky součástí podu. Tyrkysové boxy představují otevřené porty samotného clusteru či jednotlivých služeb. Pody též typicky komunikují přes své otevřené, ale to není na obrázku znázorněno, protože komunikaci do podu zajišťuje konfigurace služby a samotného podu.

3.7 Orchestrace modulů

Jak již bylo nastíněno v podkapitole 3.3.4 bude použita orchestrace modulů pomocí nástroje Kubernetes. Tento nástroj umožňuje přimočarou správu kontejnerů a také monitoruje stav jednotlivých kontejnerů a v případě potřeby provádí jejich restart.

Na obrázku 3.8 je znázorněno schéma komunikace výsledného Kubernetes clusteru. Kubernetes cluster má otevřený určitý port, který je přímo navázán na serverový program `secmux_server`, který běží v rámci služby `secmux_service`. Součástí této služby je `pod`, který obsahuje kontejner s naším programem. V rámci této služby funguje persistentní úložiště, které obsahuje konfigurace jednotlivých tenantů a jejich účtování.

Program `secmux_server` přepíná mezi jednotlivými službami na základě konfigurace tenantů. Každá služba má vystaven port, na kterém očekává požadavky od informačního systému (přeposlané přes `secmux_server`). V rámci každé služby je možné replikovat kontejnery pro rozložení zátěže.

Kapitola 4

Implementační detaily infrastruktury multitenantních systémů

Tato kapitola se věnuje implementačním detailům infrastruktury – zejména programům `secmux_client` a `secmux_server`. Název `secmux` je odvozen od anglického spojení *secure multiplexing*, což přesně výsledná infrastruktura zajišťuje – zabezpečené spojení po síti těchto dvou vzniklých programů a k tomu bezpečné přepínání (multiplexování) služeb na základě certifikátu tenanta. V této kapitole se nejdříve budeme zabývat soubory této infrastruktury (viz podkapitola 4.1), poté sestavením a spuštěním této infrastruktury (viz podkapitola 4.2). Dále nás bude zajímat konfigurace tenantů na jejímž základě se bude přepínat v podkapitole 4.4. Poté implementace účtování požadavků (viz podkapitola 4.5). Poté se budeme bavit o tom, jak se přistupuje k modulu v podkapitole 4.6. Další důležitou částí dokumentace jednotlivých aktérů, kteří v této infrastruktuře vystupují (viz podkapitola 4.7). Poté nezanedbatelnou částí implementace je logování na standardní výstup v podkapitole 4.8. V další podkapitole 4.9 odbočíme od našich dvou hlavních programů k pomocným aplikacím. Na závěr této kapitoly se pobavíme o orchestraci modulů v podkapitole 4.10.

4.1 Soubory infrastruktury

Jak již bylo nastíněno v podkapitole 3.3 hlavními dvěma programy v infrastruktuře jsou program `secmux_client` a `secmux_server`. V rámci této podkapitoly si krátce představíme jednotlivé soubory obou těchto programů.

Jako první si představme soubory klientského programu:

- `Program.cs` obsahuje hlavní třídu programu, která zpracuje argumenty příkazové řádky při spuštění a poté předá řízení objektu třídy `Client`.
- `Client.cs` obsahuje třídu `Client`, která navazuje zabezpečený přenos s serverovým programem (viz podkapitola 4.3.2) a otevírá port pro komunikaci s informačním systémem. Tuto komunikaci přeposílá serverovému programu přes zabezpečenou komunikaci.
- `Logger.cs` obsahuje statickou třídu `Logger`, která zajišťuje logování událostí na standardní výstup programu (viz podkapitola 4.8).

Dále si představme soubory serverového programu:

- `Program.cs` je hlavní soubor programu s hlavní třídou, která zpracuje parametry příkazové řádky a předá řízení objektu třídy `Server`.
- `Server.cs` obsahuje třídu `Server`, která řídí zabezpečenou komunikaci (viz podkapitola 4.3.1), přeposílá komunikaci modulu a spolupracuje s ostatními při logování aktivit atp.
- `ActivityLogger.cs` obsahuje stejnojmennou třídu, která zajišťuje logování aktivit pro účtování (viz podkapitola 4.5).
- `ActivityLoggerManager.cs` obsahuje několik tříd včetně stejnojmenné třídy, které spravují instance třídy `ActivityLogger` pro jednotlivé tenanty (viz podkapitola 4.5).
- `ControlSequence.cs` obsahuje statickou třídu `ControlSequence` pro vytváření kontrolních sekvencí z tabulky 3.1.
- `Logger.cs` obsahuje statickou třídu `Logger`, která zajišťuje logování událostí na standardní výstup programu (viz podkapitola 4.8).
- `ModuleInterface.cs` obsahuje stejnojmennou třídu, která zajišťuje přístup k modulu (viz podkapitola 4.6).
- `ModuleInterfaceFactory.cs` obsahuje třídu se stejným názvem jako tento soubor, tato třída na základě konfigurace tenanta vytvoří instanci třídy `ModuleInterface` pro přístup k danému modulu (viz podkapitola 4.6).
- `ServiceConfigurationManager.cs` obsahuje stejnojmennou třídu, která spravuje konfigurace tenantů, zodpovídá za jejich načítání a zpřístupňování (viz podkapitola 4.4).
- `SslStreamInfo.cs` obsahuje stejnojmennou statickou třídu pro logování vlastností navázaného SSL spojení.

4.2 Spuštění infrastruktury

Ve výpisu 4.1 jsou příkazy, které sestaví oba programy. V případě lokálního spuštění je ještě třeba ve složce se sestaveným serverovým programem[1] vytvořit adresář s názvem `client_certificates` a naplnit ji adresářovou strukturou podobnou jako na obrázku 3.6, aby mohli být daní tenanti autentizováni a poté i obslouženi.

```
cd client
dotnet build
```

Výpis 4.1: Sestavení obou programů.

Ve výpisu 4.2 je příkaz pro spuštění serverového programu. Spouští se pomocí programu `dotnet`, soubor `server.dll` obsahuje jazyk střední úrovně, který vznikl kompilací z jazyka C# (viz podkapitola 2.3.5). Nyní si projdeme jednotlivé parametry spuštění:

- `$CERTIFICATE_FILE` je cesta na disku k souboru certifikátu serveru ve formátu `.pfx`,

[1] typicky `server/server/bin/Debug/net6.0`

- `$CERTIFICATE_PASSWORD` je heslo k zadanému certifikátu,
- `$SERVER_PORT` je celé číslo portu, na kterém bude serverový program poslouchat příchozí data od klientské části přes SSL,
- `$MODULES_HOSTNAME` je výchozí hostitelské jméno pro moduly, které jej nemají uvedené ve své konfiguraci, lze použít například `localhost`,
- `$ACTIVITY_LOGGING_INTERVAL` je počet minut, které značí, jak často se má zapsat záznam o účtování na disk.

Ještě lze nastavit proměnnou prostředí `REFRESH_TIME` pro změnu intervalu načítání konfigurace – lze například využít při testování.

```
dotnet server.dll $CERTIFICATE_FILE $CERTIFICATE_PASSWORD
$SERVER_PORT $MODULES_HOSTNAME $ACTIVITY_LOGGING_INTERVAL
```

Výpis 4.2: Spuštění programu `secmux_server` lokálně.

Ve výpisu 4.3 je ukázka spuštění klienta[2]. Na rozdíl od serverového programu není třeba přidávat žádné adresáře atp. Klient může být spuštěn „ihned“ po sestavení. Jednotlivé parametry spuštění jsou:

- `$SERVER_HOSTNAME` je hostitelské jméno serveru – IP adresa, doménové jméno nebo v případě spuštění v rámci jednoho počítače `localhost`,
- `$SERVER_PORT` je port serveru, který očekává komunikaci od klienta přes SSL (pozn. při spuštění serveru ve výpisu 4.2 je tato proměnná pojmenována stejně),
- `$SERVER_NAME` je název serveru, pole `CN` v certifikátu serveru (musí shodovat jinak nebude spojení navázáno),
- `$CLIENT_CERTIFICATE` je cesta na disku k souboru certifikátu klienta ve formátu `.pfx`,
- `$CLIENT_PASSWORD` je heslo k zadanému certifikátu,
- `$OUT_PORT` je celé číslo portu, které bude otevřeno na straně klienta pro poslech příchozích dat od informačního systému,
- `$LOCAL_IP` je lokální IP adresa klienta.

```
dotnet client.dll $SERVER_HOSTNAME $SERVER_PORT $SERVER_NAME
$CLIENT_CERTIFICATE $CLIENT_PASSWORD $OUT_PORT $LOCAL_IP
```

Výpis 4.3: Spuštění programu `secmux_client`.

Ve výpisu 4.4 jsou znázorněny příkazy, které vedou ke spuštění serverového programu v rámci Kubernetes. Nejdříve je potřeba ve složce `server` sestavit Docker obraz a ten načíst do Kubernetes – v případě použití programu `minikube` lze využít uvedený příkaz. Poté stačí ve složce `kubernetes` aplikovat soubor s konfigurací pro serverový program. Klientský program se v tomto případě stále spouští lokálně. V případě použití `minikube` je třeba ještě spusti tunelování portů[3], aby bylo možné se k serverovému programu dostat.

[2] Typicky výstup sestavení klienta je ve složce: `client/client/bin/Debug/net6.0`.
[3] příkaz: `minikube tunnel`

```
cd server
docker build -t secmux-server .
minikube image load secmux-server
cd ../kubernetes
kubectl apply -f secmux.yaml
```

Výpis 4.4: Spuštění programu `secmux_server` přes Kubernetes.

Před spuštěním serveru je třeba opět nakonfigurovat tenanty ve zmiňovaném adresáři `client_certificates`. Jsou dvě možnosti:

1. vytvořit složku `client_certificates` ve složce `server` a její obsah se vloží do Docker obrazu,
2. vytvořit persistetní úložiště v rámci Kubernetes a tenanty nakonfigurovat zde.

4.3 Zabezpečený přenos

Zabezpečený přenos mezi sebou ustanovují programy `secmux_client` a `secmux_server`. Oba jsou implementovány v jazyce C#. SSL přenos je obsluhován pomocí vestavěné třídy `SslStream` [13]. Tato třída je používána jak na serverové straně, tak i na straně klienta.

4.3.1 Zabezpečený přenos na straně serveru

Na straně serveru nejdříve dojde k otevření portu, na kterém server bude přijímat SSL připojení. K tomuto otevření dojde vytvořením objektu třídy `TcpListener` a zavoláním jeho metody `listen`. Poté vstupujeme do nekonečné smyčky, která začíná voláním blokující metody vytvořeného objektu `AcceptTcpClient`. Jakmile dojde k připojení tato metody vrací objekt třídy `TcpClient`. Stream tohoto objektu předáme konstruktoru třídy `SslStream` společně s callbackem `ValidateClientCertificate`, který ověří, zda klientský certifkát neobsahuje některou chybu z výčtu `SslPolicyErrors` a také zda je certifikát známý. Tedy že je obsažen v načtené kolekci certifikátů.

Pokud kontroly výše projdou, pak se náš program pokusí autentizovat na straně klienta na základě certifikátu serveru, který byl zadán při spuštění tohoto programu. Pokud je i tato část úspěšná, pak se vytvoří instance struktury `NewClient`, které jsou předány do konstruktoru:

- identifikátor klienta – celé číslo, pořadové číslo připojeného klienta,
- objekt třídy `TcpClient`,
- úspěšně oboustranně autentizované spojení v objektu třídy `SslStream`.

Tento nově vytvořený objekt třídy `NewClient` je předán nově vytvořenému klientskému vláknu, které bude dané TCP spojení obsluhovat. Objekt obsahuje pouze tři atributy uvedené výše, které lze veřejně číst, ale nelze je zapisovat mimo konstruktor (ve smyslu zápisu jiného objektu) po zbytek programu.

V rámci nově vytvořeného klientského vlákna dojde na základě konfigurace tenantů k napojení na daný modul (viz podkapitola 4.4 a 4.6). Pokud je vše v pořádku z hlediska konfigurace, tak se očekává zpráva `HELLO`, kterou server obdrží od tohoto klienta přes SSL,

na kterou server zareaguje totožnou zprávou (viz 4.3.2). Poté je zabezpečený přenos obsluhován pomocí dvou metod objektu třídy `SslStream`:

- `Read` – blokující metoda, která přečte data požadované délky ze streamu a vloží je do bajtového bufferu,
- `Write` – metoda, která zapíše data z bajtového bufferu do streamu.

4.3.2 Zabezpečený přenos na straně klienta

Na klientské straně je průběh podobný serverové straně. Při spuštění programu dojde k „pokusné autentizaci“, kdy se klient napojí na server přes SSL odešle mu zprávu `HELLO`, na kterou očekává stejnou reakci. To je z několika důvodů:

- ihned po otevření programu je vhodné zjistit zda se lze u serveru autentizovat přes zadaný certifikát a případně program rovnou ukončit,
- v případě neúspěšné autentizace klientským certifikátem objekt třídy `SslStream` nevyhazuje žádnou výjimku, je tedy třeba provést testovací spojení, které úspěšnost autentizace spojení ověří.

Po testovacím spojení klientský program obdobně jako server otevírá port pro příchozí požadavky, tentokrát pro informační server. Opět se tak děje pomocí objektu třídy `TcpListener` a jednotlivá spojení ze strany informačního systému jsou akceptována metodou `AcceptTcpClient` v nekonečné smyčce. Tentokrát však toto příchozí spojení není šifrováno. Po přijetí spojení se vytváří objekt třídy `NewConnection`, který ve svém konstruktoru obdrží:

- identifikátor daného spojení – celé pořadové číslo spojení,
- objekt třídy `TcpClient`.

Tato třída je de facto stejná jako třída `NewClient` na serverové straně, liší se pouze názvem a tím, že neukládá objekt třídy `SslStream`. Vytvořený objekt třídy `NewConnection` se předá nově vytvořenému vláknu spojení, které bude toto spojení dále obsluhovat.

Vytvořené klientské vlákno nejdříve vytvoří objekt třídy `SslStream` včetně zmiňovaného testu pomocí `HELLO` zpráv. Pokud vše proběhne v pořádku, pak se vytvoří pomocné vlákno spojení. To vzniká tak, že nejdříve vytvoří instanci struktury `Streams`, která v konstruktoru přijímá:

- identifikátor spojení (viz výše),
- objekt třídy `SslStream`,
- objekt třídy `Stream` – proud dat napojený na informační systém.

Obdobně jako v případě objektu třídy `NewConnection` se i objekt této třídy předá nově vytvořenému vláknu. Nyní v prvním vláknu spojení poběží nekonečný cyklus, který bude začínat přečtením z objektu třídy `SslStream` (pomocí metody `Read`) a kompletní zápis všech přijatých bajtů do streamu informačního systému. Pomocné vlákno zase přečte ze streamu informačního systému a zápiše do objektu třídy `SslStream` (pomocí metody `Write`).

```
recv = sslStream.Read(buffer, 0, buffer.Length);
stream.Write(buffer, 0, recv);
```

Výpis 4.5: Komunikace v prvním vlákně spojení.

Výpis 4.5 zobrazuje komunikaci z první vlákna spojení. Metoda `Read` načte data do proměnné `buffer`, maximálně velikosti délky bajtového pole uložené v této proměnné. Do proměnné `recv` vrátí délku skutečně uložených dat. Poté se zavolá metoda `Write` objektu třídy `Stream`, který je uložen v proměnné `stream`. Do streamu k informačnímu systému budou zapsána data z proměnné `buffer` od nulté pozice až pozici uloženou v proměnné `recv`.

Obě vlákna pro daná spojení fungují do chvíle než informační systém uzavře dané spojení nebo popř. nastane `timeout` daného spojení.

Speciální možností ukončení vlákna spojení a s ním i celého programu včetně všech ostatních vytvořených vláken je moment, kdy serverový program zjistí, že danému tenantovi propadla autentizace (o obnovování konfigurací viz podkapitola 4.4). V tu chvíli serverová strana odesílá `STOP` symbol a uzavře spojení. Vlákno spojení na obdržení `STOP` symbolu reaguje ukončením celého programu.

4.4 Konfigurace tenantů

Konfiguraci tenantů jsme navrhli v podkapitole 3.5. Konfigurace bude uložena v adresářové struktuře (viz obrázek 3.6), která obsahuje klientské certifikáty (ve formátu `crt`) a samotný soubor konfigurace `service.yaml` (viz výpisy 3.10 a 3.11). V této podkapitole budeme nejdříve diskutovat vnitřní reprezentaci této konfigurace a poté její načítání.

4.4.1 Vnitřní reprezentace konfigurace tenantů

Vnitřní reprezentaci konfigurace tenantů spravuje třída `ServiceConfigurationManager`, která je v programu reprezentována jediným objektem, který je uložen v atributu v hlavní třídě `Server`. Tato třída obsahuje dvě privátní kolekce. První takovou kolekcí je třída `X509CertificateCollection` zvaná `knownCertificates`, která obsahuje všechny známé certifikáty pro rychlé dotazování, zda je certifikát známý.

Druhou kolekcí je slovník `serviceConfigurations`, který obsahuje klíče typu textový řetězec a hodnotu třídy `ServiceConfiguration`, což je třída veřejných atributů, která má připraveny tyto atributy pro jednotlivé atributy v souboru `service.yaml`. Při načtení do této kolekce se jako klíč použije hash daného certifikátu získaný jeho veřejnou metodou `GetCertHashString`, která je součástí třídy `X509Certificate`. Poté při vyhledávání nějaké informace o tenantovi se použije hash certifikátu, pomocí kterého se autentizoval.

Jelikož jsou obě uvedené kolekce privátní je třeba k nim v objektu připravit metody, které k nim zajistí kontrolovaný přístup. Pro dotazování nad první kolekcí certifikátů `knownCertificates` slouží metoda `isKnownCertificate`, která dostane objekt certifikátu, instanci třídy `X509Certificate`, která v jazyce C# reprezentuje X509 certifikáty. Metoda vrací `true`, jestliže se daný certifikát v této kolekci nachází a zároveň se hash tohoto certifikátu nachází jako klíč i v druhé kolekci `serviceConfigurations`. Tato metoda se používá při autentizaci klienta (viz podkapitola 4.3.1).

Nad druhou kolekcí existuje několik metod o jednom parametru hash certifikátu, které vrací hodnoty z přidruženého objektu `ServiceConfiguration`. Ve výpisu 4.6 je příklad

takové metody, která na základě certifikátu vrátí přidružený port z konfigurace. Všechny další metody vypadají velmi podobně.

```
public int getPortByCertificate(X509Certificate certificate)
{
   string h = certificate.GetCertHashString();
   return this.serviceConfigurations[h].bind_port;
}
```

Výpis 4.6: Příklad metody přistupující ke kolekci konfigurací certifikátů.

Poslední metodou, kterou třída obsahuje je metoda kontrolující správnost konfigurace `getConfigurationCorrectnessByCertificate`. Tato metoda zkontroluje zda konfigurace (opět na základě hashe certifikátu) obsahuje některou povinnou kombinaci neprázdných atributů. Jsou dvě možnosti:

- typ služby je `tcpService` a port je nastaven (čili je vyšší než nula),
- typ služby je `cliService` a program je neprázdný textový řetězec.

Pokud platí některá z odrážek výše, pak metoda vrací pravdu jinak vrací nepravdu.

4.4.2 Načítání konfigurace tenantů

O načítání konfigurace tenantů z adresářové struktury do popsané vnitřní reprezentace se stará metoda `loadCertificates` patřící do třídy `ServiceConfigurationManager`. Tato metoda naplní zmíněné kolekce informacemi. Metoda nejdříve načte obsah složky, která obsahuje adresáře jednotlivých tenantů. V rámci složky každého tenanta navštíví adresář `authentification`, ve kterém projde všechny podadresáře. Pro každý podadresář kromě podadresáře `archive`, který obsahuje validní certifikát s příponou `.crt`:

- vloží tento certifikát do kolekce `knownCertificates`,
- dále projde všechny podadresáře daného tenanta ve složce `authorization`, které obsahují soubor `service.yaml`:
  - daný soubor načte a převede do objektu třídy `ServiceConfiguration`,
  - do tohoto objektu dále přidá klíč `user_role`, tedy uživatelskou roli – název podadresáře v autentizační složce, ve které se daný certifikát nachází,
  - název služby `service_name`, což je název podadresáře v autorizačním adresáři, ve kterém je soubor `service.yaml`,
  - cestu k autorizačnímu adresáři `authorization_directory`,
  - jméno tenanta `tenant`, což je dané názvem složky,
  - nakonec je daný objekt třídy `ServiceConfiguration` přidán do kolekce konfigurací `serviceConfigurations` pod klíčem – hashem daného certifikátu.

4.4.3 Obnovování konfigurace tenantů

Z důvodu možnosti změn v konfiguraci a dlouhodobého běhu programu je třeba periodicky načítat konfiguraci tenantů. První načtení konfigurace proběhne hned po startu programu ještě před startem cyklu obsluhujícího příjem nových spojení přes SSL. Po tomto

načtení se vytvoří validační vlákno, které se skládá z jednoho nekonečného cyklu. Krok cyklu začíná tím, že se uspí na jednu hodinu (nebo na čas určený proměnnou prostředí `REFRESH_TIME`). Poté dojde k načtení nové konfigurace metodou `loadCertificates` z uloženého objektu třídy `ServiceConfigurationManager`. Po načtení inkrementuje sdílená proměnná `last_validated`.

Proměnná `last_validated` je čítač, který značí kolikrát byla načtena konfigurace tenantů. Při vytvoření nového klientského vlákna dojde k autentizaci i autorizaci klienta. Pokud jsou úspěšné, pak vlákno si udržuje lokální proměnnou `local_last_validated`, do které se zkopíruje aktuální hodnota ze sdílené proměnné. Klientské vlákno si poté pravidelně kontroluje, že globální proměnná `last_validated` je rovna lokální proměnné, jakmile není, pak to znamená, že byla načtena nová konfigurace tenantů validačním vláknem. Na to klientské vlákno reaguje tím, že provede novou autentizaci i autorizaci klienta a pokud například klientovi od poslední kontroly jeho autentizace propadla, pak tomuto klientovi pošle `STOP` symbol a zavře s ním spojení.

4.5 Účtování

Účtování bylo navrženo v podkapitole 3.6. Logování aktivity jednoho tenanta provádí objekt třídy `ActivityLogger`, ke kterému klientské vlákno dostane přístup od objektu třídy `ActivityLoggerManager`, jehož jediná instance je uložena v atributu hlavního objektu třídy `Server`. Manažerská třída obsahuje privátní slovník `storage`, který obsahuje klíče typu textový řetězec a hodnoty struktury `ActivityLoggersRecord`. Uvedená záznamová třída se skládá ze dvou veřejných atributů pro čtení:

- celého čísla `refCount`, které značí počet referencí na daný `ActivityLogger` s počáteční hodnotou nula,
- instance třídy `ActivityLoggers`, která obsahuje dva objekty třídy `ActivityLogger`, jeden pro účtování komunikace od informačního systému k modulu a druhý opačným směrem.

Záznamová třída dále obsahuje konstruktor pro nastavení atributů a dvě veřejné metody: `incRefCount` a `decRefCount` pro inkrementaci nebo dekrementaci atributu `refCount`.

Objekt třídy `ActivityLoggerManager` při své inicializaci dostane v konstruktoru parametr `interval`, který značí interval logování. Tuto hodnotu si uloží do privátního atributu a bude jej používat při inicializaci objektů třídy `ActivityLogger`. Dále také v konstruktoru vytvoří prázdný slovník `storage`, který bude obsahovat záznamy pro jednotlivé tenanty. Jako klíč se používá jméno tenanta. Manažer obsahuje dvě veřejné metody `subscribe` a `unsubscribe`.

Metoda `subscribe` se volá na začátku klientského vlákna před zahájením komunikace. Na základě jména tenanta tato metoda vrátí příslušný objekt třídy `ActivityLoggers`. Metoda se nejdříve podívá do slovníku `storage` a pokud tam je uložen záznam pro příslušného tenanta, pak vrátí v něm obsažený objekt. Pokud záznam o daném tenantovi ve slovníku není, pak záznam vytvoří a vratí nově vytvořený objekt `ActivityLoggers`. V každém případě před návratem inkrementuje `refCount` v příslušném záznamu.

Metoda `unsubscribe` slouží k informování manažera o tom, že objekt `ActivityLoggers` nebude klientské vlákno již využívat – volá se těsně před jeho skončením. Tato metoda sníží o jedna `refCount` u příslušného tenanta. Pokud jeho hodnota klesne na nulu, pak je celý záznam ze slovníku odstraněn, jelikož žádné klientské vlákno nyní tento objekt nepotřebuje.

Objekt třídy `ActivityLogger` se vytváří pro logování jednoho směru komunikace (buď od informačního systému k modulu nebo naopak). Při tvorbě instance konstruktor obdrží název souboru, do kterého se bude logovat, dále obdrží interval logování a typ (podle zmíněného směru). Poté se očekává, že vždy při příslušném směru komunikace se zavolá metoda `log` této instance, která obdrží velikost dat v bajtech. Ta se přičte do akumulátoru, který je na záčatku nastaven na nulu. Hodnota v akumulátoru bude v určité chvíli zapsána do daného souboru.

K určení okamžiku, kdy je třeba vynulovat akumulátor a zapsat jeho původní obsah na disk slouží privátní atribut `stopClock`, který obsahuje hranici – časový údaj, do kdy je možné do akumulátoru přičítat. První nastavení stop hodin probíhá již v rámci konstruktoru. Stop hodiny se nastavují vždy na aktuální čas plus nastavený interval, to celé se ještě zaokrouhlí na celé minuty nahoru. Při zavolání metody `log` se kontrolují stop hodiny a v případě, že se zjistí, že aktuální čas již je větší než stop hodiny, tak se velikost přijatá v parametru již do akumulátoru nepřičítá. Obsah akumulátoru se zapíše do daného souboru s hodnotou ze stop hodin. Poté proběhne stejným způsobem jako v konstruktoru nové nastavení stop hodin a hodnota akumulátoru se nastaví na hodnotu velikosti vstupu, která byla přijata v parametrech při volání.

Níže je příklad záznamu zalogovaný objektem `ActivityLogger`, který značí, že dne 10. dubna 2023 bylo za interval `T` do času 10:02:00 přeneseno 35 bajtů:

```
2023-04-10 10:02:00, 35B
```

Zápis do souboru provádí veřejná metoda `writeFile`, která je volána buď zmíněnou metodou `log` nebo je volána z vnější objektu typicky objekt třídy `ActivityLoggerManager`, který se chystá daný `ActivityLogger` odstranit a ještě předtím než tak učiní nechá odstraňovaný objekt zapsat hodnotu z akumulátoru do souboru.

4.6 Přístup k modulu

Přístup k modulu zajišťuje objekt třídy `ModuleInterface`, který klientské vlákno obdrží od objektu třídy `ModuleInterfaceFactory`. Klientské vlákno brzy po vytvoření zavolá metodu `create` továrny a předá jí objekt certifikátu klienta. Továrna má přístup k objektu třídy `ServiceConfigurationManager`, od které na základě poskytnutého certifkátu nejdříve zjistí jakého typu daná služba je (TCP nebo Console). Poté si dle právě zjištěného typu z konfigurace zjistí potřebné povinné atributy dané služby. Pro TCP službu si zjistí port a hostname, pro konzoli si zjistí název programu a popřípadě i jeho argumenty příkazové řádky pro spuštění. Tyto informace předá konstruktoru třídy `ModuleInterface` a vrací vzniklou instanci.

Třída `ModuleInterface` má přetížený konstruktor. Jak již bylo naznačeno pro TCP konstruktor očekává číslo portu a název hostname. Pro konzoli očekává název program a pole argumentů. Výsledný objekt se tváří navenek stejně, ale očekává se vždy trochu jiné volání metod:

Pro TCP modul lze volat následující metody:

- metodu `Read` pro přečtení bajtů ze spojení do poskytnutého bufferu při volání,
- metodu `Write` pro zápis bajtů do spojení z poskytnutého bufferu.

Pro konzolový modul lze volat následující metody:

- metodu `Read` pro přečtení 1 řádku ze standardního výstupu do bufferu poskytnutého při volání,
- metodu `Write` pro zápis bajtů na standardní vstup z bufferu,
- metodu `CloseStandardInput`, která zavře standardní vstup spuštěné konzolové aplikace,
- metodu `ReadStandardError`, která přečte maximálně možný počet bajtů z chybového standardního výstupu do bufferu,
- metodu `ReadStandardOutput` pro čtení ze standardního výstupu, obdobně jako předchozí metoda,
- metodu `getReturnCode` pro navrácení návratové hodnoty daného programu.

V obou případech modulů se po ukončení práce volá `Close`, která zavře všechny otevřené proudy, procesy atp.

V případě TCP modulu se ke komunikaci používá objekt třídy `Stream`, který je uložen v privátním atributu. V případě konzolového modulu se používá objekt třídy `Command` z balíku `MedallionShell`[4]. Objekt obsahuje tři veřejné atributy: `StandardError` reprezentující standardní chybový výstup spouštěného programu, `StandardOutput` představující jeho standardní výstup a `StandardInput` reprezentující jeho standardní vstup. Každý z nich obsahuje veřejný atribut proudu dat – `BaseStream`, který se poté používá při komunikaci.

4.7 Aktéři

V rámci vytvořené infrastruktury vystupuje několik aktérů, kteří mezi sebou vzájemně intereagují. Program `secmux_client` považujme za jednoho aktéra nazvaného jako klientský program. Program `secmux_server` je rozdělen do tří aktérů:

- hlavní vlákno,
- validační vlákno,
- klientské vlákno.

Na krátký popis jednotlivých aktérů lze nahlédnout v tabulce 4.1. V tabulce 4.3 lze vidět jednotlivé akce, které v systému mohou nastat a jakými událostmi na ně jednotlivý aktéři reagují. V tabulce 4.2 je přehled jednotlivých událostí.

Na základě těchto akcí a reakcí vznikly automaty, které znázorňují jednotlivé aktéry. Blíže se je popíšeme dále v této podkapitole. Tyto vzniklé automaty budou dále použity v kapitole 5 (konkrétně podkapitola 5.2) při testování systému.

4.7.1 Hlavní vlákno

Na obrázku 4.1 je znázorněn automat hlavního vlákna, v tabulce 4.4 je popis jednotlivých stavů. Hlavní vlákno se spustí se startem serverového programu. Po spuštění se nachází ve výchozím stavu, ve kterém se pokusí připravit na příjem příchozích spojení. To se mu

[4] https://github.com/madelson/MedallionShell

Aktér	Popis
Hlavní vlákno	Provede základní konfiguraci serveru, vytvoří validační vlákno a poté vytváří vlákna pro jednotlivé nově připojující se klienty.
Validační vlákno	Periodicky po čase T načítá konfiguraci certifikátů a certifikáty.
Klientské vlákno	Přijímá požadavky klienta, provádí jeho prvotní autorizaci a také průběžnou autentizaci a autorizaci. Přímo komunikuje s modulem.
Klientský program	Odesílá požadavky serveru a přijímá odpovědi k těmto požadavkům.

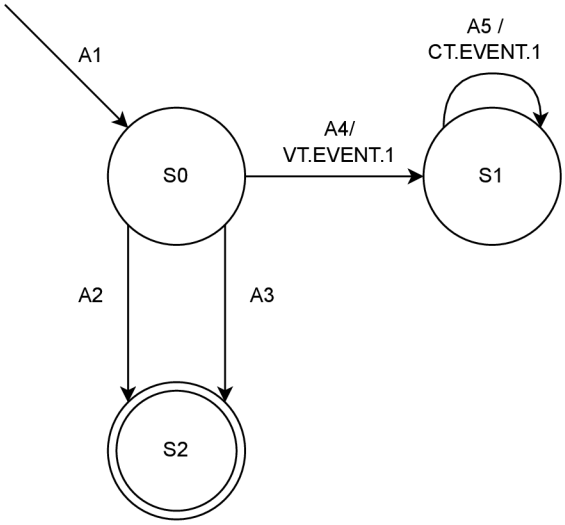
Tabulka 4.1: Přehled jednotlivých aktérů a jejich popis. Reakce těchto aktérů událostmi z tabulky 4.2 na akce jsou uvedeny v tabulce 4.3.

Identifikátor	Událost
`EVENT.1`	Vznik tohoto aktéra
`EVENT.2`	Ukončení tohoto aktéra
`EVENT.3`	Přijetí zprávy
`EVENT.4`	Provede autentizaci
`EVENT.5`	Vytvoří klientské vlákno
`EVENT.6`	Provádí novou validaci klienta
`EVENT.7`	Ignoruje tento certifikát
`EVENT.8`	Přeposlání serverové části
`EVENT.9`	Přeposlání modulu
`EVENT.10`	Přeposlání klientskému programu
`EVENT.11`	Přeposlání IS

Tabulka 4.2: Přehled událostí, tyto události jsou reakcemi na akce v tabulce 4.3.

ID	Akce	Hlavní vlákno	Validační vlákno	Klientské vlákno	Klientský program
A1	Spuštění serveru	EVENT.1	-	-	-
A2	Certifikát serveru neexistuje	EVENT.2	-	-	-
A3	Certifkát serveru není v korektním formátu	EVENT.2	-	-	-
A4	Vytvoření validačního vlákna	X	EVENT.1	-	-
A5	Připojení nového klienta	EVENT.4, EVENT.5		EVENT.1	X
A6	Selhání autorizace klienta			EVENT.2	EVENT.2
A7	Načtení nové konfigurace certifikátů		X	EVENT.6	
A8	Klientský certifikát není v korektním formátu		EVENT.7	-	-
A9	Ukončení klientského vlákna				EVENT.2
A10	Uzavření klienta			EVENT.2	EVENT.2
A11	Propadnutí validace klienta			EVENT.2	EVENT.2
A12	IS posílá požadavek				EVENT.3, EVENT.8.
A13	Klientský program přeposílá požadavek			EVENT.3, EVENT.9	X
A14	Přijetí odpovědi modulu			EVENT.10	EVENT.11
A15	Uspání validačního vlákna		X		

Tabulka 4.3: Přehled jednotlivých aktérů, událostí a reakcí těchto aktérů na ně. Akce je událost, na kterou aktér reaguje nebo ji sám vykonává, popř. ji ignoruje. Pokud je pro událost u daného aktéra symbol X, znamená to, že aktér tuto událost sám vykonává. Pokud je zde symbol pomlčky (-), tak aktér v době této události ještě neexistuje. Pokud zde nic není (prázdná buňka), pak aktér událost ignoruje nebo o ní ani neví, tj. ani nemůže nijak reagovat.



Obrázek 4.1: Diagram automatu hlavního vlákna. Přechod značí událost, která nastala a případná reakce ostatních vláken na tuto událost (událost/reakce). Jednotlivé stavy jsou popsány v tabulce 4.4.

Stav	Popis
S0	Created
S1	Waiting for incoming connections
S2	Finished

Tabulka 4.4: Popis jednotlivých stavů automatu hlavního vlákna z obrázku 4.1.

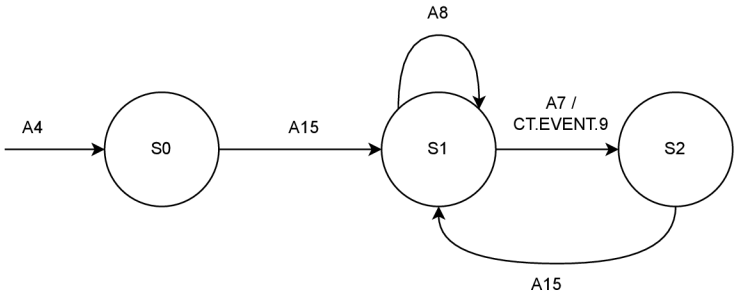
nepodaří, jestliže dostane cestu k neexistující souboru certifikátu nebo pokud soubor certifikátu není v korektním formátu. Pokud se některá z těchto dvou možností stane, pak se přesouvá do koncového stavu. Pokud vše proběhne v pořádku, vytvoří validační vlákno a pak se přesouvá do stavu, kde akceptuje jednotlivé nově příchozí spojení, ve kterém se nachází po zbytek běhu programu.

4.7.2 Validační vlákno

Automat tohoto vlákna je vyzobrazen na obrázku 4.2, popis jednotlivých stavů je uveden v tabulce 4.5. Validační vlákno ihned po vytvoření přechází uspáním do stavu načítání konfigurací, pokud narazí na neplatnou konfiguraci, pak ji přeskakuje a běží dál. Po dokončení načtení konfigurace přechází do stavu, kdy je konfigurace načtena a z tohoto stavu se opět uspáním vrací do stavu načítání konfigurace.

Stav	Popis
S0	Created
S1	Configuration Loading
S2	Configuration Loaded

Tabulka 4.5: Popis jednotlivých stavů automatu validačního vlákna z obrázku 4.2.



Obrázek 4.2: Diagram automatu validačního vlákna. Přechod značí událost, která nastala a případná reakce ostatních vláken na tuto událost (událost/reakce). Jednotlivé stavy jsou popsány v tabulce 4.5.

Stav	Popis
S0	Created
S1	Prepared
S2	Communicating
S3	re-Authentification & re-Authorization
S4	Finished

Tabulka 4.6: Popis jednotlivých stavů automatu klientského vlákna z obrázku 4.3.

4.7.3 Klientské vlákno

Klientské vlákno je znázorněno automatem v obrázku 4.3, jehož stavy jsou popsány v tabulce 4.6. Po vytvoření klientského vlákna se v počátečním stavu pokusí autorizovat klienta, což se nemusí podařit pokud byla konfigurace tenanta špatně nastavena. Pokud se tak nepovede, přesouvá se do koncového stavu – s tímto klientem nelze komunikovat.

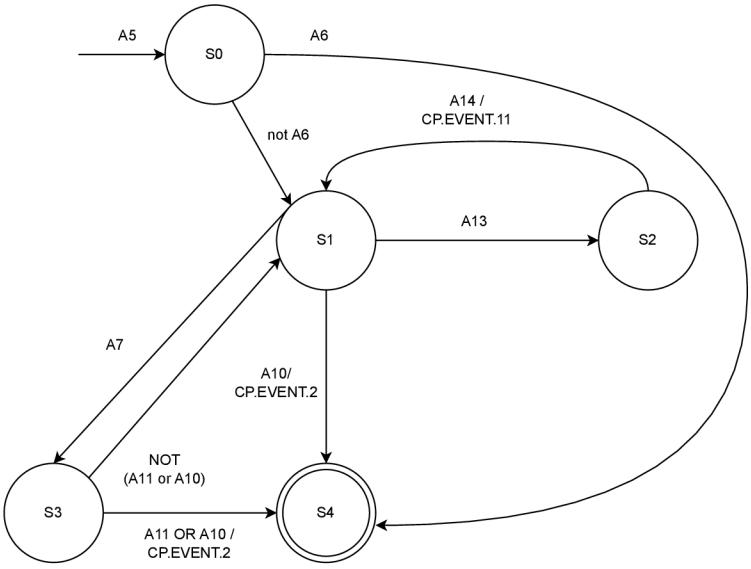
Pokud se to povede pak se jde do stavu, kdy je připraven na přeposílání komunikace. Jakmile obdrží data od informačního systému, jde do stavu, kdy komunikuje. Jakmile tato data přepošle a obdrží na ně odpověď modulu, kterou pošle zpět informačnímu systému, vrací se zpět do stavu připraven.

Jakmile dojde k načtení nové konfigurace validačním vláknem, tak při první následující návštěvě stavu připraven přechází vlákno do stavu, kdy se musí znovu autentizovat i autorizovat. Pokud se mu to povede úspěšně, pak se vrací do stavu připraven a může dál komunikovat s klientským programem. Pokud se nezadaří, pak se přesouvá do koncového stavu.

4.7.4 Klientský program

Jak bylo zmíněno klientský program jako jediný aktér není součástí serverového programu, ale jedná se o samostatný program, který komunikuje se serverovým programem. Jeho automat je na obrázku 4.4. Popis stavů je v tabulce 4.7.

Klientský program se po svém spuštění nachází v počátečním stavu. V případě úspěšného vytvoření klientského vlákna na straně serveru a neselhání autorizace se přesouvá do stavu, kdy čeká na požadavky od informačního systému. Ve chvíli, kdy přijde požadavek od informačního systému, pak se přesouvá do stavu, kdy požadavek přijal a pokračuje jeho



Obrázek 4.3: Diagram automatu klientského vlákna. Přechod značí událost, která nastala a případná reakce ostatních vláken na tuto událost (událost/reakce). Ze stavu `S3` je použita notace, kdy událost buď nastala nebo nenastala – stal se její opak. Jednotlivé stavy jsou popsány v tabulce 4.6.

Stav	**Popis**
`S0`	Created
`S1`	Ready
`S2`	Received from IS
`S3`	Sent to Server
`S4`	Finished

Tabulka 4.7: Popis jednotlivých stavů automatu klientského programu z obrázku 4.4.

přeposláním serveru. Nakonec se po přijetí a přeposlání odpovědi modulu vrací do stavu připraven.

Pokud ve stavu připraven dojde k uzavření programu či spojení, pak se přesouvá program do koncového stavu.

4.8 Logování

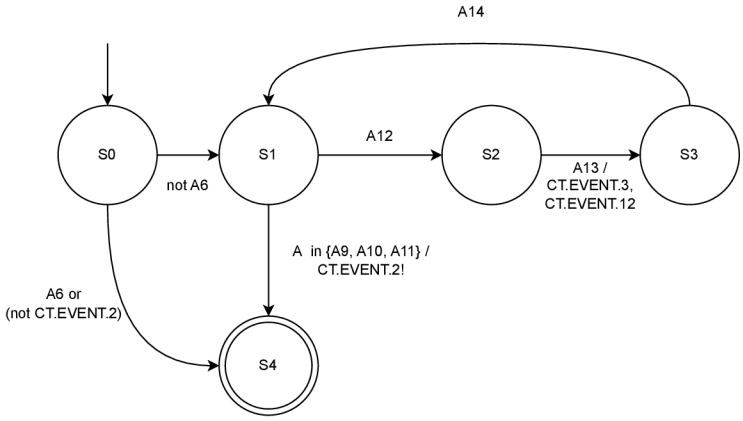
V rámci klientského i serverového programu je třeba zavést logování událostí na standardní výstup. O toto logování se stará statická třída `Logger`, která je přítomna na obou stranách. Jeden záznam v logu se zapisuje v následujcím formátu:

```
[DATETIME][TYPE][THREAD] message
```

Jako příklad si uveďme záznam o startu hlavního vlákna na straně (záznam je typu `info`):

```
[2023-04-14T08:08:17][info][mainThread] Started
```

V rámci logování rozlišujeme několik typů záznamů. Názvy všech těchto typů jsou uloženy jako konstanty ve statické třídě `LoggerType`. Jedná se o tyto typy:



Obrázek 4.4: Diagram automatu klientského programu. Přechod značí událost, která nastala a případná reakce ostatních vláken na tuto událost (událost/reakce). Z počátečního stavu se lze dostat koncového stavu i v případě, že se nepodaří událost `CT.EVENT.2`. Opuštění počátečního stavu je možné pokud nedojde k události `A6`, tj. stane se její opak. Ze stavu `S1` se lze více událostmi dostat do koncového stavu, to je znázorněno tak, že pokud událost patří do množiny událostí, pak dojde k přechodu do koncového stavu. Jednotlivé stavy jsou popsány v tabulce 4.7.

- **INFO** – Informace o běžné události jako je vytvoření vlákna, načtení nové konfigurace tenantů, přijetí či odeslání komunikace atp.
- **WARNING** – Varování o neočekávané události. Tato událost může vést i k zastavení vlákna atp.
- **EXCEPTION** – Odchycená výjimka, v daném záznamu je i její název atp.

Na straně serveru jsou tři možná vlákna. Jejich názvy jsou konstanty uloženy ve statické třídě `LoggerThread`. Jedná se o tyto názvy:

- `mainThread` – hlavní vlákno,
- `validationThread` – validační vlákno,
- `clientThread` – klientské vlákno.

Obdobně jsou uložena vlákna i pro klientský program:

- `mainThread` – hlavní vlákno,
- `connectionThread` – vlákno spojení.

Pro klientská vlákna v serverovém programu i pro vlákna spojení v klientském programu, které jsou v programu více než jedenkrát je ve třídě `Logger` zaveden mechanismus číslování vláken.

Níže vidíme začátek šestého klientského vlákna. Řetězec „pomlčka nula“ za číslem vlákna značí, že se jedná o hlavní vlákno.

```
[2023-04-14T08:09:58][info][clientThread.6-0] Started
```

Dále vidíme příklad šestého klientského vlákna:

```
[2023-04-14T08:09:58][info][clientThread.6-1] Started
```

K logování událostí typu informace a varování slouží statická metoda `log`. Metoda jako první argument dostane typ záznamu, dále dostane název vlákna, pak samotnou zprávu. Dále může nepovinně dostat celočíselný identifikátor vlákna a celočíselný příznak zda se jedná o podvlákno (nebo jinak pomocné vlákno). Ve výpisu 4.7 vidíme příklad zavolání této metody hlavním klientským vláknem při vlastním začátku. Poslední parametr je vynechán.

```
Logger.log(LoggerType.INFO, LoggerThread.CLIENT, "Started", clientId);
```

Výpis 4.7: Kód zalogování startu klientského hlavního vlákna.

Ve výpisu 4.8 vidíme zavolání metody při startu pomocného klientského vlákna. Oproti výpisu 4.7 je zde navíc i zadáno, že se jedná o podvlákno.

```
Logger.log(LoggerType.INFO, LoggerThread.CLIENT, "Started", clientId, 1);
```

Výpis 4.8: Kód zalogování startu klientského pomocného vlákna

4.9 Implementace jednotlivých komponent

Tato kapitola se převážně věnovala programům `secmux_server` a `secmux_client`. V této podkapitole se budeme věnovat implementaci pomocných programů pro spouštění různých konzolových nástrojů. Všechny tyto pomocné programy jsou implementovány v jazyce Python. To je zejména z důvodu možného snadného uzpůsobení při nasazení. Všechny tyto programy byly navrženy v podkapitole 3.4.

4.9.1 Spouštěč vzdálených programů

Spouštěč vzdálených programů je REST API napsané v jazyce Python, které přijímá požadavky na spouštění konzolových aplikací, spouští je a vrací jejich výstupy. Toto REST API je napsané ve frameworku Flask[5]. Celé REST API se skládá z jednoho bodu `exec`, který je dostupný pomocí metody `POST`.

```
@APP.route('/exec', methods=['POST'])
def exec_endpoint():
   files = request.files

   if len(files) < 1:
      return exec_no_files(request.json)
   else:
      return exec_with_files(files)
```

Výpis 4.9: Ukázka obsluhy bodu `exec` ve Flasku.

Ve výpisu 4.9 je zdrojový kód obsluhy tohoto bodu. Funkce `exec_endppoint` se zavolá ve chvíli, kdy je zavolán bod `exec`. V případě, že požadavek na tento bod obsahuje soubory, pak je dále obsloužen funkcí `exec_with_files`, které se předá seznam těchto souborů jako parametr. Pokud naopak žádné soubory přijaty nebyly, pak je obsloužen funkcí

[5] https://flask.palletsprojects.com/en/2.2.x/

`exec_no_files`. Této funkci je předán JSON z těla požadavku, který je převedený na struktury jazyka Python (slovník či seznam).

V případě funkce `exec_no_files` je nejdříve zkontrolováno zda převedený JSON požadavek je typu slovník a dále zda obsahuje povinné klíče `program` a `args`. V případě `args` se rovněž ještě kontroluje zda se jedná o seznam. Pokud některá z uvedených kontrol selže, pak se vrací chybové hlášení.

Dále je zavolána funkce `is_allowed_program`, která dostane jako argumenty volání název programu ke spuštění a jeho argumenty příkazové řádky. Funkce vrací pravdu, jestliže daný program je povolen v souboru `allowed_programs.txt` a případně také zda zadané argumenty odpovídají předpisu v tomto souboru. Pokud kontrola selže, tak je vráceno chybové hlášení.

Dále je vytvořen pracovní adresář `tmp_workdir_X`, kde X je přidělené číslo. Název adresáře je vygenerován funkcí `get_workdir_name`. V tomto adresáři bude požadovaný program spuštěn. Program je spuštěn pomocí modulu `subprocess` a jeho funkce `run`. Veškeré výstupy jsou vloženy do JSON objektu zobrazeného ve výpisu 3.6. Tento JSON je vrácen, ale ještě předtím je smazán pracovní adresář, popř. i se vším, co v něm spouštěný program zanechal.

V případě spouštění se soubory pomocí funkce `exec_with_files` je nejdřív načten obsah souboru `json` a převeden do Python struktur. Následuje obdobná kontrola jako u funkce `exec_no_files` – povinné klíče a jestli je program povolen. Poté je stejným způsobem vytvořen pracovní adresář, ve kterém jsou navíc vytvořeny složky `input` a `output`.

Dále jsou všechny soubory kromě souboru `json` uloženy do pracovního adresáře, předpokládá se, že názvy souborů začínají řetězcem `input/`. Poté je program spuštěn obdobným způsobem jako u předchozí funkce. Výsledky jsou uloženy do souborů ve složce `output`. Celý tento adresář je zabalen do archívu `tar`. Tento archív je vrácen. Po zavolání příkazu `return` se ještě zavolá funkce, která smaže kompletní pracovní adresář.

4.9.2 Falešná aplikace

Falešná aplikace se tváří jako konzolový program, který nahrazuje. Jejím hlavní úkolem je předat data od uživatele do spouštěče vzdálených programů. Opět tuto aplikaci lze rozdělit na dva druhy – pracující nebo nepracující se soubory. Program je implementovaný v jazyce Python, jelikož se předpokládá, že bude před každým nasazením ke klientovi poupraven a na rozdíl od jazyku C# jej není nutné kompilovat (viz podkapitola 2.3.5).

V případě prvního druhu, který nepracuje se soubory program vytvoří slovník se dvěma klíči – `program` a `args`. Do druhého klíče `args` se vloží všechny přijaté argumenty příkazové řádky. Do prvního klíče je více možností, co vložit. Nejjednodušší možností je vložit zde napevno textový řetězec s názvem požadovaného programu či prostě `argv[0]`, který obsahuje název spouštěného programu. Zde jsou dvě možnosti, buď program přejmenovat na požadovaný název nebo pomocí symlinku program provázat s požadovaným názvem s spouštět jej tímto způsobem.

Dále je zavolán příslušný bod API `exec` pomocí modulu `requests` a funkce `post`. Tato funkce vrátí výstup tohoto bodu. Pak stačí vypsat jednotlivé položky z vraceného slovníku.

U spuštění druhého druhu, tedy se soubory je třeba vytvořený slovník s klíči `program` a `args` vložit do souboru `json`, který bude prvním odeslaným souborem. Dále je třeba projít všechny argumenty a všechny, které začínají řetězcem `input` považovat za soubory a ty přidat mezi soubory, které budou odeslány. Bod se opět volá pomocí modulu `requests` a funkce `post`. Přijatý archív je třeba rozbalit do adresáře `output` – program předpokládá,

že existuje. Dále se vypíši výsledky z tohoto adresáře (standardní výstup a standardní chybový výstup). Nakonec je třeba vrátit stejný návratový kód jako spouštěná aplikace.

```
PORT = 5556
# ...
res = requests.post("http://localhost:"+str(PORT)+"/exec", json=d)
# ...
```

Výpis 4.10: Ukázka volání bodu `exec` pomocí modulu `requests`.

Ve výpisu 4.10 je část kódu, ve kterém se volá příslušný bod API. Kód začíná nastavením konstanty na číslo portu, na kterém poslouchá program `secmux_client`. Po vytvoření slovníku s informacemi o požadovaném spouštěném programu dojde k samotnému volání. Předpokládá se, že klientský program běží v rámci stejného počítače. Zavolá se v rámci `localhost` s příslušným portem a názvem daného bodu. Jako parametr `json` se vloží daný slovník, který je v proměnné `d`. Toto je volání bez souborů. V případě volání se soubory se použije parametr volání `files`, do které je vložen seznam obsahující soubory.

4.9.3 Proxy aplikace pro standardní vstup-výstup

Proxy aplikace pro standardní vstup-výstup je program, který čte svůj standardní vstup a veškerá data v bajtech odesílá do `secmux` infrastruktury. Poté prezentuje informačnímu systému výsledky vzdáleně spuštěné aplikace. Tento program je implementován v jazyce Python – z podobných důvodů jako falešná aplikace (viz podkapitola 4.9.2).

Komunikace zajišťuje třída `socket` ze stejnojmenného modulu. Napojuje se na `secmux` infrastrukturu přes `secmux_client`. Program nejdříve přečte celý obsah svého standardního vstupu:

```
data = sys.stdin.buffer.read()
```

Uvedený řádek kódu čte data přímo v bajtech a není třeba dělat žádné konverze. Dále také přečte kompletní obsah standardního vstupu až do jeho uzavření. Poté celý obsah odešle. Dále program odešle symbol EOF (viz tabulka 3.1). Dále se připraví proměnné na přijatá data. Proměnné pro standardní výstup a standardní chybový výstup se nastaví na prázdný řetězec bajtů, podobně jako na následujícím řádku:

```
stdout = stderr = b''
```

Poté se vstupuje do nekonečného cyklu, který bude ukončen příkazem `break`. Na začátku kroku cyklu se přijme z vytvořeného socketu přesně 11 bajtů, což je délka řídící sekvence (viz tabulka 3.1). Tento stažený řetězec se dekóduje podle `UTF-8` na textový řetězec. Nyní se zjistí jaká data budou tuto kontrolní sekvenci následovat. Toto zjištění se provádí pomocí metody `find` nad textovými řetězci. Vytvoří se tři příznaky: `isStdout`, `isStderr` a `isRetCode`. Do každého tohoto příznaku je přiřazena pravdivostní hodnota tohoto výrazu:

```
control.find(R) > -1
```

Proměnná `R` obsahuje první tři znaky dané sekvence (např. pro standardní výstup `<[o` nebo pro standartní chybový výstup `<[e`). Metoda vrací pozici vyhledávaného podřetězce v daném řetězci nebo číslo -1 pokud se daný podřetězec v řetězci nenachází. Dále jsou pomocí metody `replace` nad textovými řetězci z kontrolní sekvence odstraněny všechny nečíselné znaky – voláním této metody se všemi možnými začátky i konci kontrolní sekvence. Výsledný textový řetězec by měl obsahovat pouze celé číslo – program se pokusí o jeho přetypování, které pokud selže pak je program ukončen.

Pokud se jedná podle kontrolní sekvence o návratový kód, pak je uložen do proměnné `returnCode` a cyklus je ukončen. Pokud se jedná o některý standardní výstup, pak se vstupuje do smyčky, která běží dokud není přijat odpovídající počet dat z kontrolní sekvence. Přijatá data jsou vložena do odpovídající proměnné připojením za její stavající obsah. Poté cyklus pokračuje dalším krokem.

Po skončení cyklu proměnné `stdout` i `stderr` dekódují přes znakovou sadu `UTF-8`. První je vypsána na standardní výstup a druhá je vypsána na standardní chybový výstup. Nakonec je vrácen obdržený návratový kód.

4.10 Orchestrace modulů

Ve složce `kubernetes` se nacházejí různé konfigurace pro orchestraci modulů. Základním souborem pro naši infrastrukturu je soubor s konfigurací běhu kontejneru pro program `secmux_server`, tento soubor se jmenuje `secmux.yaml`. Jeho spuštění je popsáno na konci podkapitoly 4.2.

V uvedeném souboru je nakonfigurována služba typu `LoadBalancer`, která je pojmenována `secmux-service`. Tento typ služby byl zvolen, aby byla tato služba dostupná zvenčí clusteru. Tato služa má otevřený port 6000 a pracuje s jedinou aplikací zvanou `secmux-server`, která se skládá z jednoho kontejneru `secmux-server`, který pracuje v rámci nasazení `secmux-deployment`.

Jelikož lze předpokládat možnost přidání nového tenanta za běhu programu a z důvodu toho, že adresářová struktura konfigurace tenantů obsahuje i soubory s účtováním, je třeba vytvořit persistentní a snadno přístupné úložiště. Toto úložiště je nakofigurováno v souboru `secmux-volume.yaml`. Toto úložiště je vytvořeno přímo v uzlu. V souboru `secmux.yaml` je poté přímo nastaveno napojení (anglicky *mount*) tohoto úložiště na složku `client_certificates` v serverovém kontejneru. V případě použití `minikube` se lze k úložišti dostat příkazem:

```
minikube ssh
```

Do tohoto úložiště si pak příkazem `mount` lze napojit libovolnou složku z hostitelského počítače.

Při vytváření nového tenanta s vlastním kontejnerem jsou nejdůležitější tyto údaje:

- název kontejneru,
- název služby,
- otevřený port služby a správné provázání s portem kontejneru.

Jako první je třeba mít název kontejneru, pokud se jedná o vlastní kontejner, pak je nutné nastavit `ImagePullPolicy` na nikdy (anglicky *Never*), aby se Kubernetes nesnažilo stahovat daný kontejner z repozitáře. Dále je nutný název služby, který je třeba zapsat nejen do konfigurace pro Kubernetes, ale i do konfigurace daného tenanta jako jeho hostitelské jméno, protože pomocí něj bude serverový program ke kontejneru tohoto tenanta přistupovat. Dále je nutné mít v obou zmíněných souborech zapsaný i port služby (který navíc musí být správně provázán s portem kontejneru). Tento port je stejně jako název služby nutný pro komunikaci v rámci Kubernetes clusteru.

Narozdíl od služby `secmux` bude služba modulu nastavena na typ `ClusterIP`, protože modul by měl být dostupný pouze uvnitř clusteru. Ke službě tedy bude přístup pouze přes serverový program.

Kapitola 5

Ověření funkcionality

Tato podkapitola se zabývá ověřením funkcionality vytvořené infrastruktury jejím testováním. V první podkapitole 5.1 jsou představeny základní testy, které otestují základní vlasnosti – zda programy spolu dokáží komunikovat či zda jsou tenanti izolováni a nevidí si vzájemně do svých dat. V další podkapitole 5.2 jsou otestováni aktéři z podkapitoly 4.7 na základě kritéria pokrytí všech párů hran. Zde je též otestována obsluha více klientů najednou. V podkapitole 5.3 se zaměříme na testy reakcí, testy v této podkapitole budou ověřovat zejména výstupy v rámci logů či souborů účtování. Na závěr v podkapitole 5.4 se zaměříme na testy spouštění konzolových aplikací, ať už těch, které pracují se svým standardním vstupem, tak i těch, co pracují se svými parametry a odeslanými soubory.

5.1 Základní testy

Základní testy zkoumají některé základní vlastnosti výsledného systému. Každý test se skládá z alespoň jedné Redis databáze, která obslouží požadavky od alespoň jednoho jednoduchého programu reprezentující informační systém, tento program je napsán v jazyce Python. Komunikace je zajištěna prostřednictvím infrastruktury `secmux`, kdy informační systém komunikuje s programem `secmux_client` a Redis databáze je modul běžící na serverové straně jako modul dostupný prostřednictvím program `secmux_server`. Při testech se používají operace `set` a `get`.

V tabulce 5.1 je přehled jednotlivých testů. Test se vždy skládá z alespoň jednoho tenanta, který prostřednictvím klientů C komunikuje přes `secmux` infrastrukturu s Redis databází. Dohromady je tedy v každém testu $T \cdot C$ klientů.

Každý test má obdobný průběh. Hlavním souborem testu je vždy skript `run_testX.sh`, kde `X` je číslo příslušného testu. Před spuštěním takového skriptu je nutné provést dvě věci:

- nainstalovat program `secmux_client` na počítač, aby jej bylo možné spustit příkazem `client`,

	TEST1	TEST2	TEST3	TEST4
Tenanti T	1	1	2	1
Klienti C	1	2	1	1

Tabulka 5.1: Základní testy s počty tenantů a jejich klientů.

- spustit program `secmux_server` s příslušnou konfigurací tenantů, ta je vždy k dispozici v podadresáři `server` daného testu i se spouštěcím i s uklízecím skriptem. Spouštěcí skript též rozjede i požadované Docker kontejnery Redis databáze.

Spuštení také předpokládá prázdnou Redis databázi. Poté již lze spustit testovací skript `run_testX.sh`. Skript vždy nejdříve spustí všechny instance programu `secmux_client` s certifikáty, které jsou připraveny v adresáři daného testu. Poté se skript uspí na 2 vteřiny, aby se všechny instance programu stihly napojit na serverovou stranu. Poté se spustí testovací programy napsané v jazyce Python (kromě testu `TEST4`), které vyzkouší určité operace nad danou Redis databází. Každý tento Python program obsahuje i ověření pomocí příkazů `assert`. Pokud projdou, pak se na obrazovce objeví zpráva, že test prošel.

Nyní si popíšeme jednotlivé testy:

`TEST1` První test se skládá pouze z jednoho tenanta, který se napojí na Redis databázi přes jednu instanci klientského programu. Test si klade za cíl jednoduché ověření komunikace. Testovací informační systém nejdřív vytvoří v databázi klíč `k1` s hodnotou 42, ten se posléze pokusí opět získat a hodnotu ověří příkazem `assert`. Dále se pokusí získat obsah nenastaveného klíče pod názvem `unknown`, zde ověří vrácení hodnoty `None`.

`TEST2` Druhý test se skládá ze dvou instancí programu `secmux_client` pro jednoho tenanta. Cílem testu je ověřit, že i při dvou připojeních na jeden certifikát, se pořád pracuje se stejnými daty v jedné databázi. První testovací informační systém vytvoří v databázi klíč `k1` s hodnotou 42 a ověří jeho hodnotu, poté zkusí získat neexistují klíč `k0`. Druhý následně spuštěný program nejdříve nastaví a ověří v databázi klíč `k2`. Poté zkusí získat klíč `k1`, který by měl v databázi být po spuštění předchozího programu. Zde ověří, že klíč obsahuje hodnotu 42.

`TEST3` Další test se skládá ze dvou tenantů, kde každý používá jednu svou instanci programu `secmux_client`. Hlavní cílem testu je ověření izolovanosti dvou tenantů, tj. že si neuvidí navzájem na svá vlastní data. Oba testovací informační systémy jsou velmi podobné informačním systémů z druhého testu. První informační systém vytvoří a získá zpět klíč `k31`, dále se pokusí přistoupit k neexistujícímu klíči `k30`. Druhý program vytvoří a získá zpět klíč `k32`. Dále se pokusí získat klíč `k31`, který by však pro něj neměl existovat. Takže se ověří, že vrácená hodnota je `None`.

`TEST4` Poslední test na rozdíl od předchozích testů nepracuje s žádným testovacím informačním systémem. Test sice obsahuje jednoho tenanta, ale tento tenant je držitelem neznámého certifikátu. Testovací skript se pokusí s tímto neexistujícím certifikátem spustit klientský program. Očekává se, že klientský program se ihned vypne a na jeho standardním výstupu bude hlášení o tom, že se nepodařilo autentizovat. Toto se také ověří.

5.2 Testy aktérů

Tato podkapitola dokumentuje testy aktérů z podkapitoly 4.7. Cílem těchto testů je pokrýt jednotlivé aktéry podle testovacího kritéria všech párů hran (anglicky *Edge-Pair Coverage*). To znamená, že všechny výsledné testovací scénáře by měly pokrýt všechny syntakticky dosažitelné páry hran.

ID	Páry hran	Pokryto
1	A2	MT.TR.1
2	A3	MT.TR.2

ID	Páry hran	Pokryto
3	A4, A5	MT.TR.3, MT.TR.4
4	A5, A5	MT.TR.4

Tabulka 5.2: Páry hran diagramu automatu hlavního vlákna z obrázku 4.1 a jejich pokrytí testovacími scénaři z tabulky 5.6. Identifikátor ID je označení daného páru, který je odkazován z tabulky výsledných scénářů hlavního vlákna ve sloupci **Páry hran**. V dalším sloupci je onen pár hran (cesta maximální délky 2, popř. délky 1, pokud hrana směřuje do koncového stavu). Poslední sloupec obsahuje identifikátor testovacího požadavku, který tento pár pokrývá. Například první řádek obsahuje jednu hranu A2 z počátečního stavu do koncového stavu, která je pokryta testovacím požadavkem MT.TR.1.

ID	Páry hran	Pokryto
1	A15, A7	VT.TR.1
2	A15, A8	VT.TR.2
3	A8, A8	VT.TR.2
4	A8, A7	VT.TR.2

ID	Páry hran	Pokryto
5	A7, A15	VT.TR.1, VT.TR.2
6	A15, A8	VT.TR.2

Tabulka 5.3: Páry hran diagramu automatu validačního vlákna z obrázku 4.2 a jejich pokrytí testovacími požadavky z tabulky 5.7.

ID	Páry hran	Pokryto
1	A6	CT.TR.1
2	not A6, A10	CT.TR.6
3	not A6, A13	CT.TR.2, CT.TR.4
4	not A6, A7	CT.TR.5
5	A7, A11 or A10	CT.TR.5
6	A13, A14	CT.TR.2, CT.TR.4

ID	Páry hran	Pokryto
7	A14, A13	CT.TR.2
8	A14, A10	CT.TR.4
9	A14, A7	CT.TR.4
10	A7, not A11 & not A10	CT.TR.4
11	not A11 & not A10, A13	CT.TR.4
12	not A11 & not A10, A10	CT.TR.3

Tabulka 5.4: Páry hran diagramu automatu klientského vlákna z obrázku 4.3 a jejich pokrytí testovacími požadavky z tabulky 5.8.

ID	Páry hran	Pokryto
1	A6	CP.TR.1
2	not A6, A12	CP.TR.2
3	not A6, end	CP.TR.3
4	A12, A13	CP.TR.2

ID	Páry hran	Pokryto
5	A13, A14	CP.TR.2
6	A14, A12	CP.TR.2
7	A14, end	CP.TR.4, CP.TR.5

Tabulka 5.5: Páry hran diagramu automatu klientského programu z obrázku 4.4 a jejich pokrytí testovacími požadavky z tabulky 5.9. Hrana zvaná end reprezentuje hranu ze stavu S1 do koncového stavu S2.

ID	Hrany	Páry hran	Popis
MT.TR.1	A2	1	Neexistující soubor certifikátu serveru.
MT.TR.2	A3	2	Certifikát serveru není v korektním formátu.
MT.TR.3	A4, A5	3	Korektní spuštění s připojením 1 klienta.
MT.TR.4	A4, A5, A5	3, 4	Korektní spuštění s připojením 2 klientů.

Tabulka 5.6: Testovací požadavky na automat hlavního vlákna z obrázku 4.1, pokryté páry hran z tabulky 5.2 a slovní popis tohoto testovacího požadavku. První sloupec tabulky představuje identifikátor, se kterými pracuje tabulka 5.10. Například první řádek je označen jako MT.TR.1 a pokrývá pár hran číslo 1, tedy hranu A2, což znamená spuštění serveru s neexistujícím souborem certifikátu pro tento program.

ID	Hrany	Páry hran	Popis
VT.TR.1	A15, A7, A15, A7	1, 5	2x načíst certifikáty
VT.TR.2	A15, A8, A8, A7, A15, A8	2, 3, 4, 5, 6	2x načíst certifikáty, složka obsahuje 2 nekorektní certifikáty.

Tabulka 5.7: Testovací požadavky na automat validačního vlákna z obrázku 4.2 pokrývající jednotlivé páry hran z tabulky 5.3.

V tabulkách 5.2 – 5.5 jsou dvojice hran (páry) hran jednotlivých automatů z podkapitoly 4.7 (obrázky 4.1 – 4.4) a jednotlivé hrany, které tvoří cestu o délce 1 z počátečního stavu do koncového stavu.

Dále v tabulkách 5.6 – 5.9 jsou testovací požadavky, které pokrývají všechny páry hran. Příslušnost daného páru hran k testovacímu požadavku je uvedena v tabulkách párů hran pro jednotlivé aktéry (tabulky 5.2 – 5.5). Testovací požadavek je vždy složen z identifkátoru, dále je v tabulce uvedena cesta automatem popsána posloupností hran. Tato cesta vždy začíná v počátečním stavu a končí v koncovém stavu. Dále je v tabulce uveden soupis jednotlivých párů hran, které daná cesta pokrývá, každý pár je zde označen svým číslem (identifikátorem), tj. každý pár má ve své tabulce uvedeny všechny testovací požadavky, které jej pokrývají a dané testovací požadavky mají ve své tabulce uvedeny jednotlivé páry hran, které pokrývají.

Konečně v tabulce 5.10 jsou uvedeny jednotlivé testovací scénáře a k nim jednotlivé testovací požadavky z předešlých tabulek, které pokrývají. Testovací požadavky jsou v této tabulce uspořádány podle aktérů. První dva testy testují pouze serverový program na chybové stavy. Zbývají třetí a čtvrtý test testují propojení mezi více klienty a jedním serverovým programem. Při testu každého klienta se nejdříve na serverový program napojí klientský program a poté se na přes tento klientský program napojí na infrastrukturu `secmux` program v jazyce Python, který představuje při testu informační systém. Daný program obsahuje i příkaz `assert`, na konci svého běhu poté vypisuje číslo klienta a zda test uspěl nebo neuspěl. Jednotlivé testovací informační systémy implementují testovací požadavky z tabulek 5.8 a 5.9. Testovací požadavek `CT.TR.1` z tabulky 5.8 a požavek `CP.TR.1` z tabulky 5.9 jako jediní nejsou implementováni pomocí testovacího informačního systému, jelikož k jejich otestování stačí spuštění klientského programu.

V případě třetího i čtvrtého testu se zkrátí doba načítání konfigurace tenantů na 20 vteřin (nastavením proměnné prostředí `REFRESH_TIME`).

ID	Hrany	Páry hran	Popis
`CT.TR.1`	A6	1	Platný certifikát, neplatná autorizace
`CT.TR.2`	not A6, A13, A14, A13, A14, A10	3, 6, 7	- Platný certifikát i autorizace, - 1. požadavek: OK, - 2. požadavek: OK, - uzavření klienta
`CT.TR.3`	not A6, A7, not A11 & not A10, A10	4, 10, 12	- Platný certifikát i autorizace, - načtení nové konfigurace tenantů, - uzavření klienta
`CT.TR.4`	not A6, A13, A14, A7, not A11 or not A10, A13, A14, A10	3, 6, 9, 10, 11, 6, 8	- Platný certifikát i autorizace, - 1. požadavek: OK, - načtena nová konfigurace tenantů, - 2. požadavek: OK
`CT.TR.5`	not A6, A7, (A11 or A10)	4, 5	- Platný certifikát i autorizace, - načtení nové konfigurace tenantů - odpojen na jejím základě.
`CT.TR.6`	not A6, A10	2	- Platný certifikát i autorizace, - uzavření klienta

Tabulka 5.8: Testovací požadavky na automat klientského vlákna z obrázku 4.3 pokrývající jednotlivé páry hran z tabulky 5.4.

ID	Hrany	Páry hran	Popis
`CP.TR.1`	A6	1	Neznámý certifikát
`CP.TR.2`	not A6, A12, A13, A14, A12, A13, A14, end	2, 4, 5, 6, 7	Klientský program pošle 2 požadavky
`CP.TR.3`	not A6, end	3	Připojení a následně propadnutí validace
`CP.TR.4`	not A6, A12, A13, A14, end	7	1 požadavek a následně propadnutí validace

Tabulka 5.9: Testovací požadavky na automat klientského programu z obrázku 4.4 pokrývající jednotlivé páry hran z tabulky 5.5.

ID	Hlavní vlákno	Validační vlákno	Klientské vlákno	Klientský program
TEST1	TR.1	–	–	–
TEST2	TR.2	–	–	–
TEST3	TR.3, TR.4	TR.1	TR.1, TR.2, TR.3	TR.1, TR.2
TEST4	TR.3, TR.4	TR.2	TR.4, TR.5, TR.6	TR.3, TR.4

Tabulka 5.10: Výsledné testovací scénáře, které pokrývají jednotlivé testovací požadavky z tabulek 5.6 – 5.9, čímž jsou pokryty všechny páry hran z tabulek 5.2 – 5.5. Každý test se skládá z identifikátoru testu a pokrytím jednotlivých požadavků rozdělených podle jednotlivých aktérů, kteří jsou uvedeni v jednotlivých sloupcích. Například první řádek tabulky reprezentuje test s identifikátorem TEST1, který pokrývá pouze jeden testovací požadavek a to požadavek s identifikátorem TR.1 (úplný identifikátor je MT.TR.1) z tabulky 5.6. Pro ostatní aktéry tento test žádný požadavek nepokrývá.

	TEST3			TEST4			
Číslo klienta	1	2	3	1	2	3	4
Klientské vlákno	TR.1	TR.2	TR.3	TR.4	TR.5	TR.6	-
Klientský program	TR.1	TR.2	-	-	TR.3	-	TR.4

Tabulka 5.11: Pokrytí testovacích požadavků z tabulek 5.8 a 5.9 jednotlivými klienty v testech TEST3 a TEST4. Například v testu TEST3 pokrývá klient s číslem 1 testovací požadavky CT.TR.1 a CP.TR.1.

Testy klientů využívají modul na straně serveru v podobě databáze Redis. Testovací informační systémy se napojují na tento modul přes modul Redis nebo případně přes modul `socket` – pro některé jednodušší testy.

Nyní si projdeme všechny čtyři testy aktérů, které pokrývají všechny páry hran:

`TEST1` První test zkoumá spuštění serverové programu s cestou k neexistujícímu souboru certifikátu. Po spuštění programu `secmux_server` se očekává, že na standardním výstupu programu bude hlášení, že soubor nebyl nalezen.

`TEST2` Další test testuje spuštění serverového programu s cestou k existujícímu souboru, který ovšem není platný certifikátem. U testu se použije prázdný soubor, který je serverovému programu předán jako certifikát. Očekává se ukončení programu a hlášení na standardním výstupu o tom, že certifikát není v korektním formátu.

`TEST3` Při třetím testu je serverový program spuštěn s konfigurací tenantů, která obsahuje tři korektní konfigurace tenantů, aby byl otestován testovací požadavek `VT.TR.1`. Jedna z těchto konfigurací však obsahuje nekorektní nastavení služby, což má způsobit chybu až při autorizaci. Dále se napojí na serverový program klienti `client1`, `client2` a `client2`. První klient `client1` využívá pouze klientského programu a očekává se, že kvůli autorizační chybě bude ihned odpojen. Další klient `client2` se po spuštění klientského programu úspěšně napojí svým informačním systémem, který v Redis databázi nastaví a získá zpět klíč. Poslední klient `client3` se napojí přes modul `socket` a uspí se na 20 vteřin dokud nedojde k načtení nové konfigurace certifikátů a poté se odpojí.

`TEST4` Poslední test začíná spuštěním serverového programu s konfigurací, která ke korektním konfiguracím obsahuje i dvě nekorektní, aby byl pokryt požadavek `VT.TR.2`. Krátce po spuštění serverového programu je smazán jeden konkrétní certifikát, aby mohly být otestovány požadavky, které předpokládají napojení a poté odpojení kvůli propadnutí validaci. Test se skládá ze čtyř klientů `client1`, `client2`, `client3` a `client4`. První klient `client1` předpokládá napojení na testovacího informačního systému na klientský program, kdy bude nastaven klíč v Redis databázi, dále je program uspán (aby byla načtena nová konfigurace) a poté je tento klíč získán úspěšně zpět z databáze. Další klient `client2` se přes `socket` napojí na klientský program a uspí se na 20 vteřin dokud mu nepropadne konfigurace. Poté se pokusí odeslat databázi zprávu, na kterou by neměl dostat žadnou odpověď. Třetí klient `client3` se napojí přes modul `socket` a hned se uzavře. Poslední klient `client4` nastaví v Redis databázi klíč, uspí se na 20 vteřin, aby mu propadla konfigurace a poté se pokusí získat tento klíč, což se mu nepovede, protože je vyvolána výjimka kvůli ukončení spojení ze strany Redis.

V tabulce 5.11 je pro oba testy `TEST3` a `TEST4` uvedeno jaké testovací požadavky daný klient pokrývá.

5.3 Testy reakcí

Testy reakcí jsou dva jednoduché testy, které spočívají v kontrole logů serverového programu `secmux_server`, modulu či souboru pro účtování. Jako modul se používá program `tcpLogger`, což je program napsaný v jazyce Python, který vše, co obdrží na svém otevřeném TCP portu vypíše na svůj standardní výstup. Na rozdíl od základních testů či

testů aktérů popsaných v podkapitolách 5.1 a 5.2 se zde očekává, že serverový program i modul běží v Kubernetes clusteru. Testovací skripty tedy spouští pouze `secmux_client` a očekávají jako parametr názvy příslušných podů serverového programu a zmiňovaného modulu.

Nyní si blíže popíšeme vytvořené testy v této části, obdobně jako v minulých podkapitolách budeme hovořit o testech `TEST1` a `TEST2`:

`TEST1` První test začíná spuštěním programu `secmux_client`. Dále pomocí programu `netcat` (zkráceně `nc`) odešle klientskému programu zprávu, která je 19 znaků dlouhá. Poté jsou načteny logy serverového programu a TCP logovacího modulu. Ze serverového programu je pomocí programu `tail` přečteno posledních 10 řádků. Poté pomocí programu `grep` se hledá řetězec s informací o přijetí dat velikosti 19 bajtů z SSL připojení. Pokud se podaří najít je výstup neprázdný, pokud nikoliv výstup je prázdný. Tato část testu tedy projde, jestliže je výstup programu `grep` neprázdný.

Z TCP modulu je načten pouze poslední řádek, ten by měl obsahovat odeslanou zprávu a číslo 19 (velikost zprávy). Obdobně jako v minulém případě se toto ověří pomocí programu `grep`.

Poslední částí je ověření zalogování dat v souboru účtování. Z kontejneru serverového programu je načten poslední řádek souboru se záznamem o poslední aktivitě. Tato aktivita by měla obsahovat číslovku 19, což je opět ověřeno pomocí programu `grep`. Aby tento test neselhal nesmí být otevřeno vícero spojení s tímto modulem zároveň, protože pak by došlo z zalogování větší zátěže.

`TEST2` Druhý test je jednodušší než první test. Spustí se klientský program s neznámým certifikátem. Klientský program je by měl být okamžitě vypnut. Poté je obdobně jako v prvním testu načteno posledních 5 řádků logu serverového programu a očekává se, že obsahují záznam o neúspěšném připojení klienta.

5.4 Testy konzolových programů

Tato podkapitola se zabývá jednoduchými testy spouštění konzolových programů v infrastruktuře `secmux`. Testuje se zejména správné doručení výstupů testovacích aplikací. Každý test se skládá z testovací aplikace, která běží v infrastruktuře, dále je zde použita buď příslušná komponenta nebo skript, který ji nahrazuje.

Nejdříve se podíváme na test STDIN konzolové aplikace. Jako aplikace běžící spouštěné v cloudu zde figuruje program, který čte svůj standardní vstup. Jakmile se vstup uzavře, tak program předpokládá, že hodnota na posledním řádku vstupu je číslo N. Poté tento program vytiskne na svůj standardní výstup N-krát větu:

```
Hello from stdout.
```

A obdobně na svůj chybový výstup vytiskne N-krát zase tuto větu:

```
Hello from stderr.
```

V obou případech tiskne tyto věty každou na zvláštní řádek. Očekává se, že STDIN program vytiskne tyto věty v odpovídajícím počtu na odpovídající vstupy. Testovací skript vkládá na standardní vstup programu proxy aplikace pro standardní vstup-výstup soubor o několika řádcích, na jehož posledním řádku je číslovka 10. Poté jsou přečteny oba vstupy programu a očekávají se odpovídající věty na správných místech.

Druhý test zkoumá konzolovou aplikaci, která přijímá argumenty příkazové řádky a soubory. Konzolová aplikace očekává v argumentech soubory, které jsou fyzicky uloženy ve složce `input`. Aplikace všechny tyto soubory překopíruje do složky `output` a na svůj standardní výstup vypíše kolik souborů zkopírovala. Program spouštěč vzdálených programů by měl všechny výstupy ve výstupním adresáři odeslat zpět. Testovací skript vytvoří ve složce `input` na lokálním počítači soubory, jejichž počet je zadán při spuštění skriptu. Soubory jsou textové. Skript všechny tyto soubory odešle přes infrastrukturu programu spouštěč vzdálených programů. Skript poté v navrácených souborech zkontroluje, že standardní vstup obsahoval stejný počet souborů jako bylo zadáno při spuštění, že návratová hodnota byla nula a také projde všechny přijaté soubory a ověří, že jsou totožné jako ty odeslané.

Kapitola 6

Demonstrační řešení

V rámci této kapitoly budeme diskutovat čtyři příklady využití vytvořené infrastruktury. Nejdříve si ukážeme dvě možnosti TCP modulů a to Redis databázi a mikroslužbu poskytující REST API ve spolupráci s další Mongo databází. Dále si ukážeme dvě konzolové aplikace. Nejdříve program `convert`, se kterým si budeme demonstrovat úpravy obrázků v rámci infrastruktury `secmux`. Dále konzolovou aplikaci `pdftotext`, která čte PDF dokument ze svého standardního vstupu a na svůj standardní vstup tiskne textový obsah tohoto dokumentu. Rovněž si ukážeme provoz takové aplikace v rámci vytvořené infrastruktury.

Nyní si shrňme jednotlivé demonstrace před tím, než si je blíže popíšeme:

- Redis databáze je demonstrace databáze, ke které je připojeno více klientů, kteří komunikují prostřednictvím zpráv poslaných přes databází (tzv. publish-subscribe protokol),
- Rest API představuje typický příklad TCP/IP rozhraní založené na službě HTTP,
- program `Convert` slouží jako příklad jednorázového spuštění konzolové aplikace, která čte data ze souboru a generuje výstup do souboru,
- program `Pdftotext` jako příklad aplikace se standadně vstupně/výstupním rozhraním.

6.1 Redis databáze

Redis[1] je Redis databáze typu klíč-hodnota. Nám poslouží jako příklad samostatného vyčleněného modulu. Redis databáze je již hojně využívána v testech (viz kapitola 5). Redis databáze je sestavena z Dockeru obrazu Redis[2]. V rámci Kubernetes je Redis nakonfigurována jako služba typu `ClusterIP` (viz podkapitoly 2.3.2 a 4.10). Databáze komunikuje uvnitř Kubernetes clusteru pomcí svého běžného portu 6379. Přístup zvenčí k databázi je zajištěn prostřednictvím infrastruktury `secmux` s nakonfigurovaným certifikátem.

K databázi jako takové lze přistoupit dvěma způsoby. Buď prostřednictvím programu `redis-cli`, což je konzolový program, který umožňuje jednoduchou komunikaci s Redis databází prostřednictvím textových příkazů. Dalším způsobem je využití API v některém programovacím jazyce. Testovací informační systémy z našich testů jsou implementovány

[1] https://redis.io
[2] https://hub.docker.com/_/redis

Metoda	Adresa	Popis
`GET`	`person`	Vrací JSON kolekci všech uložených osob.
`POST`	`person`	Přidá novou osobu.
`GET`	`person/<id>`	Vrací osobu s identifikátorem `<id>`.
`PUT`	`person/<id>`	Modifikuje osobu s identifikátorem `<id>`.
`DELETE`	`person/<id>`	Odstraní osobu s identifikátorem **<id>**.

Tabulka 6.1: Popis jednotlivých bodů demonstračního REST API.

v jazyce Python a komunikovaly s databází prostřednictvím API pro tento programovací jazyk.

Redis databáze kromě ukládání a čtení hodnot označených klíči umožňuje i další možnosti ukládání dat. Redis umožňuje komunikaci přes kanál, ke kterému přistupují dvě role – vydavatel (anglicky *Publisher*) a odběratel (anglicky *Subscriber*). Vydavatel do příslušného kanálu publikuje zprávy a odběratel je odebírá. V tomto spočívá i ukázka využití demonstračního řešení.

Ukázka se skládá z testovacího skriptu a dvou programů – `leader.py` a `follower.py`. Oba jsou napsány v jazyce Python. První program představuje vydavatele. Jako první v databázi nastaví pod klíčem `channel` hodnotu (náhodný vygenerovaný textový řetězec), která představuje název kanálu, do kterého bude publikovat. Tuto hodnotu také vypíše na svůj standardní výstup. Poté do kanálu pod tímto názvem postupně odešle pět náhodně vygenerovaných textových řetězců. Při odesláni každý řetězec opět vypíše na svůj výstup. Druhý program představuje odběratele. Tento program se spouští s menším zpožděním po publikovateli. Program jako první přečte hodnotu uloženou v databázi pod klíčem `channel`. Poté se přihlásí k tomuto kanálu a přečte z něj pět hodnot, poté se ukončí. Postupně vypíše na svůj standardní výstup v průběhu běhu: název kanálu a pět přečtených hodnot z příslušného kanálu v pořadí, jak je obdržel. Je zřejmé, že výstupy obou programů by měly být totožné, proto jsou také výstupy testovacím skriptem po jejich skončení srovnány.

6.2 RestAPI

Toto demonstrační řešení obsahuje REST API, které poskytuje body pro správu uložení jednoduché entity osoba (anglicky *person*) s atributy jméno `name`, přijmení `surname` a primární klíčem `_id`. REST API využívá k uložení osob NoSQL databázi Mongo. Tento modul se v rámci Kubernetes skládá ze dvou služeb. První je služba, která představuje samotné REST API a druhá služba, která se skládá z oné Mongo databáze. Serverový program komunikuje pouze s REST API, které komunikuje s Mongo databází. REST API je obdobně jako program CLI Worker API (viz podkapitola 4.9.1) implementováno v jazyce Python s využitím frameworku Flask. V tabulce 6.1 je krátký popis jednotlivých bodů API.

Komunikace s Mongo databází se nastavuje přímo v souboru konfigurace Kubernetes. Pro komunikaci s databází se používá jméno Kubernetes služby a výchozí Mongo port 27 017. REST API program název této služby obdrží jako proměnnou prostředí, která je nastavena právě v jeho Kubernetes konfiguraci.

Mongo databáze není vidět zvenčí cloudu. Dokud nebude nastavena v konfiguraci tenantů pro program `secmux_server`, tak k ní tento program nebude přistupovat. Takto lze vytvářet moduly složené z více služeb, které spolu navzájem kooperují, ale ven z cloudu již komunikuje pouze jedna z nich.

K tomuto API vznikl jednoduchý testovací skript, který demonstruje jeho funkčnost. Tento skript nejdříve vytvoří osobu, poté ji zkusí získat zpět z API. Pak této osobě změní křestní jméno a zkusí ji najít v kolekci všech osob. Na závěr tuto osobu odstraní a zkontroluje zda tuto osobu již nelze najít.

6.3 Aplikace convert

Aplikace `convert` je součást balíku ImageMagick[3]. Jedná se o program, který slouží k různým úpravám obrázků. V této demonstraci se zaměříme na změnu velikosti:

```
convert $INPUT_FILE -resize 240x240 $OUTPUT_FILE
```

Tato demonstrace jako vstup potřebuje vstupní soubor, dále parametry spuštění, které stanoví, co se má provést a také název výstupního souboru. Na základě těchto parametrů a vloženého souboru bude vygenerován změněný soubor na základě těchto parametrů spuštění.

V případě takového spuštění bude velikost vstupního obrázku změněna na 240×240. Výstup bude uložen do stanoveného výstupního obrázku. Konzolový program bude spouštěn prostřednictvím CLI Worker API (viz podkapitoly 3.4.3 a 4.9.1). Nejdříve je třeba tento program zprovoznit na straně API. Nejdříve je třeba jej nainstalovat do kontejneru tohoto API zmíněný balík ImageMagick:

```
apt-get update
apt-get install imagemagick
```

Dále je třeba povolit program `convert` v souboru `allowed_programs.txt`. Následně je třeba vygenerovat certifikát a nakonfigurovat nového tenanta, který bude mít přístup ke CLI Worker API.

Dále je třeba upravit falešnou aplikaci (viz podkapitoly 3.4.4 a 4.9.2) na míru pro program `convert`. Falešná aplikace bude pracovat se soubory a posílat je API. Jako soubor bude považovat každý argument začínající řetězcem `input/`. Výstupy se budou generovat do složky `output`. Jelikož aplikace pracuje se soubory, obdrží od API archív se soubory, kde některé soubory obsahují obsahy standardních výstupů spouštěné aplikace. Kromě odeslání souborů je důležité program správně zavolat, např. jak je znázorněno ve výpisu 6.1.

```
{
   "program": "convert",
   "args": ["input/file.jpg", "-resize", "240x240", "output/file.jpg"],
}
```

Výpis 6.1: Příklad volání aplikace `convert`.

6.4 Aplikace pdftotext

Program `pdftotext`[4] je program, který slouží na převod PDF dokumentu na čistý text. Dokument může být přijat přes standardní vstup a výstup může být vytištěn na standardní výstup. Následující spuštění pracuje se standardním vstupem a výstupem:

```
pdftotext - -
```

[3] https://imagemagick.org
[4] https://www.xpdfreader.com/pdftotext-man.html

Jako první je třeba tento program nainstalovat do kontejneru serverového programu:

```
apt-get update
apt-get install poppler-utils
```

Poté je třeba vygenerovat nový nový certifikát a nakonfigurovat tenanta. Konfigurace je ve výpisu 6.2. Program bude spuštěn, jak bylo znázorněno výše.

```
 ---
name: pdftotext
type: cliService
program: pdftotext
args:
  - '-'
  - '-'
```

Výpis 6.2: Ukázka volání bodu `exec` pomocí modulu `requests`.

Nyní již stačí spustit klientský program s příslušným certifikátem a spustit proxy aplikaci pro standardní vstup-výstup (viz podkapitoly 3.4.5 a 4.9.3), ve kterém je třeba správně nastavit port. Program přečte ze svého standardního vstupu požadovaný dokument ve formátu PDF a ten odešle serverovému programu ukončené stanoveným řídícím znakem `EOF`. Serverový program veškerou přijatou komunikaci až po danou řídící sekvenci vloží na standardní vstup programu `pdftotext`. Poté serverový program ukončí standardní vstup tohoto programu a odešle zpět veškerý obsah z obou standardních výstupů, které proxy aplikace pro standardní vstup-výstup vypíše.

Kapitola 7

Závěr

Práce si kladla za cíl vytvořit multitenantní architekturu, ve které budou moduly orchestrovány pomocí nástroje Kubernetes. Vzniklá architektura umožňuje orchestrovat kontejnerizované moduly a poskytuje k nim přístup na základě certifikátu tenanta. Mezi tenantem a vyčleněným modulem funguje zabezpečený přenos přes SSL. Architektura se skládá z klientského programu, který běží na straně tenanta a serverového programu, který běží v multitenantním systému a zpřístupňuje požadované moduly.

Mimo kontejnerizované moduly lze v infrastruktuře spouštět i konzolové aplikace a ovládat je po síti. Aplikace, které zpracovávají svůj standardní vstup, se spuší přímo v kontejneru se serverovým programem. Aplikace, které přijímají argumenty příkazové řádky a soubory, pracují ve vlastním kontejneru, kde jsou spouštěny přes REST API nazvané spouštěč vzdálených programů.

Serverový program mimo přepínání modulů a zabezpečený přenos poskytuje ještě účtování, které může provozovatel cloudu používat k vyhodnocování zátěže jednotlivých tenantů.

Celá infrastruktura i jednotlivé komponenty byly otestovány. Pro serverový i klientský program byly navrženy testy, které splňují pokrytí všech párů hran pro jednotlivé aktéry, kteří vystupují v rámci těchto programů.

Bylo vytvořeno několik demostračních případů využití této infrastruktury. Jedná se o provoz databáze Redis, dále se jedná o provoz jednoduchého REST API s vlastní Mongo databází. Dále se jedná o spouštění konzolových aplikací na práci s obrázky a práce s dokumenty ve formátu PDF.

Existuje několik možných rozšíření práce do budoucnosti. Nyní certifikát tenanta umožňuje přístup pouze k jedné službě. V budoucnu lze diskutovat přístup k více službám na základě jednoho certifikátu. Dále lze hovořit o více certifikátech tenanta, kde každý certifikát bude poskytovat jinou úroveň oprávnění v daném modulu. Adresářová struktura je na tyto změny do budoucna připravena, je však třeba je navrhnout a doimplementovat.

Vytvořenou infrastrukturu lze využít ve spolupráci s jakýmkoliv systémem, jehož část se má vyčlenit do cloudového prostředí. Práce v mnoha případech umožňuje plynulé vyčlenění modulu s minimem změn na straně stávajícího systému.

Literatura

[1] Abozeid, S. *Traditional Deployment VS Virtualization VS Container* [online]. 2020 [cit. 2023-01-24]. Dostupné z: https://www.linkedin.com/pulse/traditional-deployment-vs-virtualization-container-shazly-abozeid/.

[2] Ben Kiki, O., Evans, C. a Net, I. döt. *YAML Ain't Markup Language (YAML™) version 1.2* [online]. 2021 [cit. 2023-04-05]. Dostupné z: https://yaml.org/spec/1.2.2/.

[3] Bigelow, S. J. *Multi-tenancy* [online]. 2022 [cit. 2023-01-12]. Dostupné z: https://www.techtarget.com/whatis/definition/multi-tenancy.

[4] Blinowski, G., Ojdowska, A. a Przybylek, A. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *IEEE access*. Piscataway: IEEE. 2022, sv. 10, s. 20357–20374. ISSN 2169-3536.

[5] *Use containers to Build, Share and Run your applications* [online]. [cit. 2023-01-18]. Dostupné z: https://www.docker.com/resources/what-container/.

[6] *Kubernetes VS Docker Swarm – What is the Difference?* [online]. 2022 [cit. 2023-01-23]. Dostupné z: https://www.freecodecamp.org/news/kubernetes-vs-docker-swarm-what-is-the-difference/.

[7] *What is virtualization?* [online]. [cit. 2023-01-17]. Dostupné z: https://www.ibm.com/topics/virtualization.

[8] Jeřábek, F. *Orchestrace modulů multitenantních systémů*. Brno, CZ, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/25189/.

[9] Jiménez Domingo, E., Lagares Lemos, A. a Gómez Berbís, J. M. Multitenancy: A new architecture for clouds. In: *Cloud Computing: Methodology, Systems, and Applications*. 2017, s. 261–275. ISBN 9781439856413.

[10] *Úvod do JSON* [online]. 2020 [cit. 2023-04-05]. Dostupné z: https://www.json.org/json-cz.html.

[11] *Kubernetes Service* [online]. [cit. 2023-04-17]. Dostupné z: https://kubernetes.io/docs/concepts/services-networking/service/.

[12] *Kubernetes Components* [online]. 2022 [cit. 2023-01-22]. Dostupné z: https://kubernetes.io/docs/concepts/overview/components/.

[13] *SslStream Class* [online]. [cit. 2023-04-08]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/api/system.net.security.sslstream?view=net-6.0.

[14] *A tour of the C# language* [online]. 2023 [cit. 2023-04-06]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/.

[15] *The Python Tutorial* [online]. 2023 [cit. 2023-04-06]. Dostupné z: https://docs.python.org/3/tutorial/.

[16] *What is container orchestration?* [online]. 2022 [cit. 2023-01-22]. Dostupné z: https://www.redhat.com/en/topics/containers/what-is-container-orchestration.

[17] *Introduction to Multi-Tenant Architecture* [online]. 2022 [cit. 2023-01-16]. Dostupné z: https://stratoflow.com/multitenancy-introduction.

[18] Tucakov, D. *OpenSSL Tutorial: How Do SSL Certificates, Private Keys, & CSRs Work?* [online]. 2018 [cit. 2023-01-20]. Dostupné z: https://phoenixnap.com/kb/openssl-tutorial-ssl-certificates-private-keys-csrs.

[19] Velte, A. T. *Cloud computing : a practical approach*. New York: McGraw-Hill, 2010. ISBN 978-0-07-162694-1.

Příloha A

Obsah odevzdaného CD

Odevzdané CD obsahuje tyto adresáře a soubory:

- `src_text` – adresář obsahující zdrojové soubory a obrázky této zprávy,
- `secmux` – adresář obsahující impelementaci infrastruktury, příklady a testy,
  - `executable` – adresář se spustitelnými soubory,
    - `server` – adresář se spustitelnými soubory programu `secmux_server`,
    - `client` – adresář se spustitelnými soubory programu `secmux_client`,
  - `certificate_scripts` – adresář obsahující skripty pro generování certifikátů,
  - `client` – adresář obsahující klientskou část,
    - `client` – adresář s soubory implementace klienta (viz podkapitola 4.1),
    - `client.sln` – soubor Visual Studio solution pro klientský program,
  - `examples` – adresář s příklady použití,
    - `STDINprogram` – adresář s ukázkovým programem, který zpracovává svůj standardní vstup,
    - `STDINproxy` – adresář s proxy aplikací pro standardní vstup-výstup (viz podkapitola 4.9.3),
    - `cli` – adresář se spouštečem vzdálených programů a několika ukázkami jeho použití,
      - `cliAPI` – adresář se spouštečem vzdálených programů,
      - `cliExample1` – adresář s jednoduchým příkladem `soucet`,
      - `cliExample2Convert` – adresář s příkladem `convert`,
      - `cliExample3TEST` – adresář s programem, který odešle do API soubory a spustí zde program `ls`,
      - `cliExample4TEST` – adresář s programem, který odešle API soubory a očekává, že je všechny obdrží zpět,
    - `pdfSTDIN` – adresář s demonstrační příklad s programem `pdftext`,
    - `redis-channel` – adresář s demonstračním příkladem použití Redis databáze v infrastruktuře,
    - `restAPI` – adresář s demonstrační REST API aplikací,
    - `restAPIdemo` – adresář s demonstračním použitím REST API,

    - `tcpListener` – adresář s demonstračním TCP programem `tcpListener`
    - `testSTDIN` – adresář s demonstračním STDIN programem,
  - `kubernetes` – adresář obsahující konfigurace Kubernetes clusteru,
    - `client_certificates` – adresářová struktura tenantů pro serverový program,
    - `cli_worker.yaml` – soubor s Kubernetes konfigurací pro spouštěč vzdálených programů,
    - `mongo-demo.yaml` – soubor s Kubernetes konfigurací pro Mongo databázi,
    - `personRestAPI.yaml` – soubor s Kubernetes konfigurací pro demonstrační REST API,
    - `redis-demo.yaml` – soubor s Kubernetes konfigurací pro Redis databázi,
    - `redis-demo1.yaml` – soubor s Kubernetes konfigurací pro Redis databázi,
    - `secmux-volume.yaml` – soubor s Kubernetes konfigurací pro persistentní úložiště pro serverový program,
    - `secmux.yaml` – soubor s Kubernetes konfigurací pro serverový program,
    - `tcplogger.yaml` – soubor s Kubernetes konfigurací pro TcpLogger,
    - `script.py` – skript pro přidání nového tenanta,
  - `server` – adresář obsahující serverovou část,
    - `apps` – adresář s STDIN aplikacemi, které serverový program spouští,
    - `server` – adresář s soubory implelementace serverového programu (viz podkapitola 4.1),
    - `Dockerfile` – soubor pro sestavení Docker obrazu serverového programu,
    - `server.sln` – soubor Visual Studio solution pro klientský program,
    - `certificate.pfx` – certifikát serveru pro demonstrační spouštění,
  - `tests` – adresář s testy infrastruktury,
    - `automata_tests` – adresář s testy pokrývající jednotlivé automaty aktérů (viz podkapitola 5.2),
    - `basic_tests` – adresář s základními testy (viz podkapitola 5.1),
    - `reaction` – adresář s testy reakcí (viz podkapitola 5.3),
- `README.md` – soubor s krátkým popisem projektu, sestavením atp. (anglicky),
- `manual.md` – manuál na zprovoznění infrastruktury na lokálním stroji,
- `container_start.sh` – skript na start kontejnerů při lokálním spuštění.