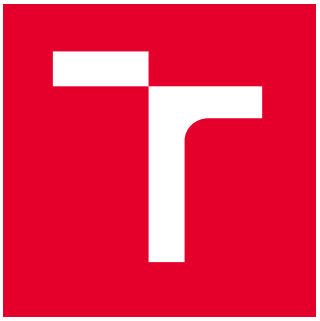


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

DEPARTMENT OF INTELLIGENT SYSTEMS

WEBOVÝ MANAŽER PRO HRANÍ DESKOVÝCH HER

WEB MANAGER FOR BOARD GAMES

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — **Petr Večeřa**

AUTHOR

VEDOUCÍ PRÁCE — **Ing. Jaroslav Rozman, Ph.D.**

SUPERVISOR

BRNO 2025

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

163930

Ústav:	Ústav inteligentních systémů (UITS)
Student:	**Večeřa Petr**
Program:	Informační technologie
Název:	**Webový manažer pro hraní deskových her**
Kategorie:	Web
Akademický rok:	2024/25

Zadání:

1. Nastudujte deskové hry a Manažer pro deskové hry, který vznikl na FIT VUT. Nastudujte v současnosti používané webové technologie.
2. Navrhněte úpravu Manažera tak, aby umožňoval plánování a spouštění turnajů a her přes webové rozhraní. Pokud bude nutné, navrhněte i úpravu koster programů pro jednotlivé hry.
3. Navržené webové rozhraní implementujte a dále implementujte i vzorové programy pro jednotlivé hry, které Manažer podporuje.
4. Všechny vytvořené programy otestujte.

Literatura:

- Čížek Tomáš, Manažer pro deskové hry s paralelním hraním turnajů, bakalářská práce, VUT FIT, 2014
- Kouřil Jan, Manažer stolních deskových her, bakalářská práce, VUT FIT, 2008

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:	**Rozman Jaroslav, Ing., Ph.D.**
Vedoucí ústavu:	Kočí Radek, Ing., Ph.D.
Datum zadání:	1.11.2024
Termín pro odevzdání:	14.5.2025
Datum schválení:	31.10.2024

Abstrakt

Práce pojednává o pokračování projektu výukové pomůcky na demonstraci a vizualizaci algoritmů používané pro umělou inteligenci při hraní deskových her. Vytvořený nápad z desktopové aplikace přesouvá do webového prostředí, tak, aby se umožnilo nástroj koncipovat jako souhrnnou pomůcku v předmětu i s přítomností učitele.

Abstract

This thesis is the continuation of the already executed idea about creation of study help tool for vizualization and demonstration of algorithms used for artificial intelligence during desk games matchmaking. The project is now pushed to the web realms to enable the tool to become a complete tool now with incorporation of the subject lecturer.

Klíčová slova

Docker, Webový Manažer, brain, kontejner, virtualizace, ASP.NET, C#

Keywords

Docker, Web Manager, brain, container, virtualzation, ASP.NET, C#

Citace

VEČEŘA, Petr. *Webový manažer pro hraní deskových her*. Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jaroslav Rozman, Ph.D.

Webový manažer pro hraní deskových her

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Jaroslava Rozmana Další informace mi poskytli Filip Orság a Roman Jašek. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Petr Večeřa
14. května 2025

Poděkování

Chci poděkovat vedoucímu p. Rozmanovi za vstřícnou a hladkou komunikaci v průběhu řešení bakalářské práce, zároveň chci tímto předat poděkování za inspiraci a zajímavé nápadu p. Orságovi a také p. Jaškovi za užitečné rady a tipy na koncepci systému jako celku

Obsah

1 Úvod **5**

2 Abstrakt **6**

3 Dosavadní podoba manažera **7**
3.1 Obecný princip fungování Manažera 7
3.1.1 Princip brainů 7
3.1.2 Forma turnajů 8
3.1.3 Hodnocení turnajů 9
3.2 První verze Manažera 9
3.3 Druhá verze Manažera 10
3.4 Architektura aktuálních manažerů 10
3.5 Změna architektury na webovou 11

4 Virtualizace brain procesů **13**
4.1 Koncepce virtualizace 13
4.2 Přehled dostupných metod virtualizace 14
4.2.1 Aplikační virtualizace 14
4.2.2 Virtualizace platformy operačního systému 14
4.3 Hypervisor 14
4.3.1 Hypervisor prvního typu 14
4.3.2 Hypervisor druhého typu 16
4.4 Obecný princip kontejnerizace virtuálních systémů 18
4.4.1 Popis prostředků pro podporu kontejnerů v systému Linux 19
4.5 Docker 20
4.5.1 Architektura Dockeru 21
4.5.2 Izolace Dockeru od hosta 22
4.5.3 Docker v dockeru 23
4.5.4 Docker vedle dockeru 24
4.5.5 Sysbox 24

5 Návrh architektury webové aplikace **25**
5.1 Funkční požadavky na webovou aplikaci 25
5.2 Vrstva přístupu dat 26
5.2.1 Databáze 26
5.2.2 Repozitář a Unit of work 27
5.3 Vrstva doménové logiky 28
5.3.1 Nasazování brainů do kontejnerů 28

5.3.2 Fasády . . . 29
5.3.3 Úlohy na pozadí . . . 30
5.4 Webová vrstva . . . 30
5.4.1 Uživatelské účty . . . 31
5.4.2 Realtime komunikace . . . 32

6 Realizace webového manažera **33**
6.1 Turnaje . . . 33
6.1.1 Vyřazovací systém na dvě porážky . . . 33
6.1.2 Řešení diskvalifikací . . . 33
6.1.3 Řešení lichých počtů . . . 34
6.1.4 Paralelní pořádání turnajů . . . 34
6.1.5 Plánovač turnajů . . . 35
6.2 Vytvoření docker obrazu . . . 36
6.3 Uživatelské rozhraní . . . 39
6.3.1 Přihlášení a správa účtu . . . 39
6.3.2 Správa Docker obrazu . . . 41
6.3.3 Pořádání turnajů . . . 42
6.3.4 Správa brainů uživatele . . . 45
6.3.5 Validace uživatelského vstupu . . . 47

7 Testování **49**
7.1 Seedování databáze . . . 49
7.2 Testování využití paměti . . . 49
7.3 Testování náročnosti na procesor . . . 52

8 Závěr **54**

Literatura **55**

A Protokol komunikace **57**

Seznam obrázků

3.1 Schéma komunikace manažera s brainy. Obrázek převzatý z [18] 8
3.2 Uživatelské rozhraní Manažera 1. Obrázek převzatý z [17] 9
3.3 Uživatelské rozhraní Manažera 2. Obrázek převzatý z [19] 10
3.4 Diagram používání dosavadní verze manažera v praxi 11
3.5 Diagram navrhované architektury webového manažera 12

4.1 Ilustrace systému s Hypervisorem prvního typu 15
4.2 Vrstvy v systému s hypervisorem prvního typu 16
4.3 Ilustrace systému s Hypervisorem druhého typu 17
4.4 Vrstvy v systému s hypervisorem prvního typu 18
4.5 Vrstvy v systému Linux s kontejnery 19
4.6 Zjednodušené schéma klient-server docker architektury 21
4.7 Schéma použití přístupu docker v dockeru pro docílení izolace případné hostitelské aplikace, která by potřebovala ovládat docker kontejnery nezávisle na hostitelském systému, kde privilegovaný kontejner běží. 23

5.1 Schéma databázového modelu tak, jak jsou mapované třídy vůči tabulkám v databázi 27
5.2 Zjednodušený diagram tříd v DAL vrstvě programu 28
5.3 Schéma fungování Blazor serveru v hybridním klient/server režimu 31

6.1 Schéma zpracování předávání řízení mezi vlákny při asynchronních operacích 38
6.2 Přihlašovací stránka webového manažera 40
6.3 Stránka s registrací u webového manažera 40
6.4 Stránka se správou uživatelského účtu 41
6.5 Stránka se správou Docker obrazu s možností sledovat reálný výstup konzole 42
6.6 Hláška signalizující úspěšné nasazení Docker obrazu 42
6.7 Stránka se seznamem turnajů 43
6.8 Stránka s detailem turnaje před spuštěním turnaje 43
6.9 Chybová hláška zamezující spuštění turnaje, pokud chybí Docker obraz . . 44
6.10 Zamčený detail turnaje v případě probíhajícího turnaje 44
6.11 Výsledky turnaje z pohledu organizující osoby 45
6.12 Stránka správy brainů a jejich verzí 46
6.13 Navigační panel, který vidí studenti s běžným uživatelským oprávněním . . 46
6.14 Zobrazení výsledků turnaje z pohledu uživatele 47
6.15 Výběr brainů uživatele v přehledu brainů, zobrazuje se vždy zvolené jméno brainu, který si uživatel nastavil 47
6.16 Upozornění na neplatné data u formuláře 48
6.17 Hláška upozorňující na neplatné přihlašovací údaje 48

7.1 Graf využití paměti při pořádání turnaje pro vybrané sety brainů - horní polovina 50
7.2 Graf využití paměti při pořádání turnaje pro vybrané sety brainů - spodní polovina 51
7.3 Graf využití procesoru při pořádání turnaje pro vybrané sety brainů - horní polovina 52
7.4 Graf využití procesoru při pořádání turnaje pro vybrané sety brainů - spodní polovina 53

Kapitola 1

Úvod

Umožnit studentům si prakticky vyzkoušet probíranou látku v předmětu má vždy své pozitivní stránky. Odborné předměty nabízí praktické seznámení například formou laboratorních cvičení. Většinou studenti mají možnost si prakticky látku vyzkoušet ve školním prostředí, ale pokud student onemocní, může o takovou možnost přijít. Pro předmět Základy umělé inteligence se nabízí možnost, jak si vyzkoušet možnost práci s umělou inteligencí, což je téma, které dnes rezonuje světem. Práce může být v podobě interaktivního nástroje, kde si student může prakticky vyzkoušet probírané algoritmy na přednáškách. Zároveň lze takto studenty motivovat k hledání nových a lepších řešení, které můžou být nad rámec probíraného učiva v předmětu. Samotná myšlenka nástroje se začala již realizovat několik let zpět v podobě počítačového programu, který mohl takto vytvořené algoritmy interaktivně ukazovat. Pojmout myšlenku a vytvořit takový nástroj bylo výzvou již několika mých předchůdců, kteří ukázali své možnosti, jak takovou pomůcku vytvořit. Dostal jsem za úkol jejich přínosy posunout o krok dál. Navrhnout takový systém, který bude sloužit jako celek jak pro učitele, tak i studenty bez rozdílu jejich zkušeností. Dnešní pomůcky hodlám posunout o krok dál tak, aby byly přínosnější a reflektovaly potřeby dnešních nároků předmětu a potenciálně i dalších oblastí mimo studijní dosah daleko za hranice Fakulty Informačních technologií. Dokument bude postupně rozkrývat momentální stav pomůcek. Ukáže jejich nedostatky a jejich přínos. Dále rozeberu možnosti úprav pro dnešní dobu, jak lze postupovat a co všechno pro to bude potřeba. Při popisu vymyšleného řešení budu klást za cíl, abych problematiku dostatečně srozumitelně popsal takovým způsobem, aby bylo zřetelné, proč jsem se takto rozhodl. V závěru ukážu praktické testování nároků, jestli je navržené a vytvořené řešení použitelné v praxi a zhodnotím závěry a východiska práce.

Kapitola 2

Abstrakt

Práce pojednává o pokračování projektu výukové pomůcky na demonstraci a vizualizaci algoritmů používané pro umělou inteligenci při hraní deskový her. Vytvořený nápad z desktopové aplikace přesouvá do webového prostředí, tak, aby se umožnilo nástroj koncipovat jako souhrnnou pomůcku v předmětu i s přítomností učitele.

Kapitola 3

Dosavadní podoba manažera

Před tím, než se zaměřím na samotné jádro této navazující práce, chci ještě před tím nastínit současný stav Manažera, jeho obecné fungování a rozlišení verzí. Tato práce totiž již navazuje na tři bakalářské práce a v této chvíli tak existují dvě nezávislé verze manažerů.

3.1 Obecný princip fungování Manažera

Manažer je komunikační prostředek mezi uživatelem manažera a hráčů zápasu. Konkrétně se jedná o počítačový program s uživatelským rozhraním pro správu turnajů. Byl především dosavadními autory vytvořený jako učební pomůcka do předmětu o výuce základů umělé inteligence.

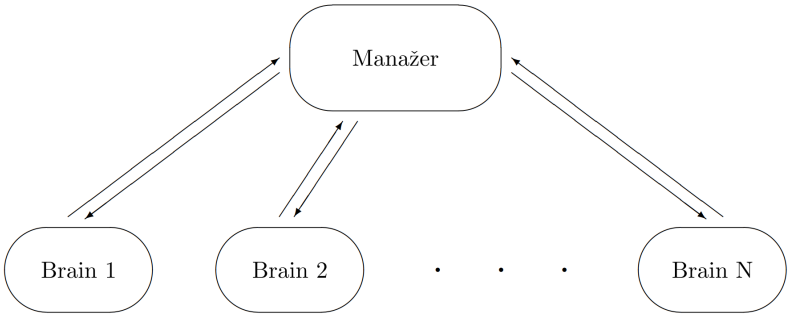
Manažer jako takový provádí turnaje a řídí celkově průběh zápasu. Obě verze taktéž mají schopnost zápasy zaznamenat do souboru XML s možností si zápas do manažera načíst a projít si jednotlivé tahy, které protihráči udělali. Manažer podporuje celkem 5 deskových her:

- Dáma
- GO
- Piškvorky
- Šachy
- Reversi

3.1.1 Princip brainů

Jednotliví hráči deskových her jsou reprezentovaní v programech jako brain. Brain je v obecném pojetí jednotka, která provádí tahy, může se jednat o umělou inteligenci nebo o člověka. Tento brain pak následně komunikuje s manažerem, kde sděluje svůj další tah. Manažer naopak průběžně brain informuje o průběhu hry. Jednotlivé brainy u konkrétní hry mezi sebou přímo nekomunikují, manažer je totiž prostředníkem pro takové brainy.

Motivací u prací bylo, že studenti předmětu budou moci si takové své brainy naprogramovat a následně je nechat hrát vybranou deskovou hru v manažeru. V minulých pracích jsou přiložené brainy udělané v jazyce C++. Jelikož jsou brainy samostatné programy, využívá manažer pro komunikaci s brainy síťové sokety na protokolu TCP.



Obrázek 3.1: Schéma komunikace manažera s brainy. Obrázek převzatý z [18]

Pro spolehlivou a jednoznačnou komunikaci byl vytvořený komunikační protokol s brainy, který se sestává ze zpráv a případně jejich argumenty. U obou verzí je protokol stejný.A.1

3.1.2 Forma turnajů

Jednotlivé brainy lze nechat hrát vybranou deskovou hru ve formě zápasů. Manažer umí pořádat takové turnaje pomocí tří turnajových systémů:

- Skupinový systém
- Švýcarský systém
- Systém hry každý s každým

Skupinový systém

Skupinový systém rozděluje jednotlivé hráče do skupin definované velikosti. Pro danou skupinu lze mít specifikované, kolik hráčů, kteří vyhráli, postoupí do dalšího kola. Je vhodné volit počet skupin, aby každá skupina byla stejně velká. Pro skupinu, kde by počet soupeřů nebyl jako je v dalších skupinách by se skupina sestávala ze zbývajících soupeřů (zbytek po dělení). U této situace je vhodné nechat takto zbývající hráče se přidat do nějaké skupiny, nebo pokud je hráčů více jak 2, nechat vytvořit skupinu zcela novou. Ve skupině probíhá hra každý s každým a podle výsledků, kdo vyhraje nejvíce ze skupiny postoupí. [17] [19]

Tento systém, pokud bude mít skupina vždy dva účastníky, lze považovat za vyřazovací systém a to buď na jedno kolo, nebo na dvě kola s tím, že pokud by počet hráčů nebyl sudý, může se přebývající hráč utkat až v druhém kole s náhodným vítězem z prvního kola. [6]

Švýcarský systém

Tento systém při prvním kole rozděluje náhodně hráče do dvojic, ve kterých se v prvním kole utkají proti sobě. Po skončení prvního kola, se hráči rozdělí do nových skupin podle jejich výsledku, tedy v druhém kole budou skupiny 1-0, 0-1 a 1-1 [12]. V manažeru se konkrétně seřadí hráči podle nejlepšího a utvoří se skupiny nové. Hraje se tolik kol, kolik bylo specifikováno na začátku před turnajem, po skončení jsou hráči seřazeni od nejlepšího po nejhoršího. [17]

Systém každý s každým

Jedná se o nejspravedlivější systém, ale zároveň časově nejvíce náročný.

3.1.3 Hodnocení turnajů

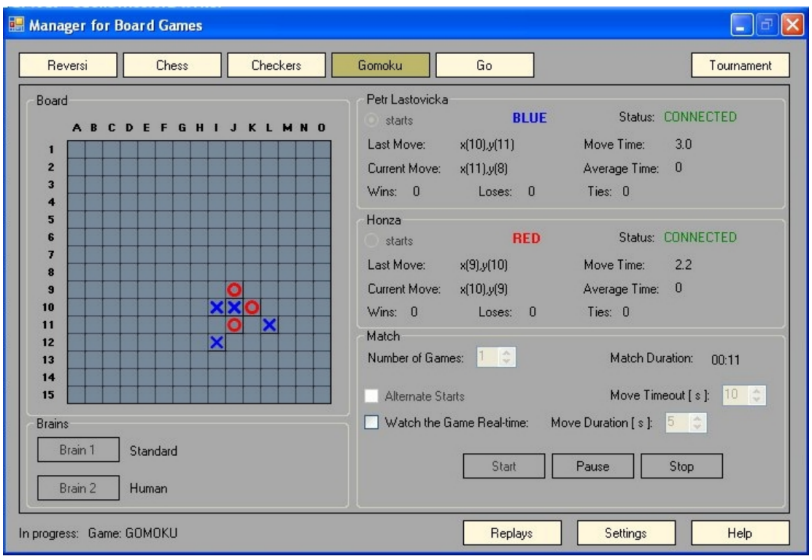
Při pořádání zápasů se v manažeru nachází dva systémy hodnocení. První hodnocení je za výsledek hry. Při výhře obdrží brain tři body, při remíze získá bod jeden a při prohře nezíská bod žádný. Toto hodnocení se pak také využívá pro umístění jednotlivých brainů v turnajích.

Vedle tohoto hodnocení je v manažeru počítaný z dlouhodobého hlediska i hodnotící systém ELO. Tento systém naopak počítá brainům celkové hodnocení, které je ovlivněné výsledkem při každém zápasu pro každý brain. Všechny brainy mají toto skóre na hodnotě 2000 a v průběhu zápasu se hodnocení snižuje nebo zvyšuje. Na rozdíl od konstantního přírůstku v prvním hodnocení podle výsledku zápasu, ELO naopak upřednostňuje vyšší změnu skóre u zápasu proti soupeři, který má výrazně odlišné hodnocení. Systém ELO především významně přidává u výhry proti silnějšímu soupeři, naopak při prohře proti podstatně slabšímu soupeři body také značně ubere. V případě vyrovnaných soupeřů tedy k zásadní změně hodnocení obou soupeřů nedochází. Podrobnější popis i s případným výpočtem použitým v manažeru je uvedený v pracích kolegů předešlých manažerů [18] [17].

3.2 První verze Manažera

První verzí, která byla vytvořena v technologii .NET framework a v jazyce C# je bakalářská práce, kterou začal v roce 2007 Jiří Kusák [18] a následně o dva roky později navázal a rozšířil Jan Kouřil [17] (dále jako Manažer 1). Program je určený na platformu Windows.

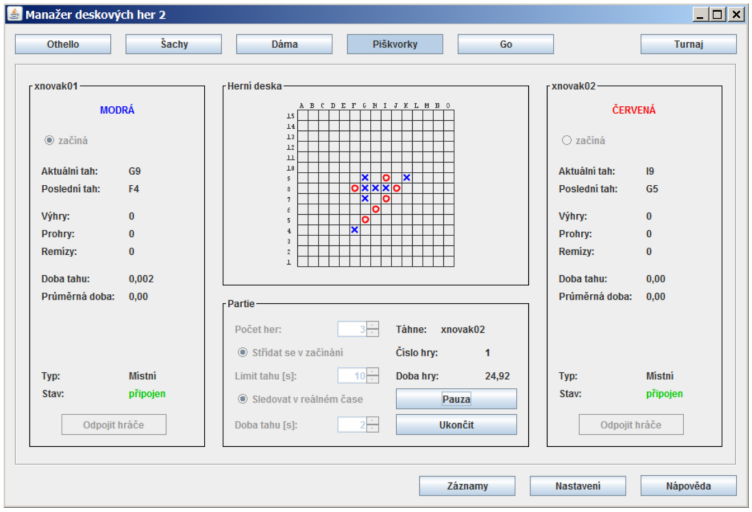
Manažer 1 podporuje všechny výše uvedené deskové hry a systémy turnajů. Při turnaji je manažer schopen řídit pouze jeden zápas současně. Dále neumí pořádat turnaje, kde při zápasu budou hrát proti sobě dva lidi.



Obrázek 3.2: Uživatelské rozhraní Manažera 1. Obrázek převzatý z [17]

3.3 Druhá verze Manažera

Druhá verze (dále jako Manažer 2) vznikla v roce 2015 jako bakalářská práce Tomáše Čížka [19]. Manažer 2 byl napsaný v jazyce Java a oproti Manažeru 1 umožňoval spustit jednotlivé turnaje navzájem paralelně. Naopak postrádal možnost zobrazovat při turnaji zápasy v reálném čase. Mezi jeho další přednosti dále patří navíc podpora anglického jazyka a u hry piškvorky a go umožňuje nastavit velikost herní plochy.

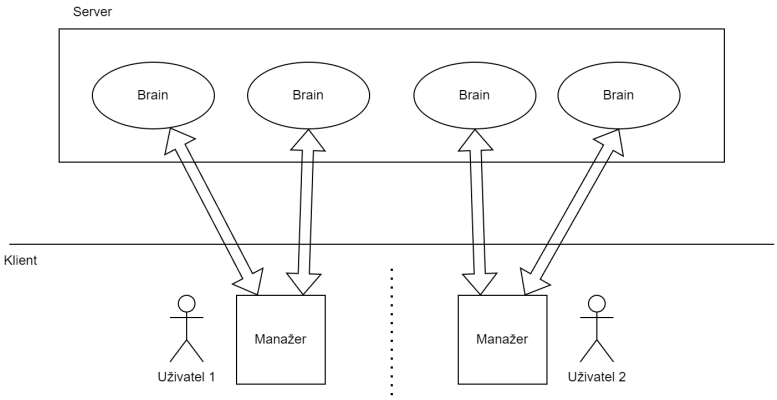


Obrázek 3.3: Uživatelské rozhraní Manažera 2. Obrázek převzatý z [19]

3.4 Architektura aktuálních manažerů

Manažer v obou verzích se dá považovat za dodělaný po funkční stránce. Po stránce praktické s ohledem na okolnosti a skutečně případné užití v praxi při výuce v semestru lze najít možnosti, jak udělat celou myšlenku lepší.

Přestože aktuální návrh manažerů je pro potřeby studentů dostačující, omezuje celkovou použitelnost vzhledem ke studentům v předmětu. Oba manažeři předpokládají, že brainy poběží na serveru. S tímto předpokladem bylo testování funkčnosti prováděno. Brainy běžely na serveru Merlin a manažer se spouštěl lokálně na počítači uživatele (resp. studenta nebo učitele). Aktuální architektura manažera tak omezuje používání manažera pouze za předpokladu, že brainy se nějakým způsobem dostanou na školní server Merlin, tam se následně mimo vliv manažera spustí, a pak může student nezávisle vyzkoušet funkcionalitu tím, že se jako klient připojí k brainům. V rámci třídy studentů v předmětu tak není žádná centrální správa, stejně tak studenti nejsou nijak ve vazbě s učitelem. Manažer je tedy pro zatím spíše jako pomůcka, kterou si student může lokálně vyzkoušet.



Obrázek 3.4: Diagram používání dosavadní verze manažera v praxi

Jak je vidět na diagramu užití 3.4, brainy běží na serveru nezávisle na sobě a nezávisle na klientech. Stejně tak klienti a jejich jednotlivé instance manažerů o sobě navzájem neví.

Mezi verzí Manažera 1 a Manažera 2 se vyměnily role mezi brainem a manažerem. Zatímco v první verzi byl manažer z pohledu těchto dvou entit jako klient, ve verzi dva se role z důvodu paralelismu prohodily [1]. I přes tuto změnu to nicméně neřeší problémy uváděné v této kapitole, protože se nejedná o celistvý pohled na systém jako takový.

Jako další zásadní nedostatek tohoto systému je také bezpečnost serveru, kde běží brainy. Vzhledem k tomu, že manažer ve své pozici nemá kontrolu nad tím, jak brainy jsou na serveru spuštěné, hrozí riziko, že špatně napsaný brain ať už úmyslně nebo neúmyslně může celý server shodit.

3.5 Změna architektury na webovou

K eliminaci předešlých problémů je nutné vytvořit novou architekturu manažera her tak, aby se sjednotily brainy, uživatelé a místo, odkud se k brainům přistupuje. V dnešní době je vhodné modelovat takový informační systém ve webovém prostředí. Použití webu totiž přináší výhodu nejen přenositelnosti – k manažerovi lze přistupovat prakticky odkudkoliv a z jakéhokoliv zařízení. Manažer 2 sice řešil přenositelnost použitím jazyka Java, který může běžet všude, kde je podporovaný JVM[2], nicméně neřeší širší náhled v podobě sjednocení systému.

Při návrhu webové architektury je nutné začínat s projektem prakticky od začátku. Použití stejného programovacího, ve kterém je napsaný manažer může sice pomoct ušetřit psaním některých částí funkcí, neušetří nicméně v celkové koncepci systému.

Takový systém, aby byl použitelný pro potřeby předmětu IZU[3], je nutný udělat centralizovaně. Jako inspiraci lze považovat tzv. ISU Hub, který vytvořil p. Ing. Filip Orság Ph.D. pro potřeby výuky předmětu ISU[4].

Pro srovnání – systém ISU Hub je používaný na spouštění studentských programů v jazyce Assembly především při konání testů v průběhu semestru. Informační systém musí

[1] Tím, že manažer bude brainy spouštět paralelně, je jednoudšší, když brain bude v roli klienta a manažer v roli konkuretního serveru, aby se nemusely řešit prodlevy s vytvářením soketů na straně brainů

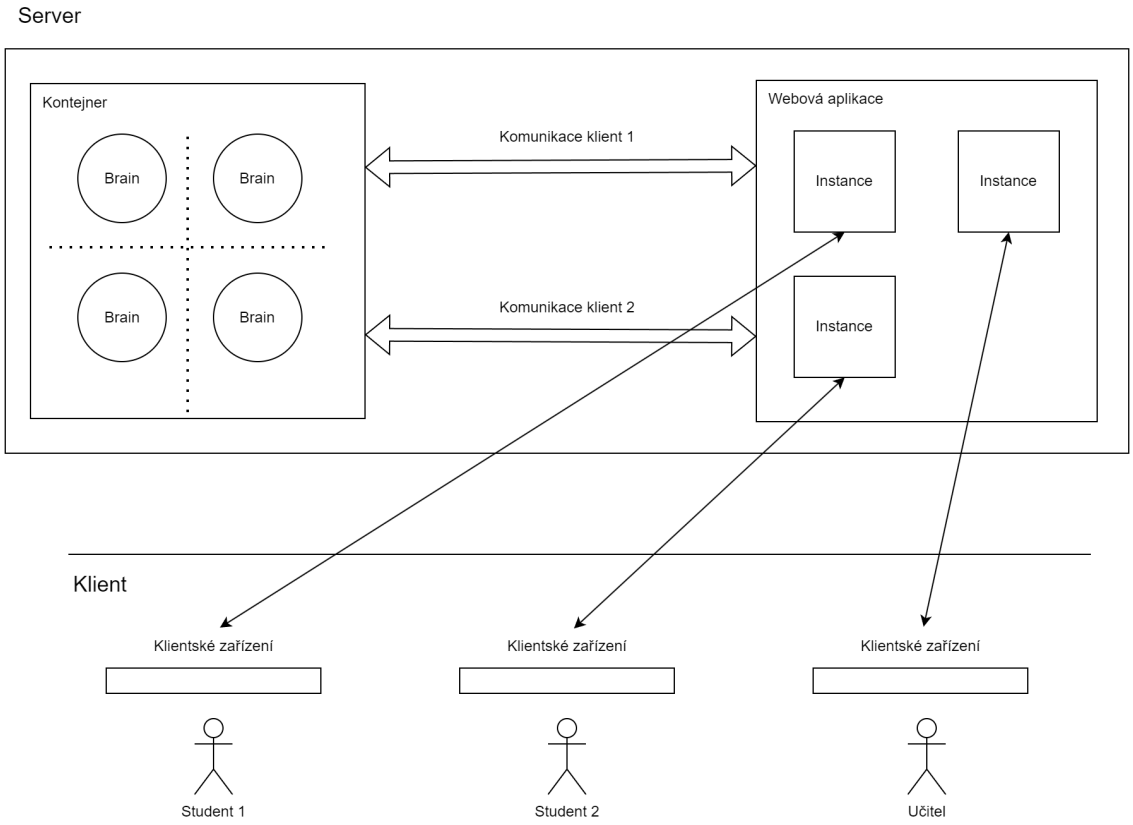
[2] Java Virtual Machine – runtime používaný pro spuštění zdrojových souborů psaných v jazyce Java

[3] Předmět Základy umělé inteligence

[4] Předmět Programování na strojové úrovni

zajistit, aby tyto programy byly dostatečně izolované navzájem a nezpůsobily pád celého systému, zároveň se klientská část (tj. přístup pro studenty) nachází na webu, který je v této chvíli i propojený se Studis systémem.

ISU Hub se minimálně v části spouštění studentských kódu bezpečně a centralizovaného přístupu shoduje s požadavky na funčknost nového webového manažera.



Obrázek 3.5: Diagram navrhované architektury webového manažera

Z diagramu 3.5 je patrné, že se u této architektury bude řešit hned několik nových problémů, které v dosavadní verzi řešeny být nemusely:

1. Komunikace s klienty (uživateli) přes síť
2. Řízení uživatelských přístupů a oprávnění
3. Komunikace s brainy v kontejneru
4. Správa kontejneru s brainy

Všechny tyto problémy a jejich řešení budou popsané v následujících kapitolách. I přes zmíněné problémy tento hrubý návrh řeší všechny nedostatky aktuální architektury manažera.

Kapitola 4

Virtualizace brain procesů

Manažer bude umožňovat spouštět programy (brainy) na hostovaném serveru. K tomu, aby se zamezilo, aby studentem napsaný program způsobil neplechu na serveru ať už úmyslně, nebo neúmyslně, je nutné každý takový program rozumnými prostředky izolovat. Na obrázku 3.5 jsou tyto brainy reprezentované v kontejneru.

Již výše zmíněný ISU Hub provádí izolaci studentských assembly programů, nicméně na tomto systému nejsou požadavky na izolaci tak přísné, jaké budou v manažeru. Na rozdíl od assembly instrukcí, které lze před kompilací přímo zakázat a tím zajistit relativně bezpečný program, programy brainů budou složitějšího charakteru. Zejména budou komunikovat přes síťové sokety a není žádoucí zbytečně omezovat možnosti použití knihoven a dílčích funkcí systému pro studenty. Tato kapitola se zaměřuje na popis a hledání řešení, která lze ve webovém manažeru uplatnit.

4.1 Koncepce virtualizace

Pojem virtualizace lze formálně popsat jako proces fyzické abstrakce výpočetních zdrojů. Výpočetními zdroji lze pak považovat procesor (CPU), paměť, disky, grafický procesor (GPU) nebo síťové prvky. Takto virtualizovaný stroj se běžně v literatuře zkracuje na VM (z ang. virtual machine). Systém, který VM v sobě provozuje se pak nazývá host[1]. Host takto vyhrazené hardwarové prostředky propůjčí VM. Jednotlivé VM v sobě host vidí pak jako běžící programy s vyhrazenými prostředky.[13]

V obecnějším pojetí se totiž virtualizace nevztahuje pouze na abstrakci operačního systému (dále již OS), ale i na obecnější metody, technologie a způsoby (pomocí hardwarového, softwarového rozdělování, simulace na hardwaru, softwaru nebo sdílení času), práce se bude zabývat pouze virtualizací systémů.[11]

Virtualizovaný systém má možnost zjistit, zda-li konkrétně běží virtualizovaně. Intel a AMD vyhrazují ve svých procesorových registrech tzv. hypervisor bit, který se nachází na 31. pozici ECX registru při zavolání instrukce CPUID. Jednotlivé procesory od těchto výrobců nastavují bit na hodnotu 0. Při běhu systému virtualizovaně, tedy pod hypervisorem (pojem bude později rozebrán podrobněji) je bit nastavený na hodnotu 1.[3]

[1] Slovo "Host"v tomto kontextu je spíš převzaté anglické slovo, které má přesně opačný význam, než český ekvivalent slova host

4.2 Přehled dostupných metod virtualizace

Při použití virtualizace se nejčastěji myslí virtualizace platformy, existují ale i další druhy virtualizace, které pomůžou k pochopení konceptu virtualizace.

4.2.1 Aplikační virtualizace

V tomto případě dochází pouze k přerozdělování zdrojů jediné aplikaci. Virtualizace namísto na úrovni OS je o vrstvu výše a reprezentuje běžící prostředí (např. ve formě programu), ve kterém aplikace běží. Zpravidla dochází k překladu systémových volání tak, aby běžící aplikace "nabyla dojmu", že běží na jiném systému. Tento přístup se používá například u starších aplikací, které nejsou na novém systému kompatibilní. Virtualizací lze i zajistit běh aplikací na zcela jiném systému.

Příkladem takového prostředí může být známý program Wine, který je v linux komunitě známý svými vlastnostmi simulovat prostředí Windows volání tak, že jednotlivé systémové volání Windows překládá v reálném čase na POSIX volání [2]. Díky tomu jsou programy jinak kompatibilní pouze na systémech Windows spustitelné i na Linux distribucích. [11]

Dalším příkladem může být i program Firejail, který také dokáže aplikace uzavřít do sandboxu. Tento program je mimo jiné použitý v programu ISU Hub.

4.2.2 Virtualizace platformy operačního systému

Tato metoda virtualizace je více používaná a považovaná za metodu virtualizace obecně. Virtualzace totiž probíhá na úrovni celého OS se všemi svými aplikacemi a prostředky. Program běžící na hostovi zajišťující běh virtualizovaných OS se nazývá hypervisor[2] [13] [11]

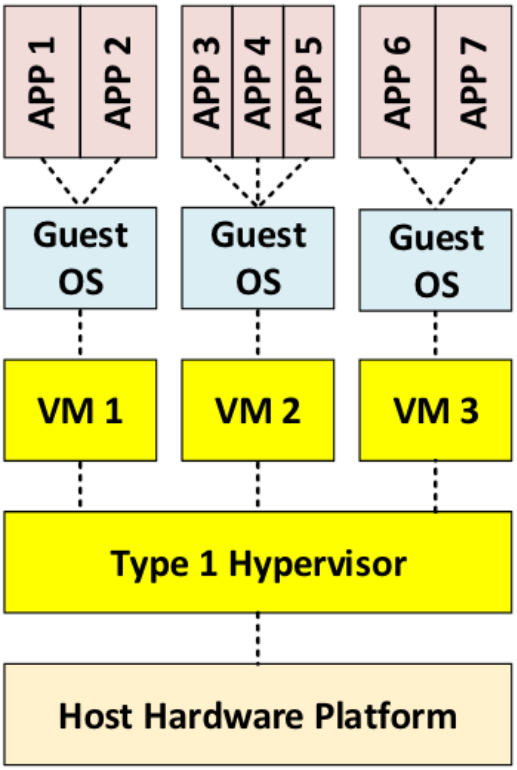
4.3 Hypervisor

Hypervisor se stará o to, aby jednotlivým OS přiřadil celý set virtualizovaného hardwaru každému VM izolovaně tak, aby se navzájem nemohly ovlivňovat. Více virtualizovaných OS běží na hostovaném hardwaru "vedle sebe"a pokud se nepoužije například síťové komunikační rozhraní na komunikaci mezi těmito systémy, nemají jinak znalost, že takto vedle nich běží další virtualizované systémy. Podle místa a nasazení hypervisoru se hypervisor dělí na dva typy. [13] [11]

4.3.1 Hypervisor prvního typu

Hypervisor 1. typu běží přímo na hardwaru systému (bare-metal) a poskytuje veškerou alokaci zdrojů všech výpočetních prostředků. Toto ilustruje obrázek 4.1 [9].

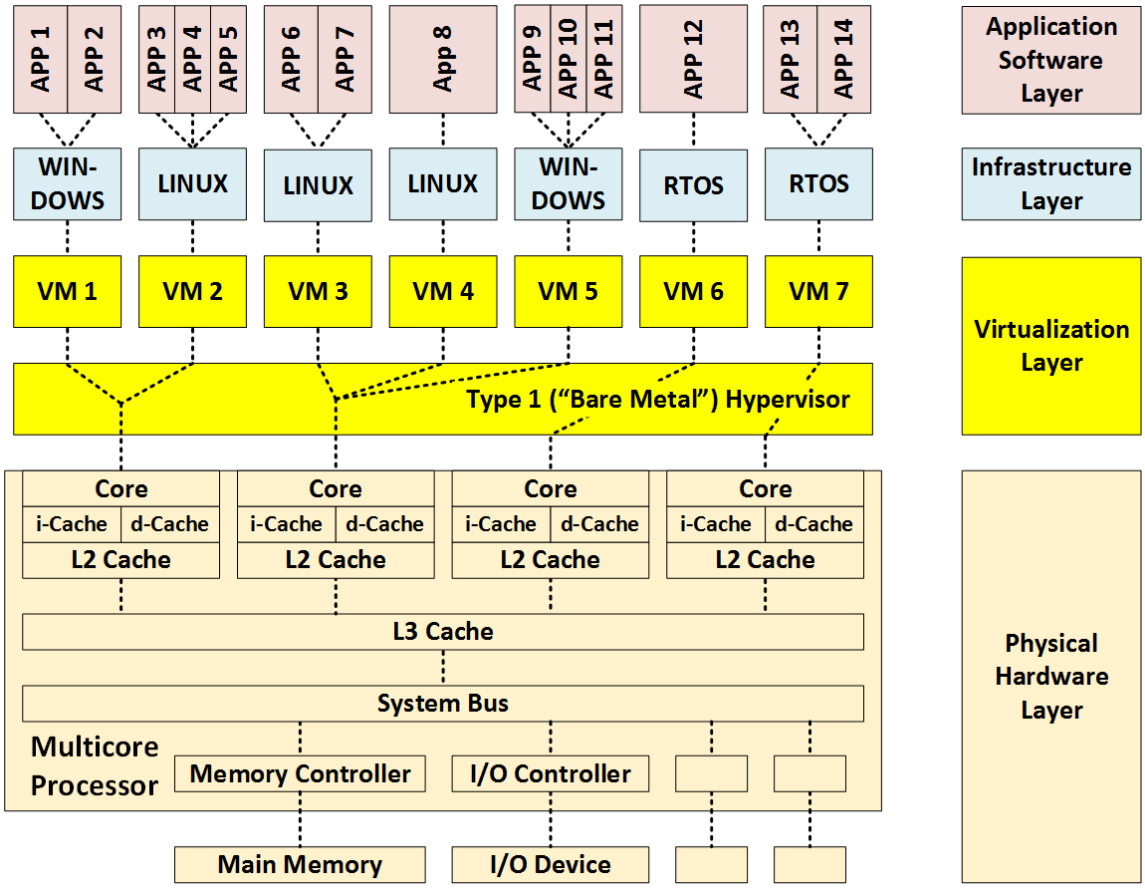
[2] Jinak také nazývaný jako monitor virtuálních strojů (ang. virtual machine monitor zkracovaný v literatuře jako VMM)



Obrázek 4.1: Ilustrace systému s Hypervisorem prvního typu

Díky tomu, že běží buď ještě pod samotným operačním systémem (OS) na hostovaném stroj nebo vedle něj s přímým přístupem k hardwaru, má tento typ zpravidla malý dopad na výkon samotných virtualizovaných systémů[3]. Celkový pohled na systém, kde je vidět tok dat z výpočetních prostředků hostovaného počítače je ilustrováno na obrázku 4.2 [9].

[3] V řádech desetin až jednotek procent celkového surového výkonu hardwaru na hostovaném stroji



Obrázek 4.2: Vrstvy v systému s hypervisorem prvního typu

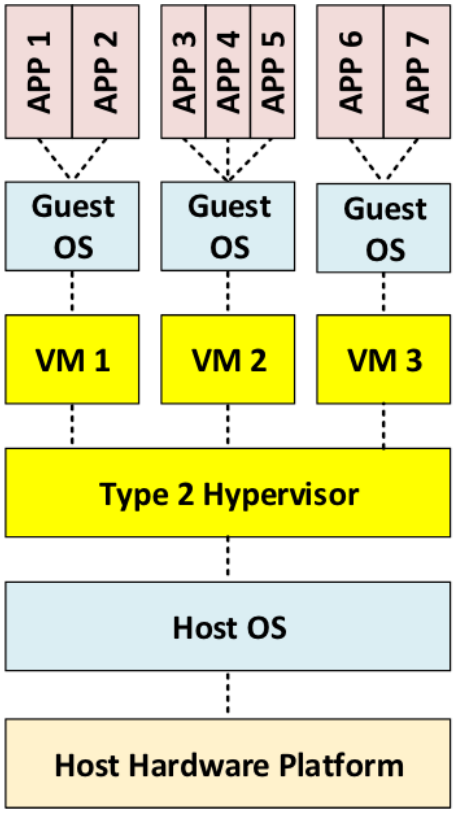
Jejich nevýhodou bývá omezený set dostupného hardwaru, který poskytuje ovladače na tyto hypervisory, aby mohly na hardwaru běžet a virtualizovat daný hardware. Některé hypervisory poskytují minimalistický OS, který musí na hostovi běžet v privilegovaném režimu. Tento systém je častokrát spíš komunikační konzolí a nazývá se Domain-0 nebo Dom0. [11] [9]

Příklady hypervisoru prvního typu lze považovat Kernel Virtual Machine známý pod zkratkou KVM nacházející se v jádře systému Linux. Další známým příkladem lze považovat Virtuální prostředí od společnosti Proxmox, která nabízí hypervisor řešení umožňující běh VM v serverovém prostředí. Vedle této virtualizace nabízí i webovou konzoli na správu jednotlivých nainstalovaných systémů.

Pro platformu Windows je po boku Windows kernelu nainstalovaný Hyper-V, který má také přímý přístup k hardwaru. Technologii Hyper-V také využívá systém nejnovější verze WSL (modul Windows subsystem for Linux), který po boku Windows dovoluje mít nainstalované různé distribuce systémů Linux.

4.3.2 Hypervisor druhého typu

Hypervisor druhého typu naopak ke svému správnému běhu potřebuje OS. To znamená, že běží zpravidla jako program na OS hosta (viz obrázek 4.3 [9]). Toto může mít výhody v kompatibilitě hardwaru, protože hypervisor se již přímo nestará o virtualizaci hardwaru, ale spoléhá na to, co mu OS poskytne. [11]

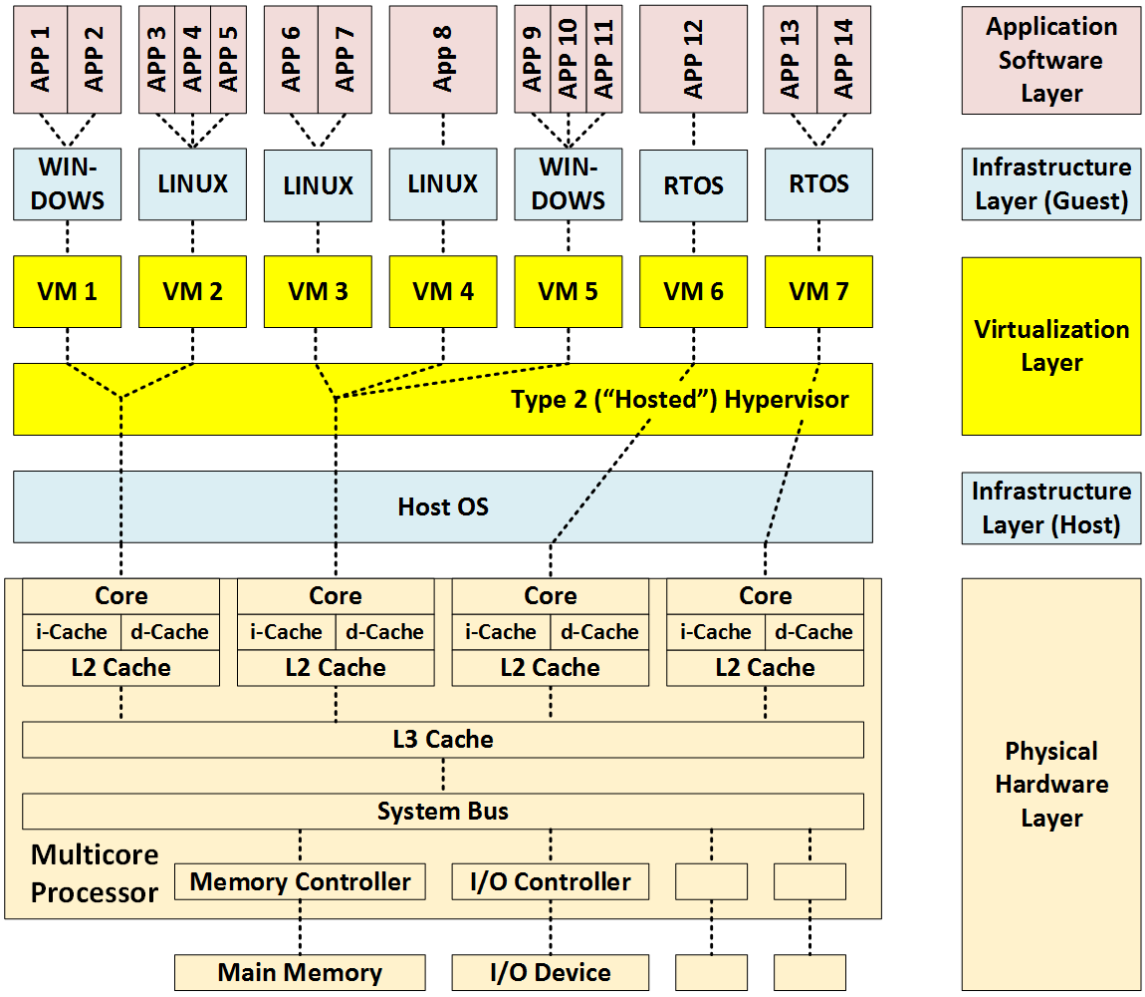


Obrázek 4.3: Ilustrace systému s Hypervisorem druhého typu

Příklady typických hypervisorů druhého typu může být například program VirutalBox, který je opensource. Jeho známou alternativou, dříve placenou (nyní bezplatné pro osobní užití), může být VmWare Workstation software pro platformy Windows. Oba programy umožňují virtualizaci celých OS.

Vedle programů na virtualizaci OS lze také zařadit i prostředí pro běh některých programovacích jazyků, kde zejména spadá JVM (Java virtual machine), což je běhové prostředí, které umožňuje spouštět java kód na jakékoliv počítačové platformě, kde se bude JVM nacházet. Alternativně lze i považovat za hypervisora i CLR (Common language runtime), který se naopak stará o běh jazyka IL (itermidiate language). Do tohoto jazyka je přeložený jazyk C#.

Oproti hypervisoru prvního typu naopak vyžaduje vyšší část celkového výkonu hardware na virtualizaci pro VM, protože VM je jako další vrstva nacházejícího se od hardware. Toto ukazuje obrázek 4.4 [9]. Jednotlivý tok z hardwaru tak musí procházet přes další vrstvu hypervisora, než se dostane na obsluhu VM.

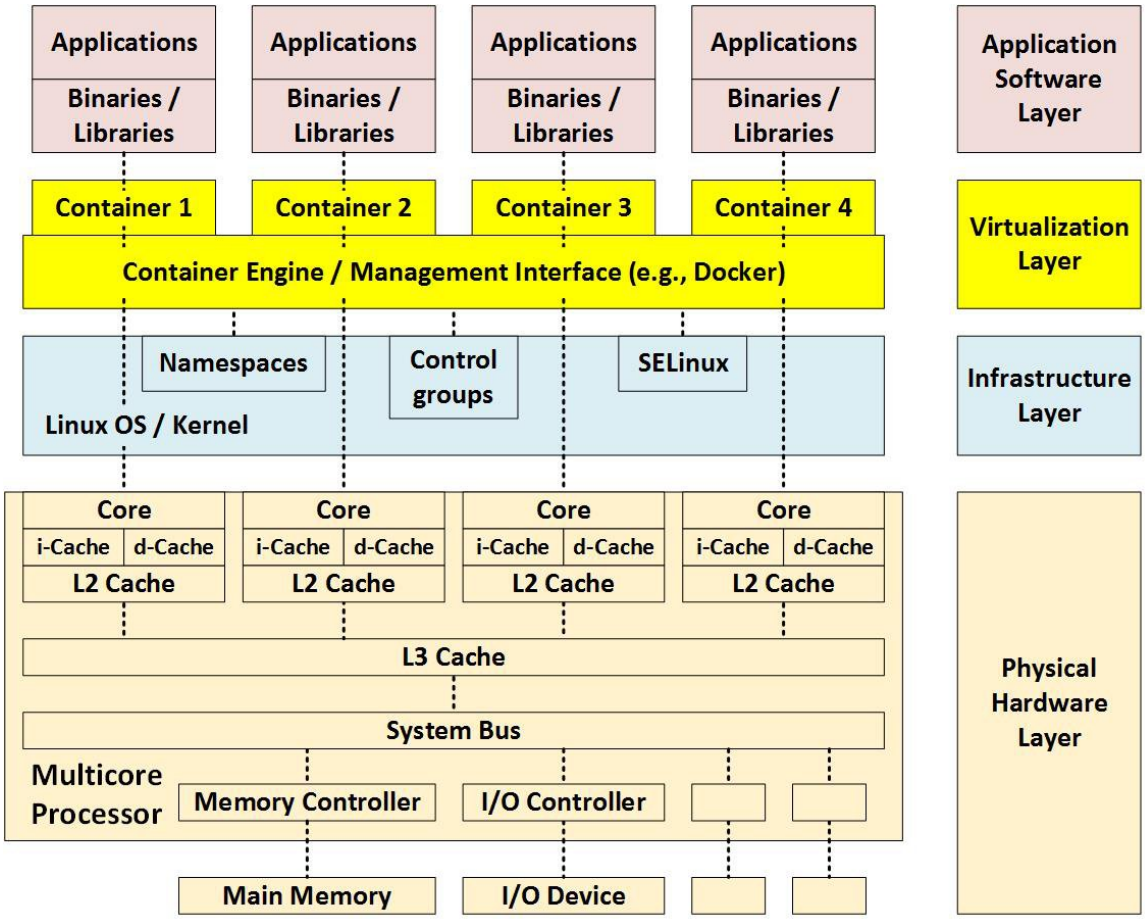


Obrázek 4.4: Vrstvy v systému s hypervisorem prvního typu

4.4 Obecný princip kontejnerizace virtuálních systémů

Přestože výše zmíněné metody hypervisorů a virtualizace přináší své výhody v kompletní izolaci a kontroly přerozdělení hardwarových prostředků, jedná se u obou přístupů ke zvýšené náročnosti na hardware. U virtualizace OS je nutné virtualizovat celý systém se všemi jeho knihovnami, což zabírá nemalé množství paměti. Vzhledem k tomu, že manažer bude spouštět několik stovek VM, není možné tyto přístupy využít v takové podobě, jaké jsou prezentované.

Kontejnerizace posouvá virtualizaci o úroveň výše na úroveň samotného OS. Jednotlivým kontejnerům virtualizuje OS tím způsobem, že sdílí jádro OS mezi kontejnery. Jednotlivé aplikace v kontejnerech si nejsou vědomi, že další kontejnery běží vedle nich. [10]



Obrázek 4.5: Vrstvy v systému Linux s kontejnery

Jednotlivé vrstvy jsou ukázané na obrázku 4.5 [10]. Řídící rozhraní kontejnerového engine zajišťuje propůjčování společných zdrojů jednotlivým kontejnerům. Ty se na obrázku nachází v aplikační vrstvě. Tato vrstva tak obsahuje minimální aplikační vybavení, které stačí k tomu, aby aplikace v kontejneru mohla běžet.

Lze tak docílit velmi malých kontejnerů i v řádech megabajtů v operační paměti, zároveň ale poskytovat plnou virtualizaci a izolaci procesů, které v kontejnerech běží.

Nevýhodou kontejnerů je nemožnost spustit jiný OS, než na kterém rozhraní pro kontejnerizaci běží. To znamená, že například není možné spustit kontejner OS Windows na hostovaném OS Linux, protože kontejner pouze sdílí jádro a společné prostředky. Toto není omezení, protože návrh manažera je koncipován tak, že bude běžet celý na OS Linux.

4.4.1 Popis prostředků pro podporu kontejnerů v systému Linux

Přestože mají kontejnery sdílené jádro hostovaného systému, Linux obsahuje různé mechanismy, které zaručují izolaci a zabezpečení vnitřku kontejnerů i vnějšku při jejich správné konfiguraci.

Namespaces

Namespace zapouzdřují globální systémové prostředky[4] do druhu abstrakce, který pro jednotlivé procesy vytváří dojem, že vlastní svoji izolovanou instanci globálního systémového prostředku. Jednotlivé změny u globálních prostředků jsou viditelné pro takovou skupinu procesů, které jsou součástí namespace, ale nejsou viditelné pro ostatní procesy. Namespace se mimo jiné používají pro implementaci kontejnerů. [15]

Control groups

Control groups (zkracovaný také jako CGroups) je skupina procesů, která má jádrem přidělené omezení na využití systémových prostředků. Jednotlivé skupiny mohou být jádrem systému sledovány na využívání prostředů. Skupiny jsou v jádře Linuxu tvořené pomocí pseudo-souborového systému, zatímco jejich monitorování je docíleno pomocí subsystémů[5] pro každý systémový prostředek.[16]

SeLinux

NSA Security-Enhanced Linux (zkracovaný jako SeLinux) je flexibilní povinný systém pro správu kontroly přístupu. Daná architektura poskytuje možnosti uplatnit jednotlivé politiky na souborový systém, uživatelské role nebo na systém jako celek.[14]

4.5 Docker

Na poli dnešních řešení je hojně užívaná otevřená platforma Docker. Docker umožňuje vytvářet kontejnery, spravovat je a řídit. Na řízení jednotlivých docker kontejnerů používá Docker API / CLI na jejich správu. Jedná se především o sadu příkazů, které řídí jednotlivé části Dockeru.

Docker API je podle OCI (Open Container Initiative)[6] 100% kompatibilní, to znamená, že API volání na Docker CLI lze jednoduše převést na volání jiných systémů na správu kontejnerů jako je například Podman.
Pro ilustraci, následující příkazy:

- `docker run`
- `docker pull`
- `docker push`

lze převést včetně jejich argumentů příkazové řádky na ekvivalentní volání v programu podman:

- `podman run`
- `podman pull`
- `podman push`

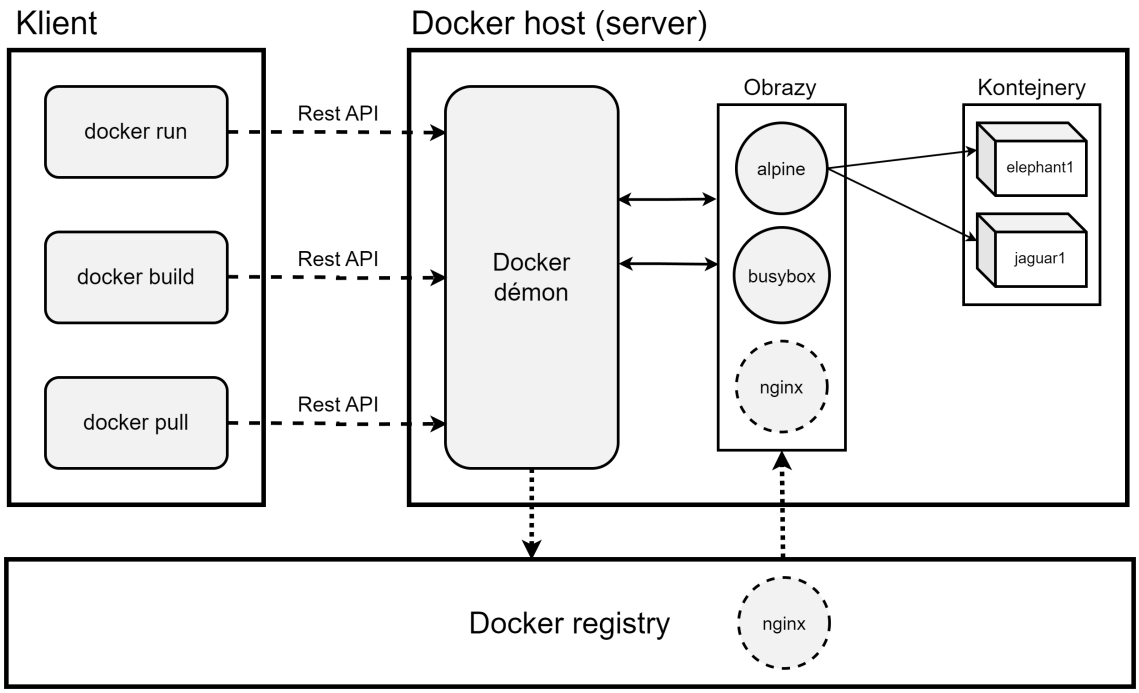
[4] Rozdělení typů namespace pro prostředky jsou: IPC, síť, mount odkazy pro disky a uložiště, PID, čas, uživatelé

[5] Subsystém je komponenta jádra která dokáže ovlivňovat chování jednotlivých procesů v cgroups

[6] OCI, je organizace, která zajišťuje otevřený standard na ovládání různých systémů na správu kontejneru. Stránka OCI: https://opencontainers.org/

4.5.1 Architektura Dockeru

Docker se skládá z klient-server architektury. Na straně server části se nachází docker démon. Démon se stará o nasazení a spouštění jednotlivých kontejnerů. Klientská část se pak nazývá docker client, přes kterou se primárně komunikuje např. při používání CLI rozhraní. Klient i démon komunikují pomocí REST API přes UNIX sokety[7]. Díky tomu nemusí být démon ani klient na stejném zařízení, ale můžou komunikovat nezávisle, kde jsou instalovány. [1]



Obrázek 4.6: Zjednodušené schéma klient-server docker architektury

Docker image

Na obrázku 4.6 se nachází popisované schéma docker architektury. Každý kontejner vychází z obrazu, který určuje, jakým způsobem se má virtualizovat daný kontejner, jaké knihovny bude mít a co vše v něm bude prvnotně zahrnuté. Obraz slouží pro kontejnery tedy jako šablona.

Každý takový image může vycházet z jiného image, lze takto vrstvit jednotlivé závislosti knihoven a dílčích částí systémů zahrnuté v kontejneru. Díky tomuto přístupu lze kontejnery v dockeru provozovat velmi efektivně, protože pokud obraz B vychází z obrazu A, který je již nasazený, nemusí si při vytváření kontejneru stahovat znovu obraz A, ale jednoduše se použije při obrazu B a stáhnou se jen závislosti, které stažené v Docker hostu nebyly.

[7] primární soket se nazývá docker.sock a nachází se v lokaci /var/run/docker.sock

Docker registry

Docker registry představuje repozitář s obrazy. Veřejný a používaný repozitář se nazývá Docker Hub[8], kde se nachází všechny publikované oficiální i neoficiální obrazy komunitou. Docker registry lze i provozovat soukromě v rámci například lokální sítě.

Jednotlivé obrazy mají vrstvenou architekturu. Nová verze obrazu si uchová změny od původní verze. Podobně funguje vytváření obrazů na základě již existujících obrazů. z Docker Hub lze stáhnout pomocí příkazu `docker pull <nazev_image>`. Konkrétně tak například pro jeden z nejznámějších obrazů kontejneru - alpine linux, je pak příkaz následuící:

```
docker pull alpine
```

Obraz se takto lokálně uloží do dockeru na hostovaném systému a bude k dispozici při vytváření kontejnerů. Při stažení obrazu, který vychází z alpine linux se stáhnou pouze soubory potřebné pro rozšiřující obraz, zbytek závislostí se použije z již staženého alpine obrazu.

Dockerfile

Kromě stahování obrazů z registrů lze takto lokálně obraz sestavit. Na to, jak má obraz následně vypadat slouží dockerfile, který má definovanou strukturu na stránkách dockeru[9].

V Dockerfile může být definované z jakého obrazu nový obraz vychází, co se do něj má kopírovat při spuštění, jaké příkazy spustit, nebo i jaké porty mají být přístupné ven hostovi a jak se do docker kontejneru mapují. Takový dockerfile potom lze jednoduše použít příkazem `docker build`, který obraz sestaví. Do Docker registrů se takový dockerfile dá následně nahrát a zpřístupnit ho ostatním.

4.5.2 Izolace Dockeru od hosta

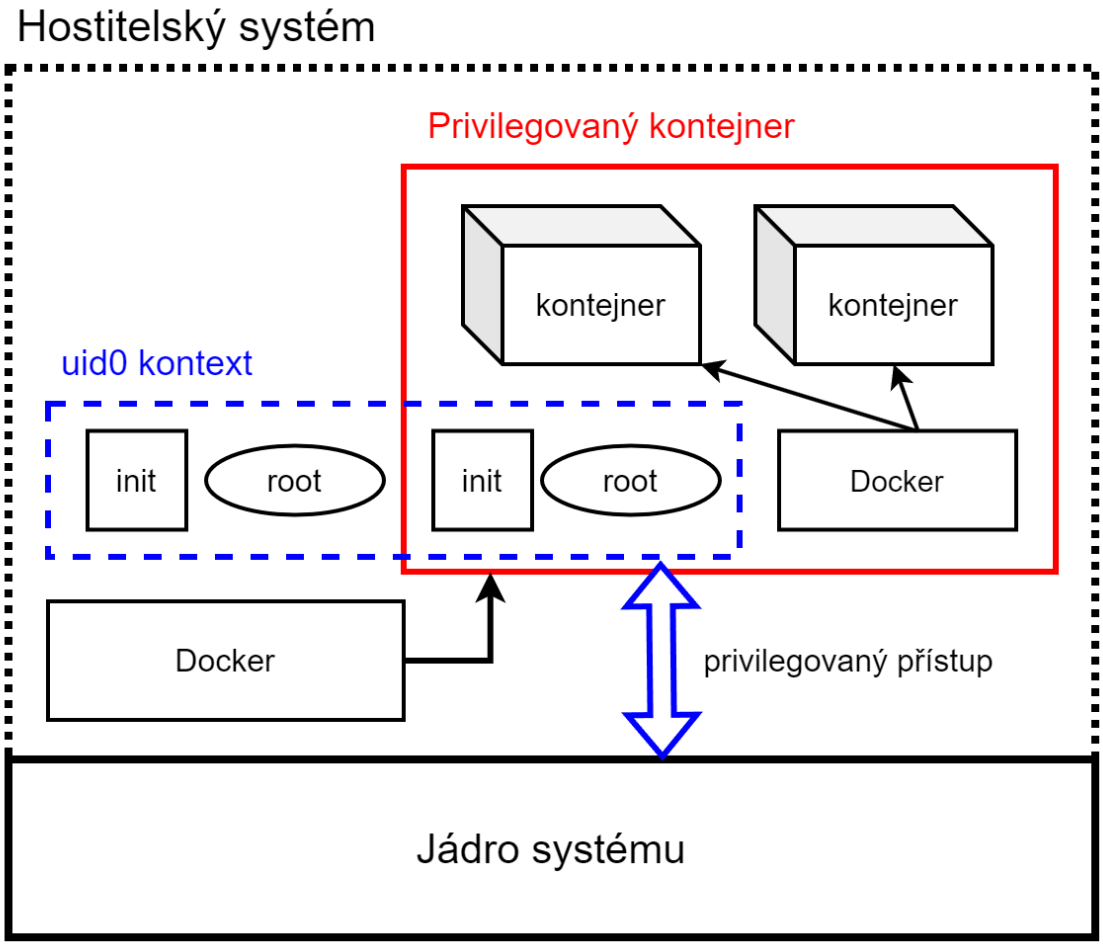
V základním nastavení je kontejner v dockeru izolovaný od hostitelského OS. Nicméně je nutné se vrátit zpět na architekturu návrhu webového manažera, kde je jeden z požadavků izolovat kontejnery s brainy ideálně každý ve svém kontejneru. Na to ale, aby mohli komunikovat ze vnějšku pomocí TCP je nutné propustit porty do hostitelského systému. Vzhledem k tomu, že takových kontejnerů může na daném systému běžet v plné zátěži až 500, může to komplikovat obecně obsazení portů. Zároveň je žádoucí, aby celkový systém bylo možné pokud možno co nejjednodušeji nasadit na jakýkoliv server se systémem Linux, proto i samotná webová aplikace manažera (která je oddělená od brainů) by měla být také v docker kontejneru. Webový manažer se nebude chovat v klientském režimu jako se budou chovat brainy, bude potřeba, aby manažer byl schopen jednotlivé docker kontejnery s brainy mít ve správě. Na řešení se nabízí dva přístupy, které tuto situaci na první pohled mohou řešit:

- Docker in docker (DinD)
- Docker out of docker (DooD)

[8] https://hub.docker.com/
[9] https://docs.docker.com/engine/reference/builder/

4.5.3 Docker v dockeru

DinD (docker in docker) je přístup, kdy se docker kontejner spustí v privilegovaném režimu a následně se uvnitř tohoto dockeru spustí tzv. child kontejnery.



Obrázek 4.7: Schéma použití přístupu docker v dockeru pro docílení izolace případné hostitelské aplikace, která by potřebovala ovládat docker kontejnery nezávisle na hostitelském systému, kde privilegovaný kontejner běží.

Docker použitý pro spuštění vnořených dockerů musí běžet v privilegovaném režimu k zajištění přístupu k prostředkům a jádru hostitelského systému bez omezení. Privilegovaný režim v tomto kontextu znamená, že user-namespace id 0 v kontejneru má stejný kontext jako namespace-id 0 v hostitelském systému. Navzájem jsou tyto skupiny mezi sebou promapované, a proto i například proces init v kontejneru má stejná práva jako má root na hostitelském systému. V kontejnerech, které nejsou privilegované je uživatel root v kontejneru mapovaný do hostitelského systému jako uživatel bez root práv. Root uživatel kontejneru tak za normálních okolností nemá práva na změny v jádru (hostitelského) systému. [8] [7]

V případě bezpečnostní chyby v kontejneru se útočník může dostat k možnosti ovládnout hostitelský systém skrz práva root. Spouštěné příkazy v rootu kontejneru by byly venku z kontejneru volány také jako root do jádra hostitelského systému.

Přestože docker v dockeru je oficiálně podporovaný[10], není doporučené kvůli bezpečnostnímu riziku takto docker kontejnery spouštět. [5] [8]

4.5.4 Docker vedle dockeru

Druhým přístupem je tzv. Docker out of docker (zkráceně DooD). Což je přístup, kdy se vybranému dockeru mapuje socket dockeru na hostitelkém zařízení, kde všechny dockery běží do jeho vnitřního docker soketu. Tím vznikne možnost komunikovat se vnějším docker démonem a ovládat tak dockery, které běží vedle daného dockeru.

Toto zahrnuje nutnost mapovat docker.sock do kontejneru, kde bude běžet webový manažer. Přístup má výhodu, že zcela obchází nutnost spustit kontejner v nezabezpečeném privilegovaném režimu, ale má také řadu nevýhod:

1. Kontejner s webovým manažerem má nejen přístup ke kontejnerům s brainy, ale má privilegovaný přístup ke všem kontejnerům na hostitelském systému

2. Při kompromitování zabezpečení může kontejner s webovým manažerem převzít kontrolu nad celým hostitelským systémem tím, že má přístup k démonu dockeru, který implicitně běží v privilegovaném režimu na hostitelském systému

3. Ztrácí se zamýšlená izolace jednotlivých kontejnerů a tím se tak zvyšuje riziko celkové nestability systému.

4. Dochází ke špatné izolaci docker kontextu. Docker CLI a Docker démon běží v každém v jiném kontextu a proto může vzniknout kolize při pokusu vytvoření nového kontejneru pomocí docker CLI, pokud již v Docker démonu takový pojmenovaný kontejner běží. To samé platí pro porty, které také vznikají na úrovni hostitelského OS a můžou být již obsazené

5. Mapované cesty do souborového systému musí být relativní vzhledem k démonu na hostitelském systému, nikoliv vzhledem k Docker CLI. Mapování se špatnou cestou pak takto vevnitř CLI kontejneru selže.

Vzhledem k těmto nedostatkům také není vhodné toto řešení v praxi použít. [5]

4.5.5 Sysbox

Řešením tohoto problému, který eliminuje problémy předchozích popisovaných dvou přístupů je použití volně dostupného runtime pro docker jménem Sysbox.

Sysbox poskytuje izolaci linuxových kontejnerů tím, že dovoluje používat přístup DinD (popisovaný v 4.5.3) bez privilegovaného stavu. Hlavní výhodou je, že hlavní docker může mít vnitřně privilegovaný režim a spouštět tak vevnitř systémové programy jako je docker, Kubernetes a další.

[10] https://hub.docker.com/_/docker

Kapitola 5

Návrh architektury webové aplikace

Na trhu se nachází mnoho frameworků, pomocí kterých lze takový systém udělat. Pro tuto práci jsem zvolil vytvořit webovou aplikaci ve frameworku ASP.NET. Pro back-end webové aplikace jsem zvolil jazyk C# s dotnet verzí 9 a pro front-end jsem využil variantu frameworku Blazor server, který používá jazyk razor pro vykreslování stránek. Vedle toho jsem použil grafickou knihovnu na komponenty MudBlazor[1], pomocí které jsem vytvořil prvky v uživatelském rozhraní. Systém jsem vyvíjel na platformě Windows v programu Visual Studio. Přestože je ASP.NET multiplatformní, kvůli dockeru jsem se nakonec rozhodl pro Linux, jako cílovou platformu aplikace. Při vývoji na Windows jsem využil možnost nasazovat kompilovaný program do WSL, takže rozhodnutí mi nijak nekomplikovalo vývoj.

5.1 Funkční požadavky na webovou aplikaci

Již v sekci 3.5 jsem zmiňoval, že je třeba vyřešit přístup několika uživatelů. ASP.NET umí řešit přístup několika uživatelům tím, že služby společně s klientským dotazem zaobalí do samostatné úlohy, zařadí do fronty a jednotlivá vlákna serveru si postupně asynchronně z fronty požadavky vyzvedávají a vykonávají. Vedle tohoto se nabízí jako další požadavky implementovat následující:

- podporovat komunikační protokol z původních manažerů
- uživatel může mít více brainů
- každý brain má N svých verzí (variací), které současně budou existovat v systému
- do turnaje se může přihlásit pouze jedna verze (variace) brainu z každého brainu
- V systému je role administrátor (učitel) a role uživatel (student)
- v turnaji, pokud to je možné, běží zápasy vůči sobě paralelně
- administrátor může pořádat turnaje
- uživatel se může zúčastnit turnaje svým brainem

[1] https://mudblazor.com/

- uživatel má možnost vybrat, kterou verzi brainu do turnaje přihlásí
- při turnaji může být brain za porušení pravidel protokolu komunikace nebo pravidel hry diskvalifikován a dále se nebude účastnit turnaje
- brain se v průběhu jednoho turnaje může učit a zlepšovat své fungování na základě zpětné vazby[2]

Při návrhu architektury a stanovených funkčních požadavků jsem zohlednil možnou rozšiřitelnost celého systému, proto jsem celý systém koncipoval pomocí vrstvené architektury. Tento přístup rozděluje systém na tři nezávislé vrstvy.

5.2 Vrstva přístupu dat

První vrstva se nazývá Data access layer a slouží pro práci se závislostmi systému po stránce přístupu k datům. U webového manažera vrstva obsluhuje přístup k databázi, komunikaci s docker engine pomocí REST API a přístupu do datového úložiště, kde jsou uložené zdrojové soubory všech brainů nahrané v systému.

5.2.1 Databáze

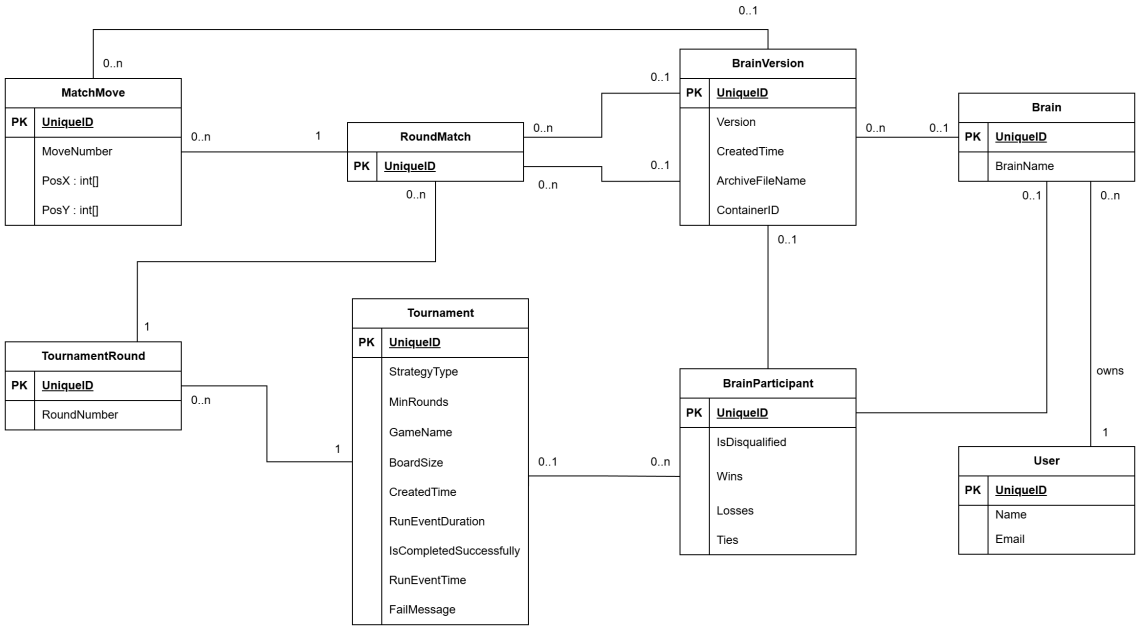
Webový manažer ve své implementaci používá SQL Lite. Systém používá dvě databáze oddělující data aplikace a uživatelské účty pro autentizaci a autorizaci. Pro samotný přístup k datům nevyužívám přímé volání SQL příkazů, ale používám Entity Core Framework (dále jako EF). Jedná se o hojně používaný framework v profesionálních systémech. Jeho výhodou je separace konkrétního databázového řešení od zbytku systému. EF jako takový poskytuje mapování entit v databázovém schématu na objekty programovacího jazyka systému. Tyto objekty pak slouží jako skutečné entity tj. jednotlivé tabulky v databázi a jsou dostupné přes tzv. databázový kontext `DbContext`. Na materializovaných objektech z kontextu lze volat pomocí LINQ metod podobné příkazy, jako jsou běžné v SQL jazyce. Příklad volání vracející konkrétního studenta z třídy podle ID:

```
var tournaments = classroomContext.
    .Where(classroom => classroom.StudentId == id).ToListAsync();
```

Zároveň lze modifikované objekty získané z kontextu zpátky do kontextu vkládat. EF automaticky zjistí, které hodnoty se změnily a tyto změny propíše zpátky do databáze.

Vedle SQL Lite podporuje další databázové řešení, ve zdrojovém kódu aplikace a tak lze velmi snadno SQL Lite nahradit za například Microsoft SQL server. Oba odkazy na připojení databáze jsou v konfiguračním souboru Webového Manažera.

[2] To v praxi znamená, že se kontejner s brainem nebude mezi koly mazat, ale jen restartovat, aby brain mohl mít uložené data v souborovém systému kontejneru



Obrázek 5.1: Schéma databázového modelu tak, jak jsou mapované třídy vůči tabulkám v databázi

Každá třída z obrázku 5.1 dědí z bázové třídy `EntityBase`, která udává povinnost třídě vlastnit ID. Systém používá `Guid` jako primární klíč. V nedávné verzi dotnet přidal možnost generovat `Guid` hodnoty ve verzi 7, které lépe předchází konfliktům, protože berou v potaz časovou značku v době jejich vytvoření.

Jelikož EF již nyní vytváří dobrou separaci doménové logiky od databáze, rozhodl jsem se separovat ve zbytku systému jednotlivé druhy primárních klíčů. Každý primární klíč každé entity je reprezentovaný vlastním datovým typem, což přináší bezpečnost před záměnou primárních klíčů při volání databázových dotazů. Tohoto jsem dosáhnul pomocí open-source knihovny Vogen[3]. Knihovna mi ušetří spoustu boilerplate kódu, kterému se v opačném případě nevyhnu. Její výhodou a možná i úskalím je, že využívá Roslyn generování kódu. C# s podporou meta-programování umožňuje psát generátory. Třídu lze označit například atributem. Takový kód se potom generuje na pozadí ve skrytých parciálních třídách. Nevýhoda tohoto přístupu spočívá ve skrývání kódu a vytvářením "magie v kódu"zejména, když bude chování neočekávané. Tohle může některé vývojáře odradit vlivem ztíženého odhalování příčiny chyb.

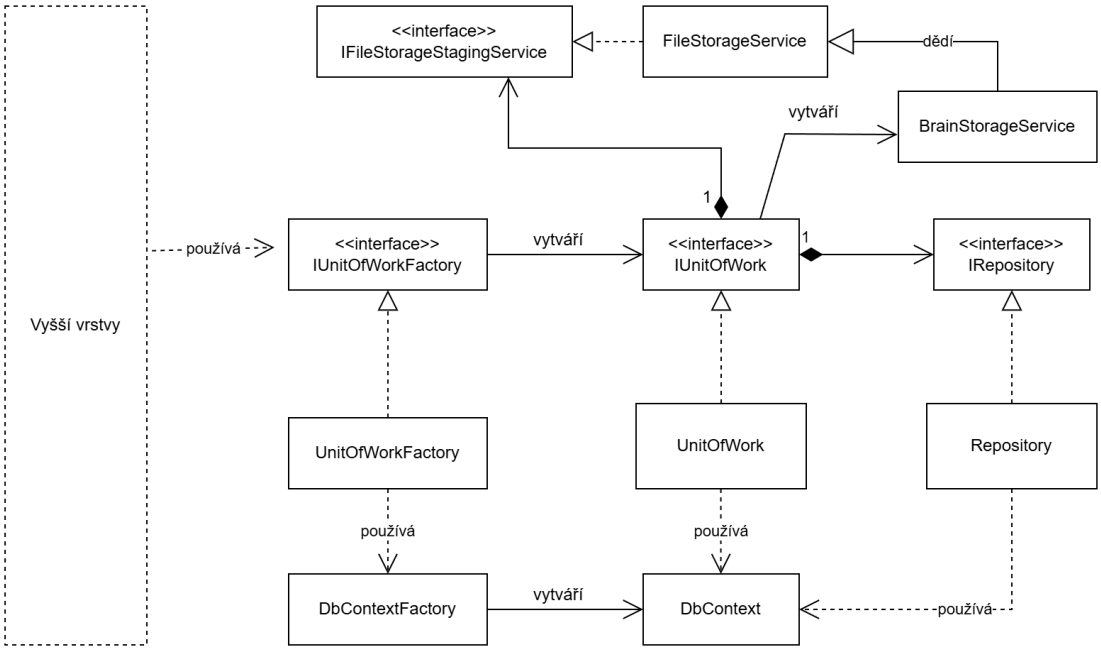
5.2.2 Repozitář a Unit of work

Databázový kontext (`DbContext`) poskytovaný EF je určený k použití v tzv. Unit of work přístupu (dále jako UOW). Každá transakce u každého případu užití daného systému se sjednotí na tzv. jednotku práce, která musí celá uspět (tj. všechny transakce a podmínky), aby celá operace do databáze byla úspěšně uložená. Z tohoto důvodu se získaný kontext nemůže používat vícekrát na různé případy použití. EF na tohle myslí a poskytuje továrnu na vytváření instancí kontextu.

[3]Dostupné z: https://github.com/SteveDunn/Vogen

Vzhledem k tomu, že vedle databáze potřebuji mít oddělený přístup k souborovému úložišti nahraných brainů, rozhodl jsem se navrhnout nad vrstvu databázového kontextu a jeho UOW vrstvu repozitáře a vlastního UOW návrhového vzoru, který zaobaluje transakce pro databázi a transakce do úložiště souborů pod jednu jednotku práce. Unikátní instanci UOW pak poskytuje pomocí návrhového vzoru továrna vlastní implementace.

Repozitář kromě poskytování základních CRUD operací je navržený takovým způsobem, že každá instance repozitáře se váže na jeden konkrétní druh entity. CRUD operace repozitář poskytuje vyšším vrstvám s již konkrétním unikátním datovým typem primárního klíče pro každou entitu. Tento přístup poskytuje typovou bezpečnost už při kompilaci a předchází chybám při záměnách primárních klíčů různých entit.



Obrázek 5.2: Zjednodušený diagram tříd v DAL vrstvě programu

5.3 Vrstva doménové logiky

Druhá vrstva nazývaná jako byznys vrstva (BL) je nejdůležitější vrstvou celého systému. Obsahuje totiž logiku specifickou pro konkrétní doménu celého systému. V BL se nachází algoritmy pro pořádání různých druhů turnaje, plánování turnaje a získávání dat podle funkčních požadavků programu 5.1.

BL vyšším vrstvám poskytuje skrz své fasády mapované modely, které jsou specifické pro doménové potřeby vyšších vrstev. U modelů není vyloučené, že může existovat N modelů pro jednu entitu a libovolný model může kombinovat data z M entit.

5.3.1 Nasazování brainů do kontejnerů

Docker umožňuje rozličnými způsoby zajistit, aby brain byl nasazený do dockeru. Prvním způsob je vytvoření image pro každou verzi brainu a tu následně nasazovat. Výhoda v tomto postupu je taková, že není nutné, aby po spuštění vytvořeného kontejneru běžel na pozadí

otevřený terminál, který by bránil zastavení kontejneru. Jeho hlavní nevýhodou je, že by bylo nutné mít desítky až stovky obrazů kontejneru pro každou verzi brainu. U návrhu samotné orchestrace nasazování brainů jsem se rozhodl pro druhý způsob. Konrétně mít vytvořený jeden obraz pro obecný kontejner s otevřeným terminálem na konci, aby docker zůstal aktivní, než se do něj zkopírují potřebné soubory.

Komunikace s dockerem

Komunikace s dockerem je realizovaná přes API pomocí knihovny Docker.DotNet [4]. Knihovna komunikuje s napojeným socketem pro docker, odkud komunikuje docker engine. Vzhledem k tomu, že webový manažer má být využitelný i ve výuce, je konkrétní adresa socketu exportovaná v konfiguračním souboru Manažera. Kromě toho soubor obsahuje i cestu pro dockerfile včetně jeho názvu, ze kterého se vytváří obraz pro všechny brainy a adresa, kam v každém kontejneru se rozbalují archivy brainů.

Návrh Manažera umožňuje kromě brainů psané v C nasadit brainy v prakticky libovolném jazyce. Vše záleží, jaký docker image se nahraje do systému. Zároveň není systém limitovaný pouze na práci s dockerem, ale může používat i například podman. Pro infrastrukturu systému se používá obecné rozhraní `IDockerService` pro komunikaci s externím poskytovatelem kontejnerů

5.3.2 Fasády

Fasáda po vzoru stejnojmenného návrhového vzoru slouží jako hlavní vstupní bod pro vyšší vrstvy. Všechny fasády mají ve webovém manažeru jednu bázovou generickou třídu `FacadeBase`, kde i mimo jiné probíhá mapovování entit na modely a zpět.

V obecném pojetí se dělá fasáda na jednu skupinu případů užití. Například má smysl uvažovat o fasádě pro brainy. Fasáda by obsahovala případy užití na vytvoření, smazání a nasazení brainu a případných podružných verzí. Kvůli tomu je fasáda často středobodem, kde se schází nejvíce služeb z informačního systému a vzájemně mezi sebou spolupracují a předávají si data.

Dependency injection

Aby bylo jednoduché fasády používat nabízí ASP.NET Dependency injection (zkracované na DI). Jedná se o komplexnější implementaci návrhového vzoru IoC (inversion of control). Na jednu stranu DI přebírá sice zodpovědnost konzumentů za dodání a inicializaci služeb, na druhou stranu centralizuje registraci tříd zpravidla na jedno místo. Konkrétní implementace Microsoftem navíc přidává možnost registrovat službu na tři různě dlouhé životní cykly:

- Singleton - služba v systému existuje v jedné instanci a existuje po celou dobu běhu programu
- Transient - služba je registrovaná u konzumentů každý s vlastní instancí služby
- Scoped - služba se vytvoří na dobu jednoho klientského požadavku (client request) a vytváří se na každý požadavek jako nová instance[5]

[4] Dostupné na https://github.com/dotnet/Docker.DotNet

[5] V tomto režimu je implicitně registrovaný databázový kontext

S využíváním DI jsem počítal v systému od samotného prvopočátku, zároveň jsem si při návrhu dal záležet, aby nikdy žádná třída nevyžadovala na registraci celý kontejner DI pro získávání libovolné služby ze systému. Takovému přístupu se říká service locator pattern a nejedná se o doporučovaný návrhový vzor.

5.3.3 Úlohy na pozadí

ASP.NET má obsluhu klientů na bázi fronty, kterou si vyzvedává thread-pool. Taková transakce má životnost jen po dobu vykonávání dotazu z klienta. Jakmile se dotaz dokončí, vykonávání úlohy se zastaví. Na dlouhotrvající úlohy má ASP.NET připravené řešení v podobě hostovaných služeb. Taková služba se musí přihlásit do Dependency injection příkazem:

```
services.AddHostedService<Service>();
```

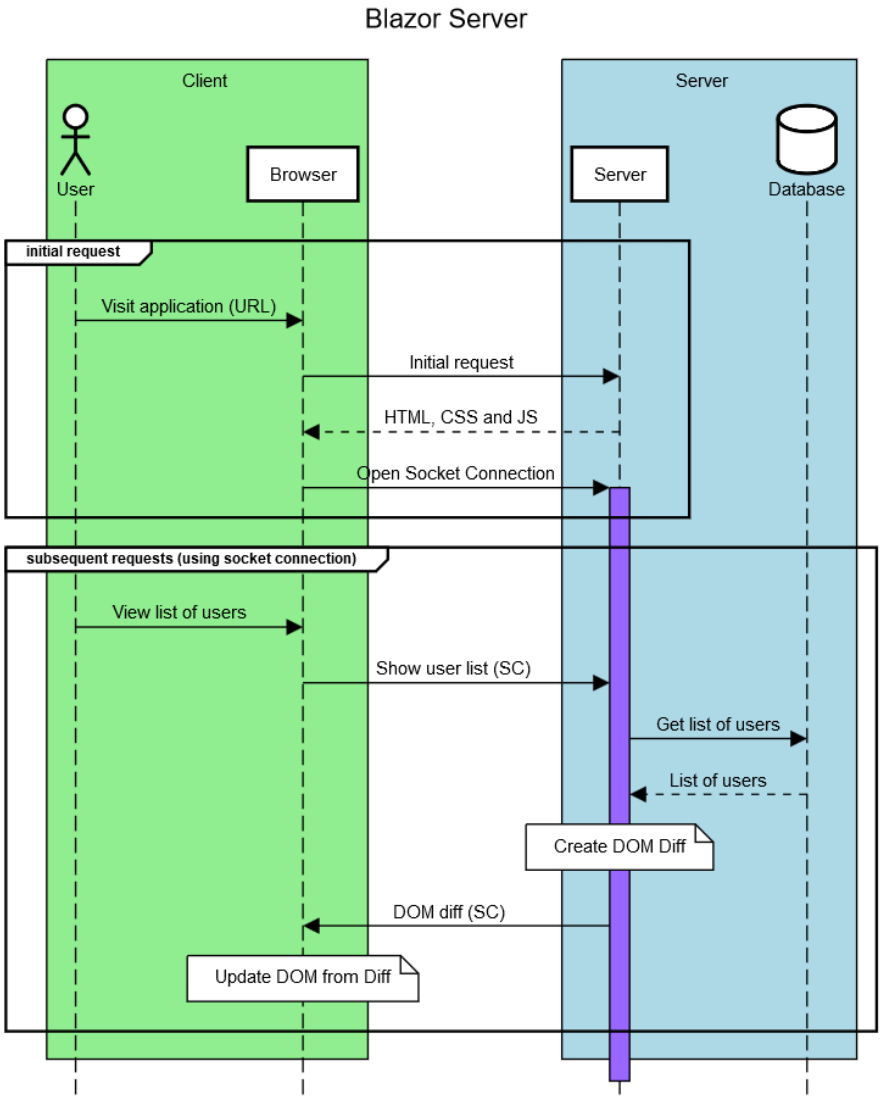
Příkaz provede registraci `Service` jako hostovanou službu. Podmínkou takových služeb je, aby implementovaly jednu ze tří možných rozhraní: `BackgroundService`, `IHostedService` nebo `IHostedLifeCycleService`.

Úlohy na pozadí využiju zejména pro správu turnajů, které bude možné naplánovat na pozdější spuštění. Je nutné aby na serveru běžela nějaká služba, která bude hlídat, jestli nastal požadovaný čas spuštění turnaje.

5.4 Webová vrstva

Webová vrstva obsahuje front-end webové stránky a její případný code-behind starající se o řízení stavu webové prezentace uživateli. ASP.NET má specifický svůj přístup ve vykreslování a dodávání dat klientskému zařízení. Přichází totiž ve dvou variantách, jako Web assembly nebo jako Server varianta.

Web assembly po načtení celá běží na klientském zařízení, klient si ze serveru stáhne runtime a data webových stránek a celou výslednu aplikaci provozuje na svém zařízení. Se serverem následně komunikuje pomocí REST API dotazů. Druhá varianta spočívá v hybridní komunikaci, kdy při prvotním dotazu klient získá nutné HTML, Javascript a CSS soubory k načtení základní stránky a poté otevře web socket spojení se serverem, odkud získává data. Výhoda tohoto přístupu je rychlost načtení a responzivita se serverem, protože má po celou dobu spojení s klientem otevřené webové sokety.



Obrázek 5.3: Schéma fungování Blazor serveru v hybridním klient/server režimu

Architekturu, kterou jsem se rozhodl použít ilustruje schéma převzaté od Jona Hiltona [4], v jeho blogu také dále popisuje i starší architektury.

Pro moje potřeby bude ideální volbou client/server rendering pomocí soketů, jelikož výhodu offline přístupu u WebAssembly nebude potřebná, Webový Manažer bude sloužit pro nahrávání brainů a čtení výsledků turnaje, jiná perzistatní data, co by bylo třeba zachovat na klientovi a případně tak ulevit serveru, nejsou.

5.4.1 Uživatelské účty

Blazor také ve svém základu nabízí řízení uživatelských účtů a ve verzi .NET 9 přichází s novou šablonou napsanou celou v Blazoru, včetně stránek s autentizací. Doposud tyto stránky byly psané v Razor pages, které používají tradiční poskytování obsahu klientovi na celé stránky a žádné webové sokety s ním neotevírají. Přístup se nazývá SSR (server-side rendering) a je to klíčová oblast, která je nutná k pochopení, aby autentizace fungovala tak, jak se očekává. U SSR přístupu je totiž dostupný `HttpContext`, pomocí kterého lze provádět

autentizaci uživatelů a přistupovat ke cookies na klientském zařízení. V praxi to znamená, že v Blazoru je nutné, aby stránky používaly atribut `[ExcludeFromInteractiveRouting]`. U návrhu webového manažera jsem se rozhodl vyzkoušet novou šablonu, která poskytuje základní infrastrukturu řízení uživatelských účtů. Konkrétní šablona je ve variantě MudBlazor šablony, používá tedy MudBlazor komponenty. Výsledkem je sjednocený design stránek pro autentizaci i stránek na práci s daty. Tento benefit byl pro mě nejdůležitějším při výsledném rozhodnutí.

Šablonu upravuji dále podle svých potřeb a konkrétně například poskytuji vlastní implementaci entit pro databází sloužící k autentizaci.

5.4.2 Realtime komunikace

Již v původní kapitole jsem zmiňoval, že Blazor nabízí webové sokety na real-time komunikaci s klientama. Na bázi webových soketů Microsoft poskytuje balíček SignalR, který pod pokličkou webové sokety používá. Nevylučuji možnost takových soketů použít k například informování o stavu probíhajícího nasazování brainů do kontejneru, nebo správu turnajů.

Kapitola 6

Realizace webového manažera

V této kapitole přiblížím implementaci webového manažera. Jeho vlastnosti, fungování, výslednou implementaci back-endu i fornt-endu a také rozeberu klíčové témata, které stojí za to přiblížit pro případné budoucí rozšíření systému.

6.1 Turnaje

Webový manažer podporuje tři druhy turnajů. Švýcarský systém a metodu každý s každým jsem rozebíral v kapitole 3.1.2. Oba druhy fungují na stejném principu jako v předchozích manažerech.

6.1.1 Vyřazovací systém na dvě porážky

Forma vychází ze skupinového systému o velikosti skupiny dvě (ve skupině proběhne jedno kolo zápasu všichni proti všem - tj. dva hráči se společně utkají v jednom zápase a ze skupiny postoupí vítězný hráč).
Implementace této metody je ve webovém manažeru následující: Na začátku turnaje jsou hráči zařazení do seznamu vítězných hráčů (W-bracket). Zahájí se turnaj a po výsledcích se všichni hráči, kteří prohráli přesunou z W-bracket na L-bracket (seznam první prohry). Oba seznamy zahájí další kolo turnaje. Po dalším kole se hráči, kteří prohráli ve W-bracket přesunou do L-bracket a hráči, kteří hráli v L-bracket a prohráli podruhé jsou z turnaje vyřazení trvale. Turnaj probíhá tak dlouho, dokud nezůstane jediný přeživší hráč ve W-bracket a v případě, že v L-bracket zůstane také jeden hráč, oba zbývající hráči se přesunou do společného seznamu a proběhne tzv. semifinále, při kterém je přesunutý hráč z původně W-bracket dovolený prohrát 1x a stále zůstat proti poslednímu hráči (protože doposud neprohrál), poté proběhne mezi dvěma poslednimi finále, kde se již vyhodnotí výsledky.

6.1.2 Řešení diskvalifikací

Webový manažer zavedl na poli turnajů oproti dvěma předchozím verzím nový způsob diskvalifikace při porušení pravidel při hře nebo nesprávné komunikaci. Při použití Webového Manažera v praxi nelze uvažovat, že všichni studenti budou mít odevzdané korektně fungující brainy. Původní verze manažerů se nijak nezmiňují o způsobech ověřování nebo diskvalifikace. Jedná se o tak o výraznou změnu v postupu, která vyžadovala i změnu způsobu, jak jsou pořádané turnaje.

Dohled nad korektní komunikací brainu zajišťuje libovolná registrovaná třída pod rozhraním `IBrainCommunication`. Potenciálně lze realizovat komunikaci i jiných druhů brainů mimo Docker implementací jiné třídy. Ve webovém manažeru je pouze implementované rozhraní pro komunikaci s brainy v dockeru. Další formy lze zaregistrovat v továrně `BrainCommunicationFactory`.

Dohled nad správným způsobem hry zajišťují třídy implementující `IGameValidation`, dále pak registrované a poskytované při smyčce zápasu pomocí továrny `GameValidationFactory`. U další metody hry je také nutné implementovat novou konstantu pro název hry ve třídě `DeskGameType`. Rozhodl jsem se unifikovat napříč projektem používané konstanty kvůli typové bezpečnosti při předávání druhu validátoru konkrétní hry a samotného textového řetězce, který se pak i dále používá při ověřování brainů, zda hru podporují.

V Manažeru je v této chvíli implementovaná jedna třída pro hraní piškvorek na 5 vítězných znaků v diagonální, vertikální nebo vodorovné ose, bez výhody začínajícího hráče.

6.1.3 Řešení lichých počtů

Liché počty se řeší pro každý druh turnaje mírně odlišně. Metoda každý s každým řeší lichý počet tak, že postupně každému hráči přidělí tzv. "HalfBye", tedy remízu a hráč se nebude účastnit aktuálního kola. Metoda na dvě porážky obdobně přiřadí "HalfBye". Švýcarský systém naopak vybírá hráče s nejmenším skóre a zároveň s nejmenším počtem přeskočených kol (bye rounds). Takovému hráči v prvním kole, rozřazovacím, započte plný bod, tj. 1 výhru (reprezentovanou v turnaji při přepočtu na 2 body). V dalších kolech získává hráč, co se přeskakuje pouze "HalfBye"stejně, jako ve zbylých dvou variantách turnaje.

6.1.4 Paralelní pořádání turnajů

Webový manažer umožňuje pořádání několika turnajů současně paralelně a zároveň všichni hráči v turnaji hrají zápasy paralelně mezi sebou. Jedná se tak o dvojitou paralelizaci. První předpoklad k předcházení konfliktů je zamezit, aby brain nebyl ve dvou současně probíhajících turnajích zároveň. Jednak podle kapitoly 5.1 považuji kontejner s brainem za perzistentní v době zápasu (v tomto kontextu to znamená, že manažer nemůže docker smazat, jelikož by to porušilo předpoklad) a druhá věc je samotné omezení databáze na jediný záznam reference id kontejneru[1]. Tuto kolizi brainů řeší Manažer na aplikační úrovni a zamezuje zaregistrovat konkrétní verzi brainu na více zápasů současně (přestože konfliktní registraci databázové schéma umožňuje).

Při spuštění turnaje se všechny registrované kontejnery s brainy z databáze podle předpokladu perzistence smažou a brainy se nasadí do kontejnerů znova tak, aby brainy začínaly ve stejných podmínkách bez výhod případného dodatečného tréninku z předchozích turnajů (systém totiž nezakazuje, aby se brain postupně účastnil několika turnajů). V průběhu dalších kol se potom kontejnery s brainy pouze restartují, aby měly počáteční stav[2].

[1] pokud by manažer držel druhé ID pouze v paměti, hrozilo by, že při nečekaném pádu serveru by kontejner zůstal trvale spuštěný a nasazený na hostovaném systému.

[2] I přes to, že původní komunikační protokol dovoluje zahájit několik her z jedné spuštěné instance. Toto pravidlo je spíše nápomocné studentům, protože budou mít jistotu, že jejich brain bude začínat každou hru na čistém počátečním stavu

6.1.5 Plánovač turnajů

Pro pořádání turnajů s odloženým počátkem bylo potřeba udělat plánovač, který by byl schopen na základě zadaného času počkat a následně spustit zvolený turnaj.

Při uvažování optimální implementace jsem vyloučil metodu aktivního čekání na nekonečné smyčce, která se používá u některých hardwarových řešení. Jednalo by se o plýtvání výpočetního výkonu. Lepší variantou by bylo udělat metodu tzv. pooling, kdy by služba periodicky kontrolovala všechny odložené turnaje. Ani tato varianta není ideální, přestože se zde neplýtvá výkonem. Je tady ale druhý problém, a to, jak krátký interval čekání má být? Při příliš krátkém intervalu by služba zbytečně obsazovala pracovní vlákno, které by jinak mohlo obsluhovat klientské sezení a při příliš dlouhém čekání by služba nebyla dostatečně responzivní a skutečně spuštěný čas by se lišil od toho původního.

Od hardwarové implementace jsem se ale vrátil přece jen zpátky na hardware a inspiroval jsem se u procesorů na moderních systémech, které využívají při plánování procesů přerušení a uspání.

Když se v běžícím procesu zavolá funkce `sleep()` případně obdobná funkce v C# `Thread.Sleep()`, operační systém dané jádro uspí na požadovanou dobu a předá řízení jinému procesu (resp. vláknu). Zároveň podle rozdílu jak dlouho je nutné ve funkci `sleep()` čekat vypočítá podle vnitřního čítače dobu, kdy má proces vyjmout z fronty uspaných procesů.

Třída `TaskScheduler` má obdobnou implementaci, namísto `Thread.Sleep()`, které je synchronní využívá asynchronní verzi `Task.Delay()`. Ve třídě se nacházejí dvě smyčky. První je na obsluhu a spouštění naplánovaných turnajů a druhá smyčka slouží na sledování stavu spuštěných turnajů a jejich vyjmutí ze seznamu běžících turnajů po dokončení (nebo selhání).

Velice zjednodušený pseudokód algoritmu hlavní smyčky vypadá asi takto:

```
while(true) {
    list toBeStartedTournaments
                = GetToBeStarted(scheduledTournamentsList);
    int newWait = schduledTournamentsList.GetFirstWithLowestDate();
    foreach(item in toBeStartedTournaments) {
            var task = RunTournament(item);
            runningTournaments.Add(task);
    }
     WaitForNext(newWait);
}
```

První problém spočívá ve vzájemném přístupu potenciálně několika vláken a může docházet souběh dat při přístupu k některé ze sdílených proměnných. Druhý problém nastává, pokud by z nějakého důvodu bylo potřeba předčasně ukončit nekonečnou smyčku. Server musí být koncipovaný tak, aby byl schopen v případě potřeby se sám nenásilně ukončit (tzv. graceful stop).

Na první problém je řešením zavést dvojici mutexů. První mutex zajistí výlučný přístup k seznamu s naplánovanými událostmi, aby se zamezilo změny kolekce v době, kdy dochází k vyjmutí turnajů, které mají začít běžet. Druhý mutex je na zajištění výlučného přístupu k seznamu již běžících turnajů, aby byl výlučný přístup pokud by externí kód potřeboval kontrolovat, zda turnaj stále běží. Druhý mutex teoreticky není tak potřebný, nicméně bylo nutné připravit takovou třídu na potenciální přístup z několika vláken současně.

Nutno podotknout, že je doporučené kvůli prevenci deadlocku takové mutexy při zamykání kritických částí zabalit do `try/finally` bloku, aby se i při vyvolání neočekávané výjimky zavolal `mutext.Release()`

Druhý problém pseudokódu řeší `CancellationToken` struktura, nacházející se v namespace
`System.Threading`. Při nastavení hodnoty tokenu na zrušení se na všech místech, kde se předala reference na token vyvolá výjimka přerušení. Výhodou je, že před vyvoláním výjimky má možnost konzument tokenu svoji práci dokončit nebo optimálně a bezpečně přerušit kontrolou atributu `token.IsCancellationRequested` a po uložení vyvolat výjimku `OperationCanceledException` zavoláním metody
`token.ThrowIfCancellationRequested()`

Způsob zrušení asynchronně probíhající práce pomocí tokenů je doporučená a hojně užívaná metoda v C#. Samotné tokeny na zrušení od `TournamentScheduler` propaguji až do samotné smyčky komunikace dvou brainů (třída `MatchRunner`) při zápase v turnaji. V případě, že by nastala potřeba turnaj ukončit, zprávu o přerušení dostanou všechny úlohy běžící na pozadí a během velmi krátké doby svoji činnost po případném zápisu do databáze okamžitě ukončí. Cestou zpátky při rozbalování řetězce volání (callstack) se takové výjimky zachycují, volitelně zpracují a předají dál.

Často přehlížená situace je u funkcí `Dispose()` a `DisposeAsync()`, které má
`TournamentScheduler` také "thread-safe". Při uvolňování prostředků, zejéma mutexů, probíhá atomická kontrola nastavené proměnné metodou Compare&Exchange, která jednak zajistí, že neproběhne uvolňování prostředků dvakrát, ale ani nedojde k neočekávaným výjimkám, které by nastaly v případě, že některé z vláken bude mít zamčený mutex u kritické operace. Na prevenci deadlocku plánovač disponuje ještě tokenem na indikaci probíhajícího uvolňování prostředků, tento signál je oznámený mutexům, které by zrovna čekaly na výlučný přístup do kritické části a vytvoří výjimku přerušení k uvolnění.

Po všech výše uvedených důvodech je optimální kritický kód zabalit do následujícího konstruktu:

```
try {
   await mutex.WaitAsync(disposingToken);
   // kriticky kod
} finally {
   if (Interlocked.CompareExchange(ref isDisposing, 0, 0) == 0) {
       mutex.Release();
   }
}
```

6.2 Vytvoření docker obrazu

Před samotným nasazení kontejnerů je třeba mít vytvořený v dockeru obraz, podle kterého se všechny kontejnery vytvářejí. Obraz se vytváří z dockerfile, který musí být specifikovaný v konfiguračním souboru Manažera. Jeho podoba může být různorodá, základní zahrnutý dockerfile pro nasazení brainů psané v C má tuto podobu:

```
FROM ubuntu:latest

# Install necessary packages
```

```
RUN apt-get update && \
    apt-get install -y make gcc g++ bash && \
    apt-get clean

# Default command to keep the container running or execute a script
CMD ["/bin/bash"]
```

Obraz vychází z obrazu Ubuntu. Pro Ubuntu jsem se rozhodl kvůli tomu, že u Alpine obrazu mi dělalo problém nasadit brain a korektně vytvořený kontejner spustit. Alpine je méně náročný a více očesaný než Ubuntu. Do obrazu ještě dále instaluji programy gcc, g++ a bash[3], které slouží pro překlad zdrojových souborů brainu do binárek.

Vzhledem k tomu, že operace nasazení image může být potenciálně dlouho trvající, rozhodl jsem se implementovat hostovanou službu běžící na pozadí Webového Manažera `DockerImageBackgroundService`. Tato třída dále implementuje rozhraní `IBrainContainerImageService`, pod tímto rozhraním je také registrovaná v DI pro dostupnost dalším službám v systému. Služba umí obraz sestavit, smazat, případně zkontrolovat jeho existenci. Na konkrétní realizaci stále používá `IDockerService` a tak její zodpovědností je samotný proces nasazování obrazu na pozadí, nikoliv samotná implementace nasazování.

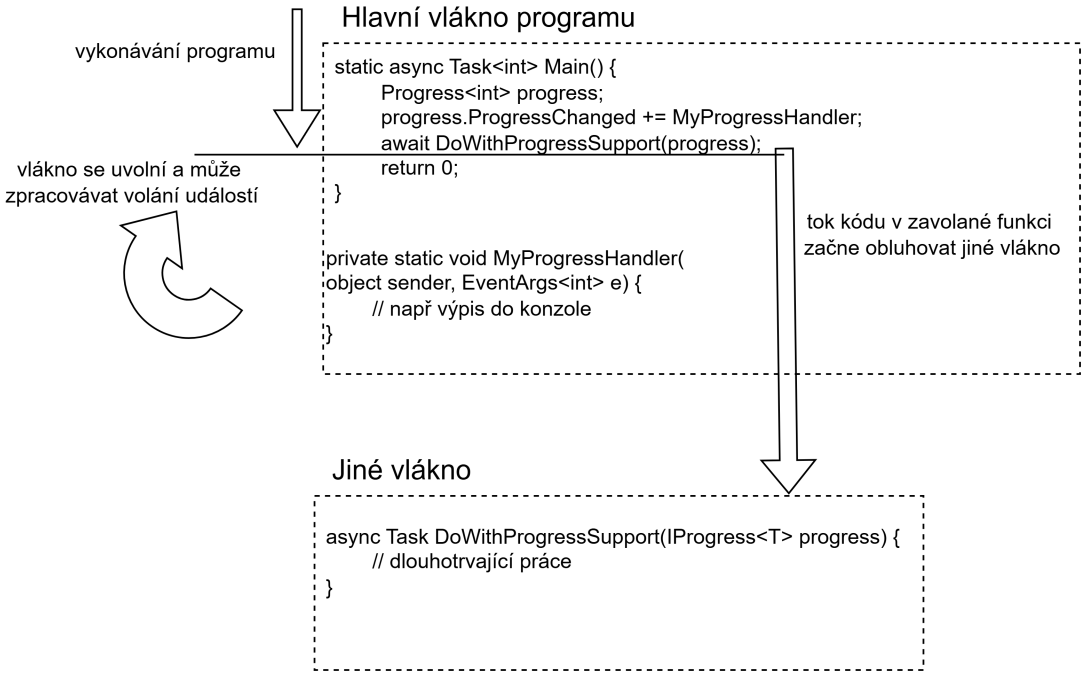
Vzhledem k tomu, že jsem si uvědomoval, že spousta úkonů se bude dotazovat této služby na existenci obrazu (protože bez obrazu de-facto nelze provádět téměř žádné operace ve Webovém Manažeru), je také po boku plánovače turnajů implementovaná s výlučným přístupem k funkcím. Na rozdíl od plánovače má mnohem jednodušší implementaci, protože v Docker prostředí může existovat pouze jediný obraz se stejným názvem[4] a tak postačí kontrolovat výlučný přístup k jediné proměnné s referencí na spuštěnou službu, namísto kolekcí.

Proces nasazování Docker obrazu může být časově náročný, při nasazování jsem měl za cíl poskytnout učiteli zpětnou vazbu, proto vedle samotného nasazování poskytuje služba zpětnou vazbu o postupu obrazu. Knihovna Docker.DotNet při vytváření obrazu používá `IProgress<T>`, což je běžně používaný způsob u jednoduchých aplikací na získávání zpětné vazby. Knihovna poskytuje informace o výstupu z příkazové řádky při nasazování obrazu, ale také aktuální stav operace (například rozbalování, stahování) a vedle toho i číslo, které se inkrementuje při postupu.

Konzument volající funkci s podporou krokování pomocí třídy `Progress<T>` (toto již není rozhraní, ale implementace, v názvu již nemá předponu "I") si vytvoří instanci, od které si registruje event handler. Handler má podobu delegátu funkce a v C# je podporovaný pomocí klíčového slova `event`. Při zavolání funkce `IProgress<T>.Report(...)` se zavolají všechny registrované funkce pod událostí instance `Progress<T>`.

[3] bash by nebyl zřejmě potřeba, protože se v Ubuntu nachází, ale pro jistotu funkčnosti jsem ho přece jen zahrnul do dockerfile

[4] v kombinaci tagů lze mít různé verze Docker obrazů, avšak s tagy webový manažer neumí pracovat



Obrázek 6.1: Schéma zpracování předávání řízení mezi vlákny při asynchronních operacích

Na obrázku 6.1 je schéma znázorněné. Použití tohoto řešení by bylo dostačující, pokud by program nebyl postavený na webové technologii. Pokud by front-end byl postavený na jiné technologií než je Blazor, tohle řešení nelze použít. Hostovaná `DockerImageBackgroundService` namísto toho používá hub pro real-time komunikaci. Výhoda tohoto řešení je nezávislost konzumenta, protože se jedná o univerzální technologii. Webová stránka tímto způsobem může zachytávat příchozí data a celý systém není na sebe vázaný. Jinými slovy lze front-end vyměnit za jinou technologii a stále bude možné využít back-end Manažera na získávání postupu.

Konkrétní implementaci hubu zaopatřuje třída `DockerLogHub` a poskytuje pro klienta funkce z hubu, od kterých může naslouchat o změně stavu nasazování image:

```
[HubMethodName(LogMessagesNames.RECEIVE_IMAGE_PROGRESS)]
Task ReceiveImageProgress(JSONMessage mesage);

[HubMethodName(LogMessagesNames.IMAGE_DEPLOYMENT_COMPLETE)]
Task ImageDeploymentComplete();

[HubMethodName(LogMessagesNames.IMAGE_DEPLOYMENT_FAIL)]
Task ImageDeploymentFailed(Exception mesage);
```

Díky tomu může být webová stránka responzivní a reagovat na probíhající změny u nasazování. U webové stránky stačí v code-behind[5] implementovat hub klienta a registrovat obsluhující metody:

[5]Jedná se o část kódu, která obsluhuje elementy webové stránky

```
_hub.On<JSONMessage>(
LogMessagesNames.RECEIVE_IMAGE_PROGRESS,
ProgressRecievedCallback);

_hub.On<Exception>(
LogMessagesNames.IMAGE_DEPLOYMENT_FAIL,
ImageFailedCallback);

_hub.On(
LogMessagesNames.IMAGE_DEPLOYMENT_COMPLETE,
ImageCompletedCallback);
```

Jelikož klientské funkce hubu akceptují textový řetězec názvu funkce, vytvořil jsem pro sjednocení konstanty názvů jednotlivých funkcí, aby se předešlo chybám v případě, že by se potřebovalo manipulovat s názvy funkcí.

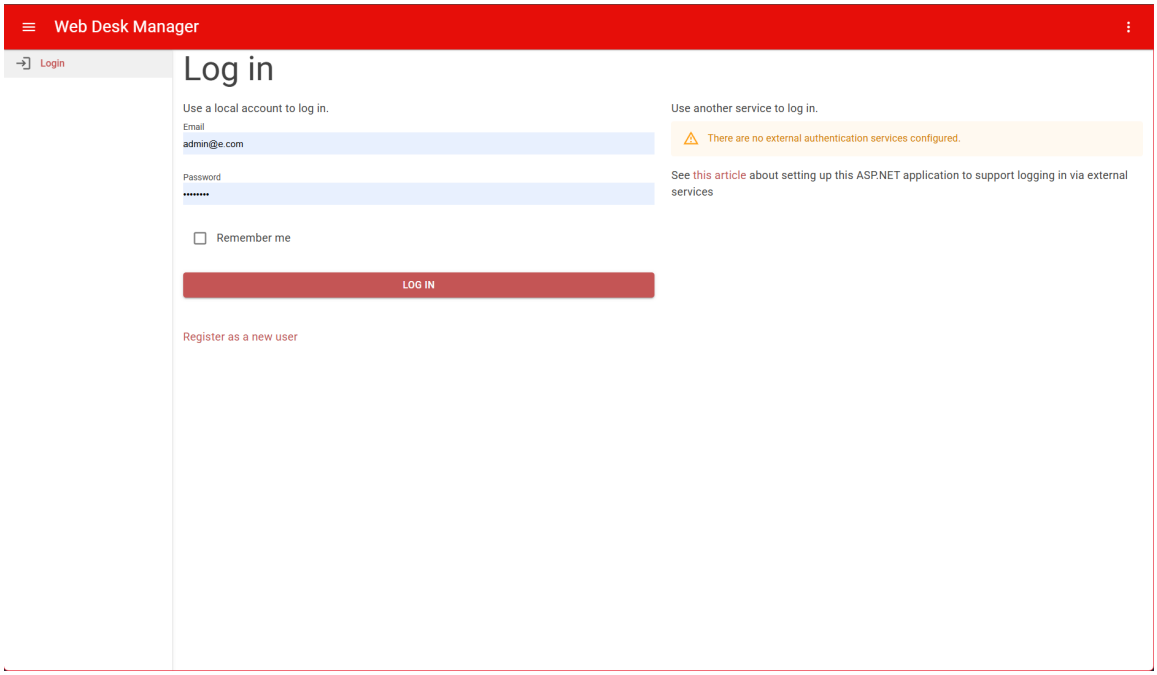
6.3 Uživatelské rozhraní

Při vytváření uživatelského rozhraní ve webovém prostředí jsem se snažil užívat vhodné praktiky na přehledný a srozumitelný uživatelský zážitek. Mezi takové předpoklady lze zařadit mimo jiné:

- responzivní rozhraní
- validace uživatelského vstupu
- zpětná vazba při akci
- optimální barevné rozložení pro správný kontrast a čitelnost

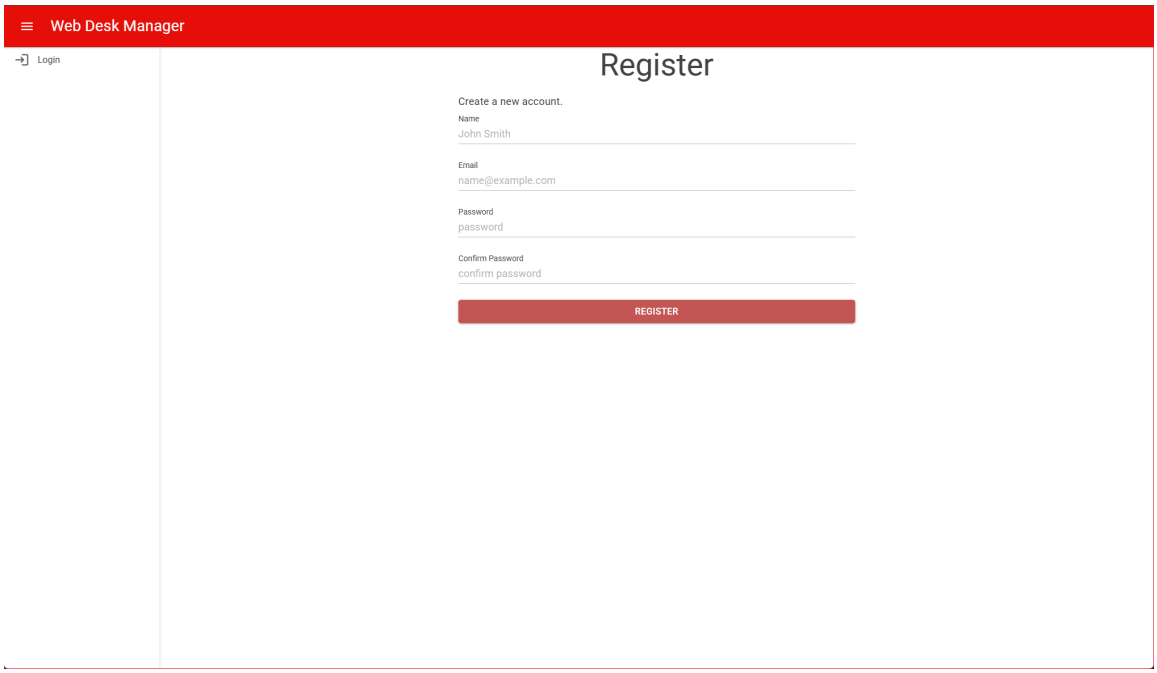
6.3.1 Přihlášení a správa účtu

První obrazovka, kterou Webový Manažer uživatele přivítá je přihlašovací obrazovka:



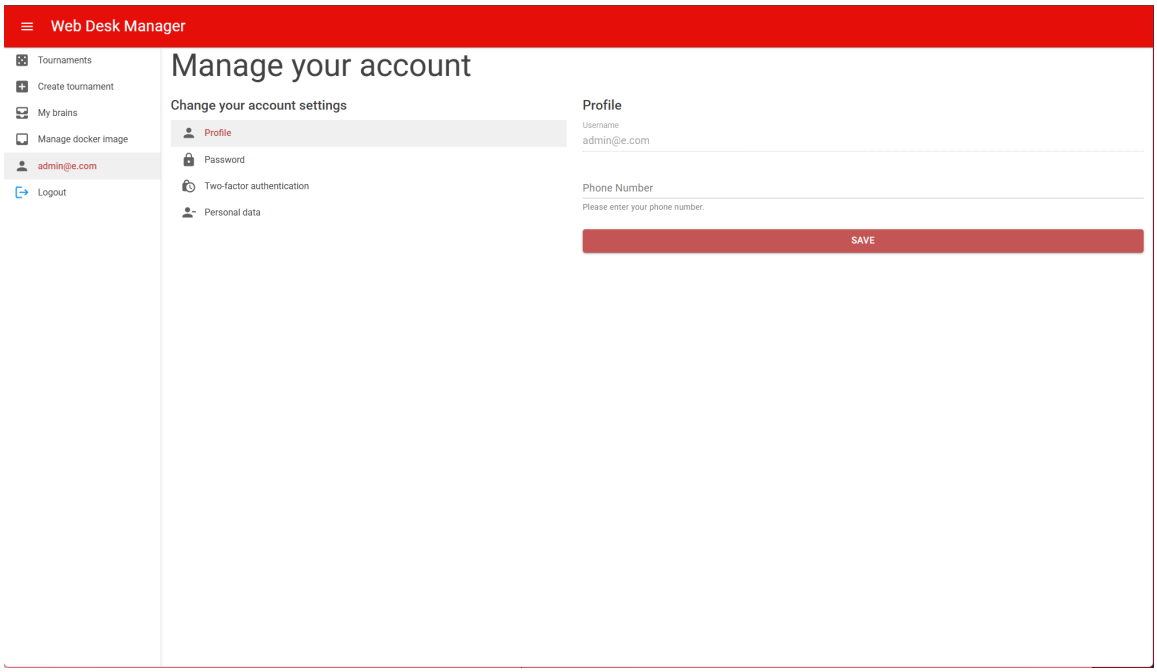
Obrázek 6.2: Přihlašovací stránka webového manažera

Záměrně jsem nechat v pravé části místo pro možnost přihlásit se přes externí poskytovatele, například pomocí JWT tokenu. Jako možné rozšíření manažera může být skutečné nasazení na školní server a propojit přihlášení s přihlášením přes systém VUT. Pokud nemá uživatel účet, může si ho založit na registrační stránce.



Obrázek 6.3: Stránka s registrací u webového manažera

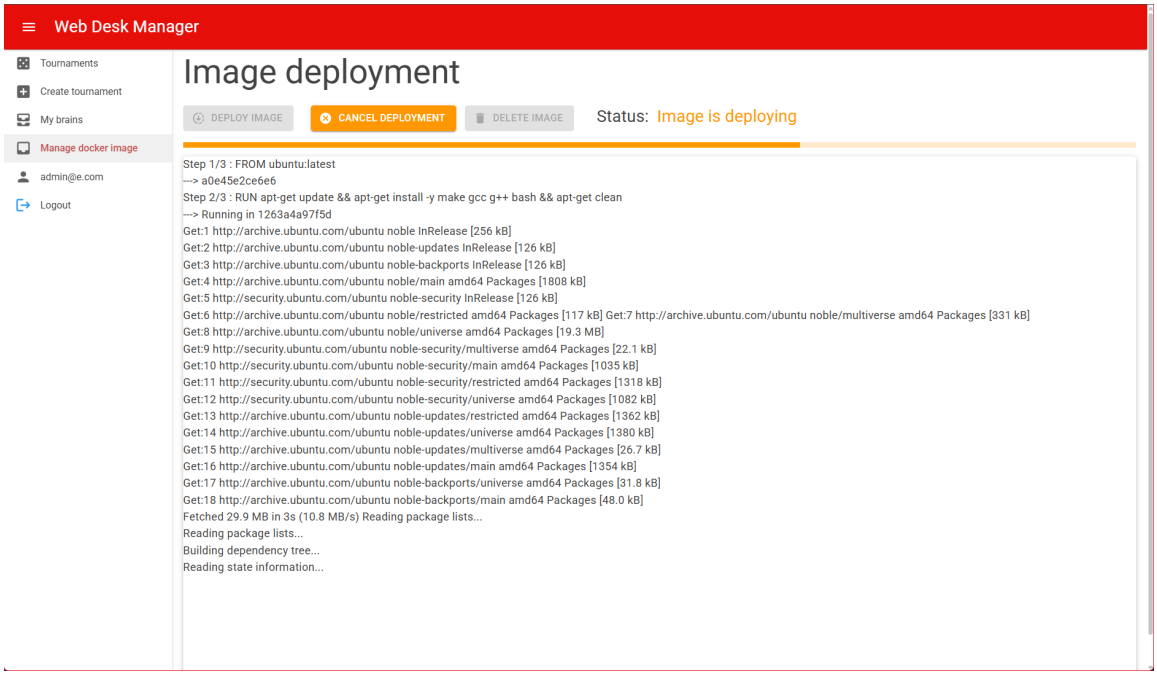
Po přihlášení může uživatel přistoupit do nastavení účtu, odkud má možnost si změnit své dosavadní heslo, stáhnout si svoje osobní data (ty zahrnují prakticky jen jméno a email), účet odtud může smazat. Dvoufaktorová autentizace je funkční, jelikož nevyžaduje žádné externí služby na to, aby mohla fungovat.



Obrázek 6.4: Stránka se správou uživatelského účtu

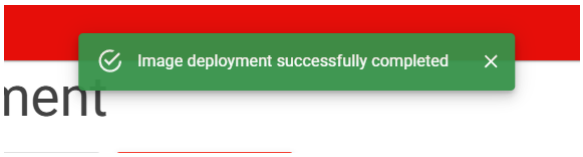
6.3.2 Správa Docker obrazu

V případě uživatelského účtu s administrátorským oprávněním má uživatel možnost vstoupit do stránky se správou docker obrazu, odkud může nasadit a sledovat v reálném čase, jak nasazování probíhá.



Obrázek 6.5: Stránka se správou Docker obrazu s možností sledovat reálný výstup konzole

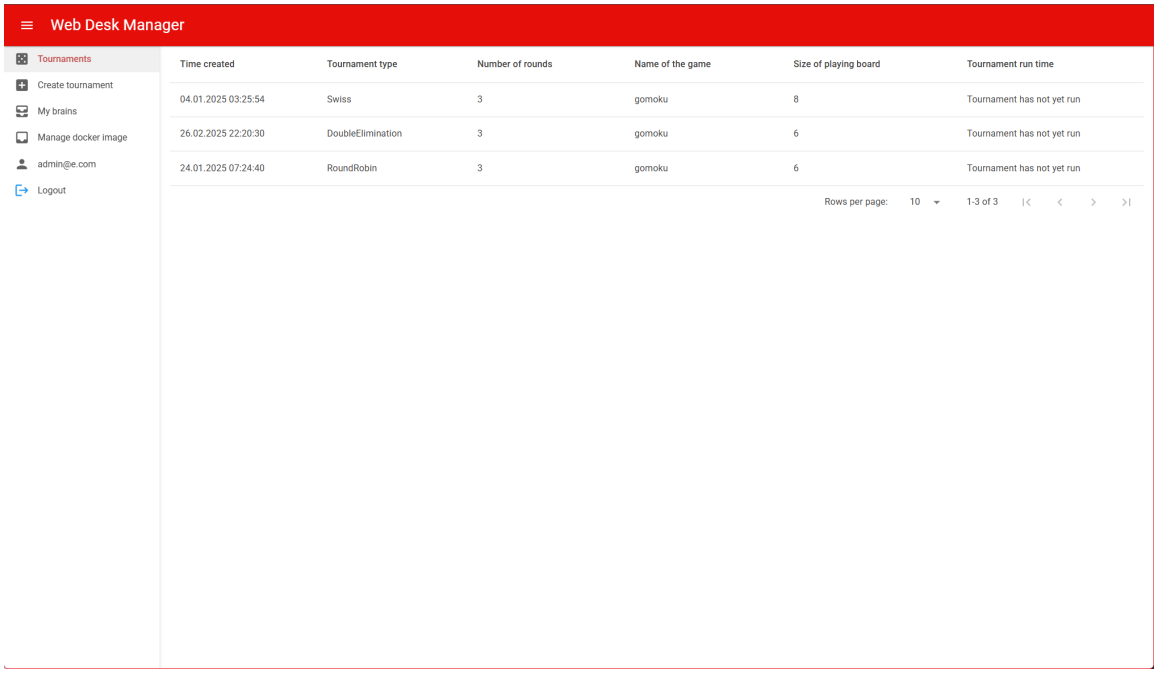
V případě bezproblémového nasazení Docker obrazu se uživateli zobrazí v horní části stránky hláška potvrzující úspěšné nasazení.



Obrázek 6.6: Hláška signalizující úspěšné nasazení Docker obrazu

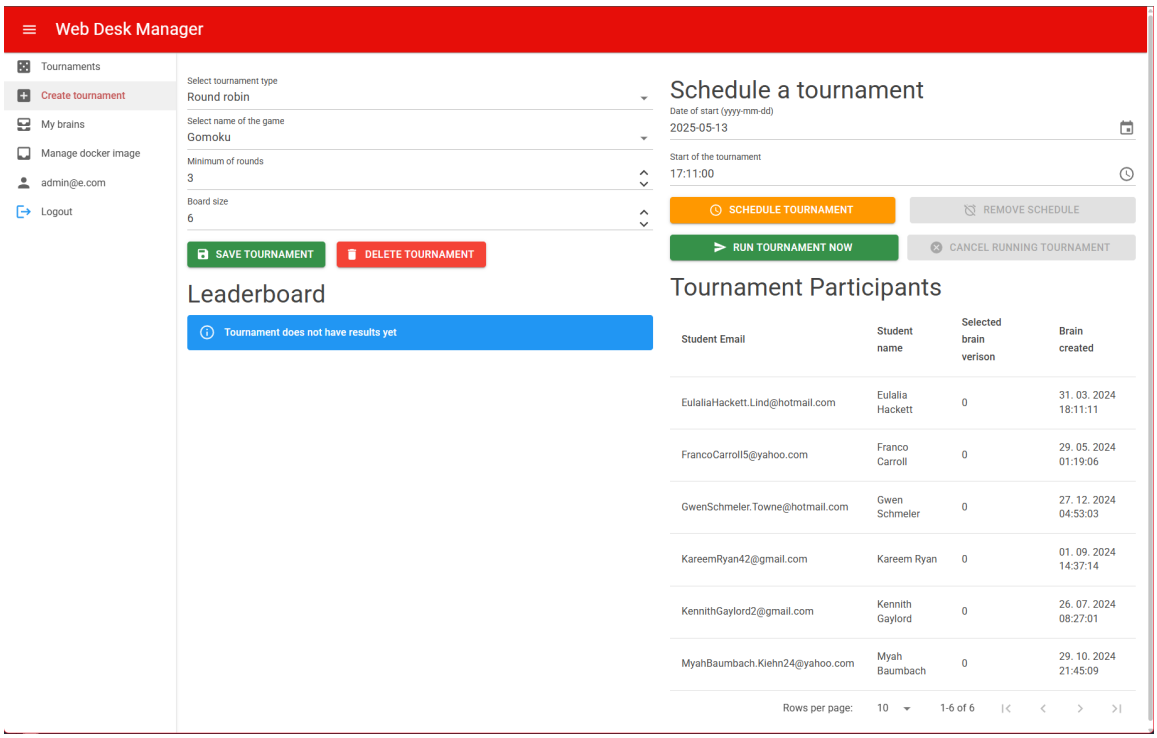
6.3.3 Pořádání turnajů

Administrátorský účet dále vidí seznam svých vytvořených turnajů. Z této stránky turnaje nelze smazat, jelikož se tento krok dělá v samotném detailu každého turnaje. Je to z toho důvodu, že pokud by turnaj běžel, nebude možné turnaj smazat.



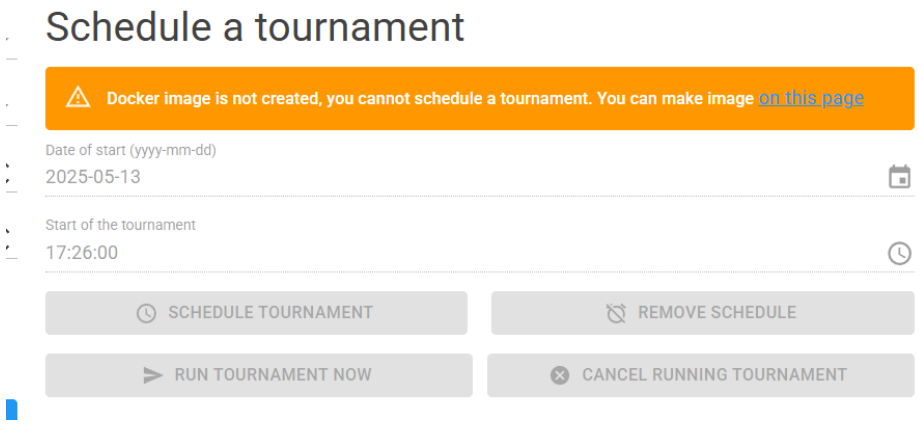
Obrázek 6.7: Stránka se seznamem turnajů

Při prvotním vytvoření turnaje nejsou dostupné výsledky, o této skutečnosti je uživateli zobrazená informační hláška v místě tabulky s výsledky. Po pravé straně lze turnaj naplánovat a vidět seznam účastníků turnaje.



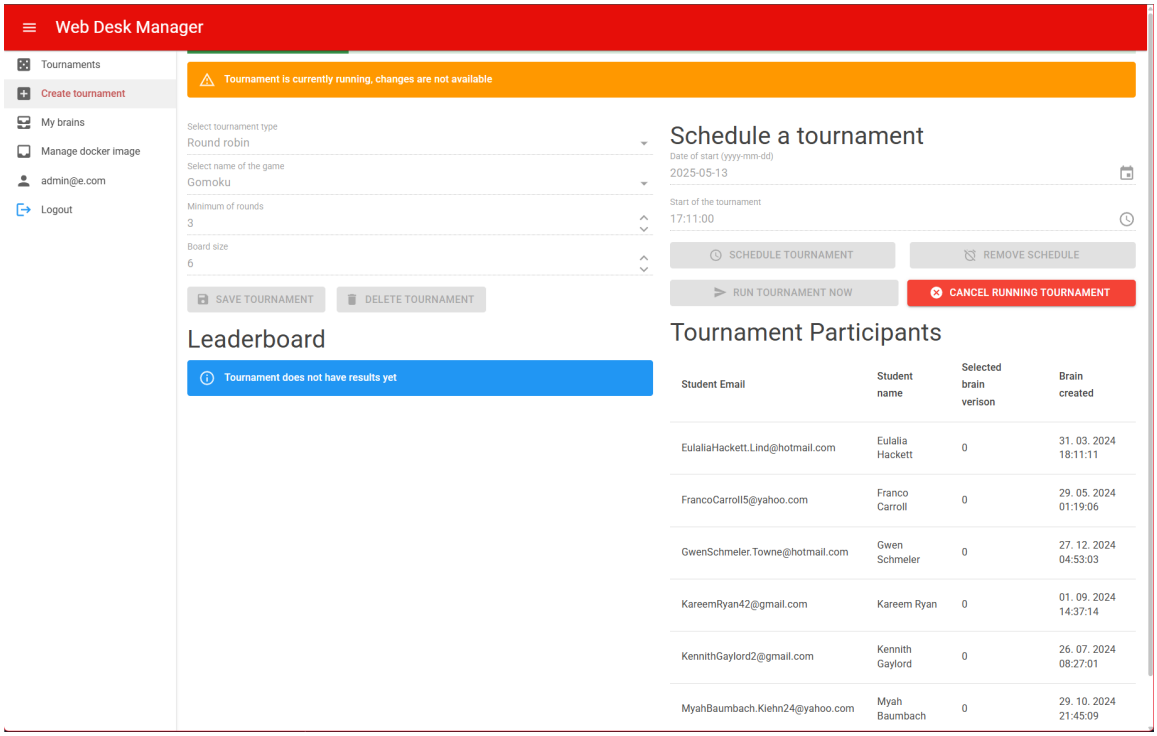
Obrázek 6.8: Stránka s detailem turnaje před spuštěním turnaje

Pokud uživatel zapomene vytvořit Docker obraz, stránka nedovolí turnaj spustit a zobrazí upozornění, pokud by se uživatel pokusil turnaj spustit. Absence Docker obrazu nicméně neomezuje uživatele, aby si turnaje předem vytvořil a nastavení uložil.



Obrázek 6.9: Chybová hláška zamezující spuštění turnaje, pokud chybí Docker obraz

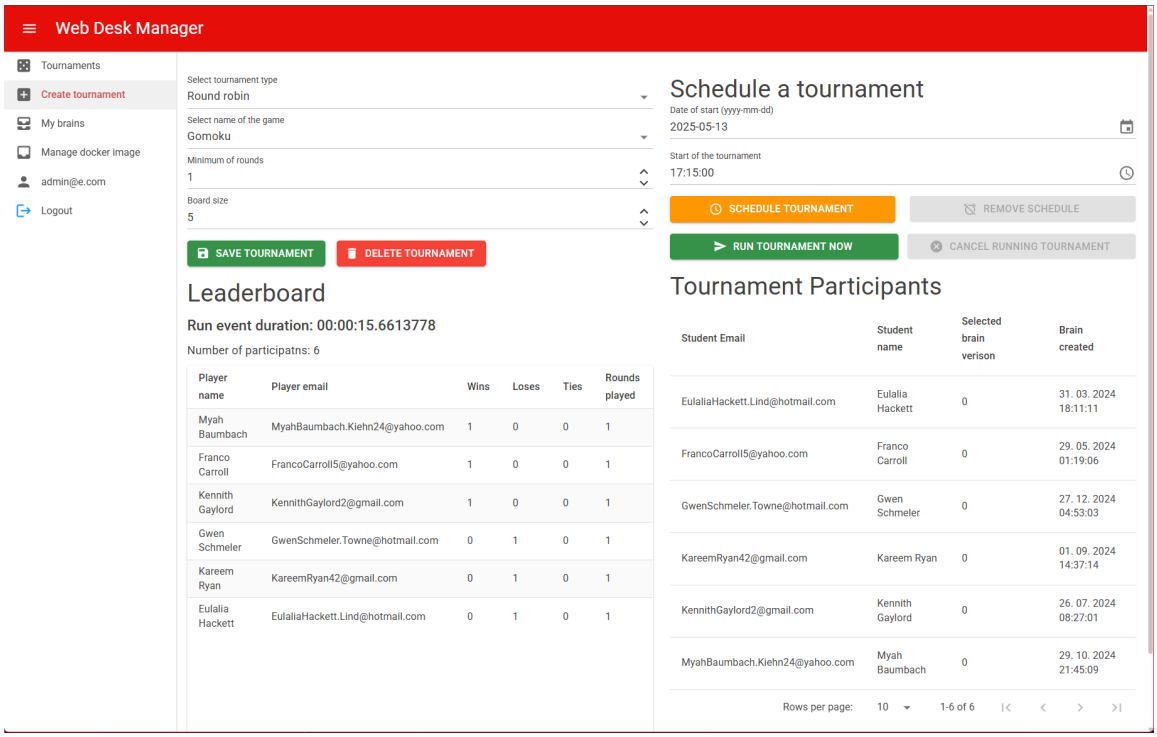
Detail stránky s turnajem je v případě aktuálně probíhajícího turnaje zamčené a je dostupné pouze pro čtení. O této skutečnosti upozorňuje svítící žlutý baner a horní část stránky je doprovázená animací.



Obrázek 6.10: Zamčený detail turnaje v případě probíhajícího turnaje

Na obrázku 6.10 je patrné, že jediná dovolená akce v případě běhu turnaje je turnaj zastavit.

Jakmile probíhající turnaj skončí, uživateli se zpřístupní pohled na výsledky jednotlivých brainů s počtem účastníků a celkovou dobou běhu turnaje.

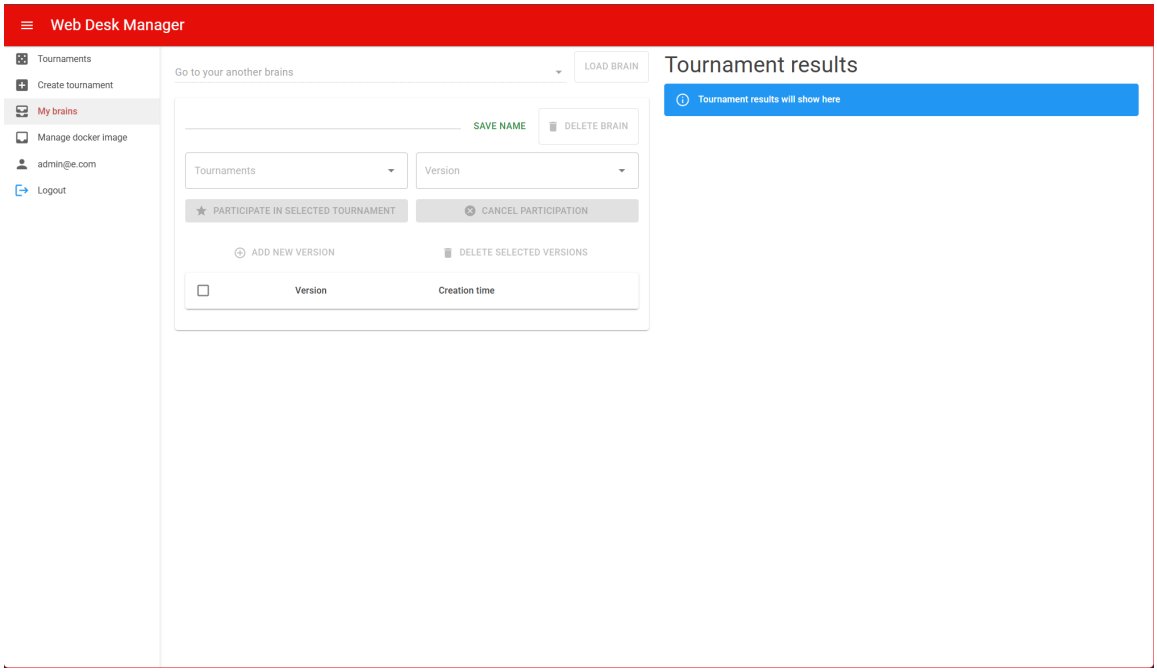


Obrázek 6.11: Výsledky turnaje z pohledu organizující osoby

Uživatel nemá zakázané turnaj opětovně pustit. Tohle rozhodnutí jsem učinil z toho důvodu, že v případě počítačových programů nedochází podobně jak u lidí k vyčerpání, a proto nezáleží, kolikrát turnaj poběží.

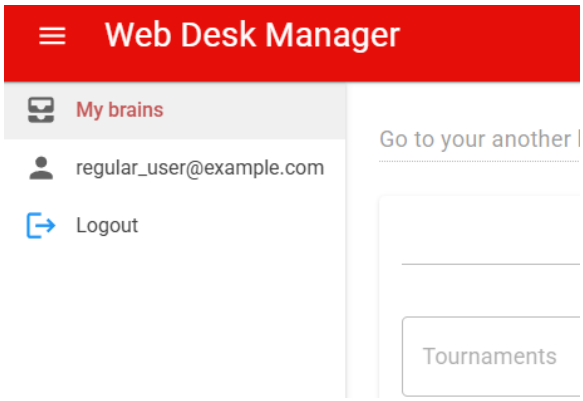
6.3.4 Správa brainů uživatele

Každý uživatel systému (včetně učitele) má přístup ke správě svých vlastních brainů. Systém dokonce umožňuje, aby učitel (administrátor) zorganizoval turnaj a poté se turnaje účastnil svými vlastními brainy. Pro tuhle možnost nebyla třeba nijak zasahovat do dosavadní logiky kódu a ani jsem nepovažoval za nutné tohle pro učitele omezovat. Případ užití této volby je vytvořit ve studentech jistou motivaci a soutěžit, který ze studentů dokáže překonat "standardizovanou"laťku poražením vzorové umělé inteligence učitele. Jako další případ užití vidím v tom, že učitel může zorganizovat neoficiální turnaj, kde nasadí své umělé inteligence a studenti budou mít možnost si nanečisto vyzkoušet, jestli je jejich implementace brainu dostatečně inteligentní.



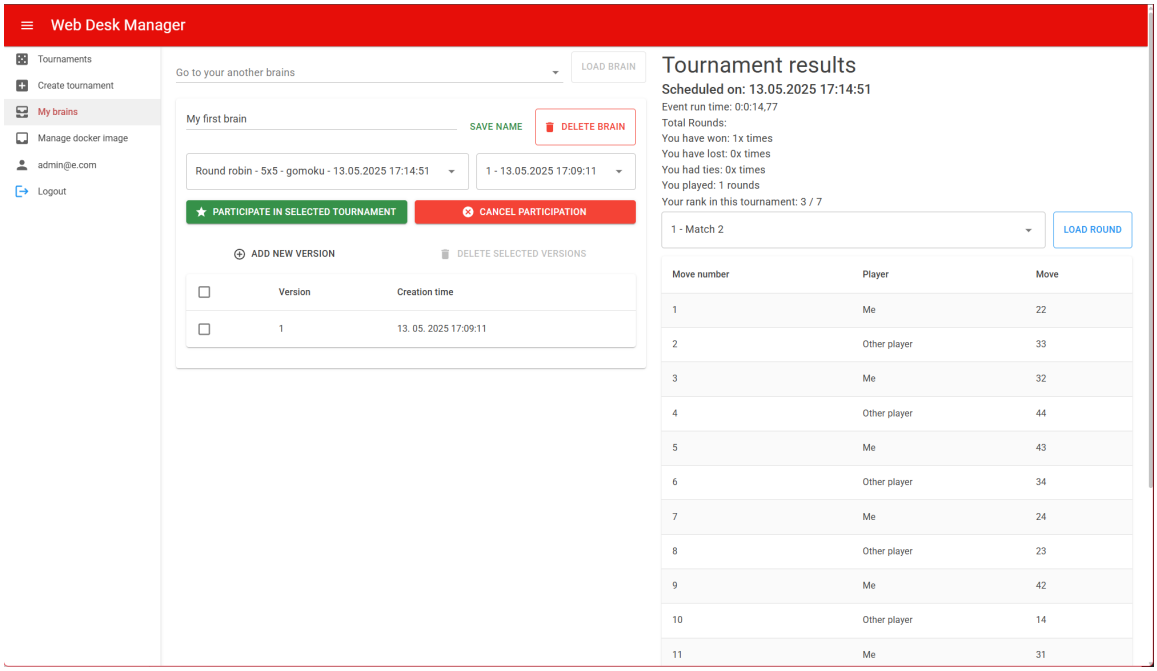
Obrázek 6.12: Stránka správy brainů a jejich verzí

Na stránce na obrázku 6.12 má uživatel nejen možnost upravit svůj brain, ale také nahrát další verzi brainu a účastnit se zvoleného turnaje ze seznamu turnajů. Pro studenty je tato stránka hlavním bodem, kdy budou s manažerem interagovat.



Obrázek 6.13: Navigační panel, který vidí studenti s běžným uživatelským oprávněním

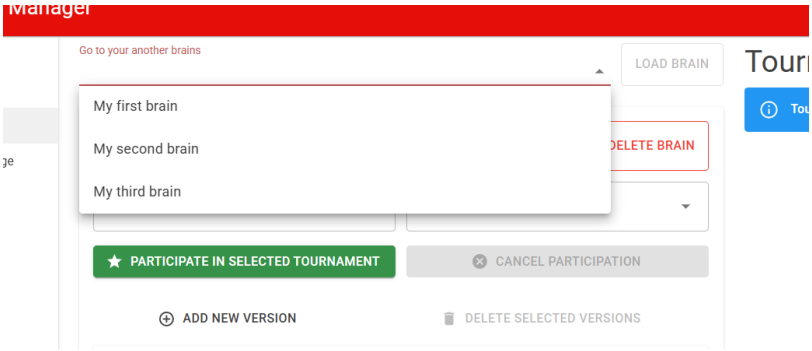
Dle obrázku 6.13 je patrné, že studenti mají řádově omezenější přístup do různých aspektů systému.



Obrázek 6.14: Zobrazení výsledků turnaje z pohledu uživatele

Po skončení turnaje může každý zúčastněný uživatel se podívat, jak si v turnaji vedl. Má možnost si i zobrazit tahy v jednotlivých zápasech. Z důvodu ochrany soukromí jsou jména brainů skrytá a zobrazuje se pouze obecný název "můj"a "cizí"brain.

Podobně jako je obrazovka s turnajem 6.10 zamčená při probíhajícím turnaji, je i obrazovka detailu specifického brainu zamčená proti úpravám. A protože přehled s brainy slouží zároveň jako seznam všech brainů uživatele, zamyká se v případě, že uživatel zvolí brain, který se účastní s vybranou verzí turnaje. Při návštěvě přehledu s brainy má uživatel možnost si zvolený brain vybrat podle výběrového okna na obrázku 6.15.

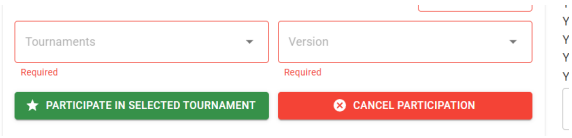


Obrázek 6.15: Výběr brainů uživatele v přehledu brainů, zobrazuje se vždy zvolené jméno brainu, který si uživatel nastavil

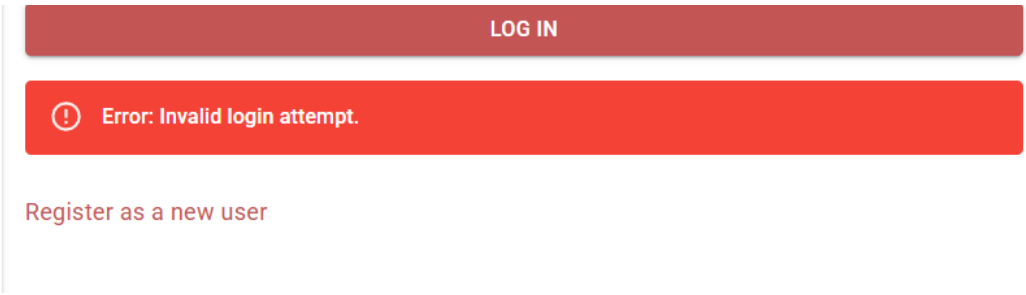
6.3.5 Validace uživatelského vstupu

Webové formuláře disponují určitou úrovní validace vstupu, aby uživatel neměl možnost zavést neplatné data do databáze. Neplatné data jsou uživateli prezentovány ve formě plo-

voucího okna v horní části obrazovky, zvýrazněním elementů, kde se nachází neplatné data, nebo výstražných banerů.



Obrázek 6.16: Upozornění na neplatné data u formuláře



Obrázek 6.17: Hláška upozorňující na neplatné přihlašovací údaje

Kapitola 7

Testování

Testování probíhalo především na vlastním počítači s procesorem Ryzen 9 5950X a 32GB RAM. Cílem testování bylo zaznamenat, jak náročný bude server na potenciálním systému, pokud by se přes něj pořádaly turnaje. Velkou roli v náročnosti na paměť jednoho kontejneru hraje zvolený základní systém. Jistě se bude byť v řádech pár megabajtů lišit využití paměti Alpine Linuxu a jinak bude zabírat paměť Ubuntu. V absolutním měřítku jsou jednotky megabajtů zanedbatelné, ale v součtu při nasazených kontejnerch u turnaje tento nepatrný rozdíl může narůst ve velký problém. Dále je také důležitou metrikou jakým způsobem a jak efektivně dokáže turnajová partie využít dostupný procesor.

7.1 Seedování databáze

Z práce Jana Kouřila mi byly poskytnuté brainy na testovování systému. Vzhledem k tomu, že Webový Manažer podporuje v této chvíli jen piškvorky, ostatní brainy jsem nepoužil. V prvé řadě bylo potřeba vytvořit data v databázi k tomu, aby se takový turnaj mohl uskutečnit. Pomocí tzv. seed metod jsem vytvořil duplikát brainu gomoku pro verzi Linux a postupně jsem podle požadovaného počtu vytvořil uživatele, brainy a verze brainů, všechny relace jsem napojil na vytvořený turnaj a brainy také zkopíroval do adresáře do složek, kde by se jinak za normálních okolností uložily.

Aby i testování bylo jednoduše replikovatelné jinými lidmi. Vytvořil jsem v konfiguračním souboru klíč "seeding", který když se nastaví na true vytvoří v databázích falešné testovací data na ilustraci funkčnosti celého systému. Pokud tato možnost bude vypnutá, bude systém začínat v prvopočátku s prázdnou databázi. V takové situaci bude prvně registrovaným uživatelem administrátor.

Při testování vzhledu jednotlivých prvků webových stránek jsem ještě využil program Bogus[1], který mi pomocí jednoduše specifikovaných pravidel pro jednotlivé atributy entit vytvořil falešné data na atributech jméno, email, datum a další.

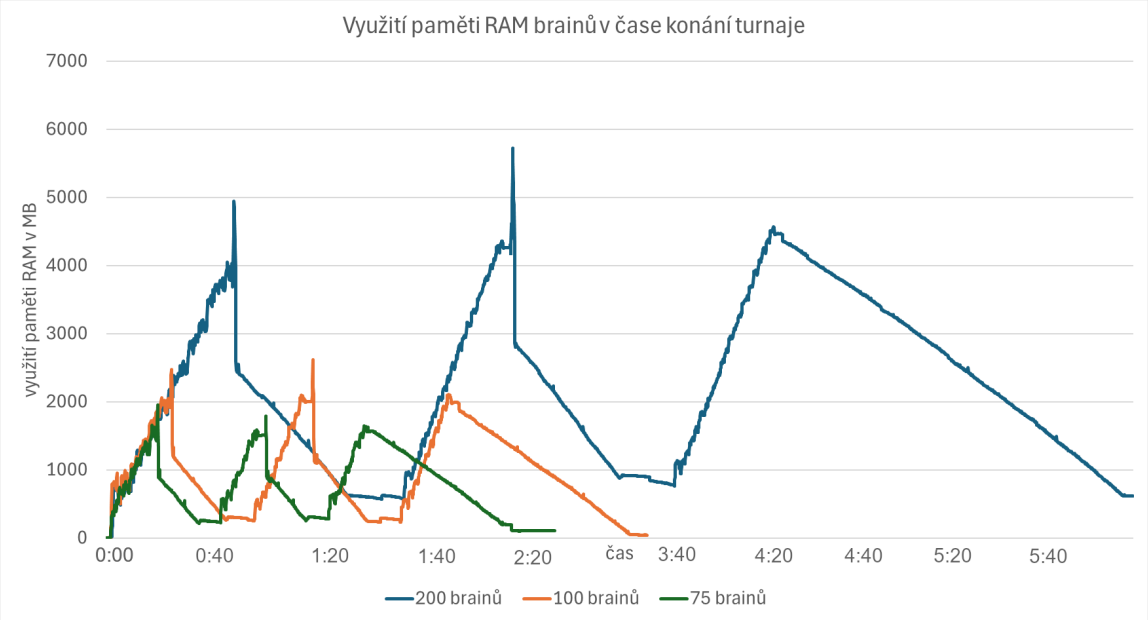
7.2 Testování využití paměti

První typ testu sleduje celkovou paměť systému. Není důležitá délka zápasu, proto herní plocha byla zvolená na nejmenší možnou a byl zvolený způsob každý s každým. V průběhu trvání sleduji využití paměti s intervalem 100ms. Na měření paměti jsem si vytvořil jednoduchý skript, který extrahuje data z příkazu `free` a dále pomocí `awk` extrahuje pouze

[1] Dostupný na: https://github.com/bchavez/Bogus

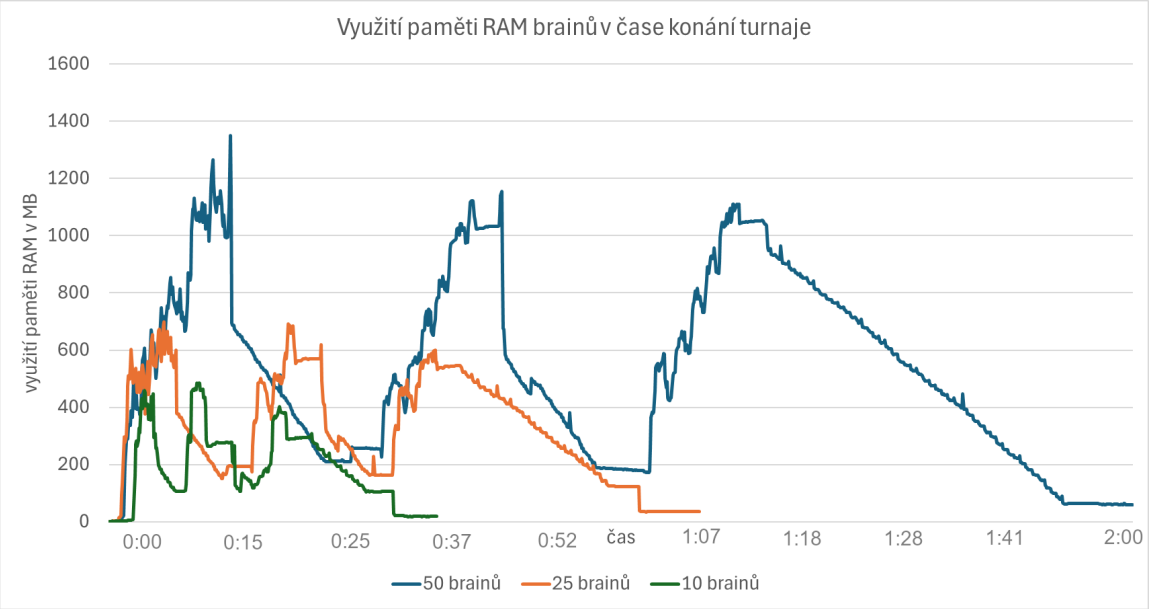
číslo. Výsledný program ve smyčce vypisuje na standardní výstup, odkud přesměrovávám hodnoty do pracovního souboru na pozdější zpracování:

```
#!/bin/bash
while true; do
    free -m | awk '/^Mem:/ {print $3}'
    sleep 0.1
done
```



Obrázek 7.1: Graf využití paměti při pořádání turnaje pro vybrané sety brainů - horní polovina

Jednotlivé grafy jsem rozdělil, aby testové sady s nižším počtem brainů šly přehledně vidět.



Obrázek 7.2: Graf využití paměti při pořádání turnaje pro vybrané sety brainů - spodní polovina

Při souhrnné analýze všech sad testů pozoruji stejný vzor využití paměti. První prudká špička je způsobená postupným nasazování brainů ve shlucích. Tento přístup byl nutný udělat, protože předchozí oddělené testování vykázalo omezení docker enginu nebo knihovny Docker.DotNet obsluhovat velké množství paralelních dotazů na kontejnery. Při experimentech jsem totiž zjistil, že u procedury vytvořit kontejner, vytvořit brain archiv, zkopírovat do kontejneru a zkompilovat mi po chvíli začalo velké množství připojení na docker soket se ukončovat, nebo selhávat vlivem vypršení časového limitu na odpověď. Při hlubším zkoumání příčiny nepomohlo ani vytvořit na každý brain vlastní klientské připojení na socket, domnívám se tedy, že docker není schopen obsluhovat velké množství paralelních dotazů (jedná se počtem o stovky paralelních spojení, ale jednotky selhání spojení vznikalo už při první překonané padesátce spojení). Moje domněnka byla omezit otestovaný počet souběžných spojení a nasazování brain kontejnerů na počet kolem 32. Nahodile číslo 32 reprezentuje i počet vláken, kterým můj procesor disponuje. V implementaci jsem proto zanechal výpočet velikosti paralelní dávky na maximální hodnotu 32, nebo nižší podle počtu vláken na instalovaném systému.

Dále lze pozorovat, že doba využití největšího množství paměti je velmi krátká, domnívám společně z logů z konzole serveru, že nejdelší dobu turnaj stráví na zápisech do databáze u kterých všechny zápasy po konci zapisují hromadně uložená data z pracovní paměti. Nedovedu si však vysvětlit tři špičky, když kola probíhaly dva, moje domněnka je, že jednu špičku způsobuje pomalé zpracování obsluhy zastavování a restartování kontejnerů. Po mnohem pozdějším experimentování se spouštěním velkého množství brainů (počtem 200 a více) jsem narazil na problém vypršení časového limitu na odpověď instance zodpovědné za komunikaci s docker engine. Postupné nasazování v dávkách je totiž v manažeru implementované jen při vytváření kontejneru a kopírování souborů do něj, tato činnost se děje jen na začátku turnaje a následné restarty kontejnerů mezi koly probíhají všechny paralelně a jsou limitované rychlostí obsluhy docker engine. Pozvolné klesání na konci průběhu

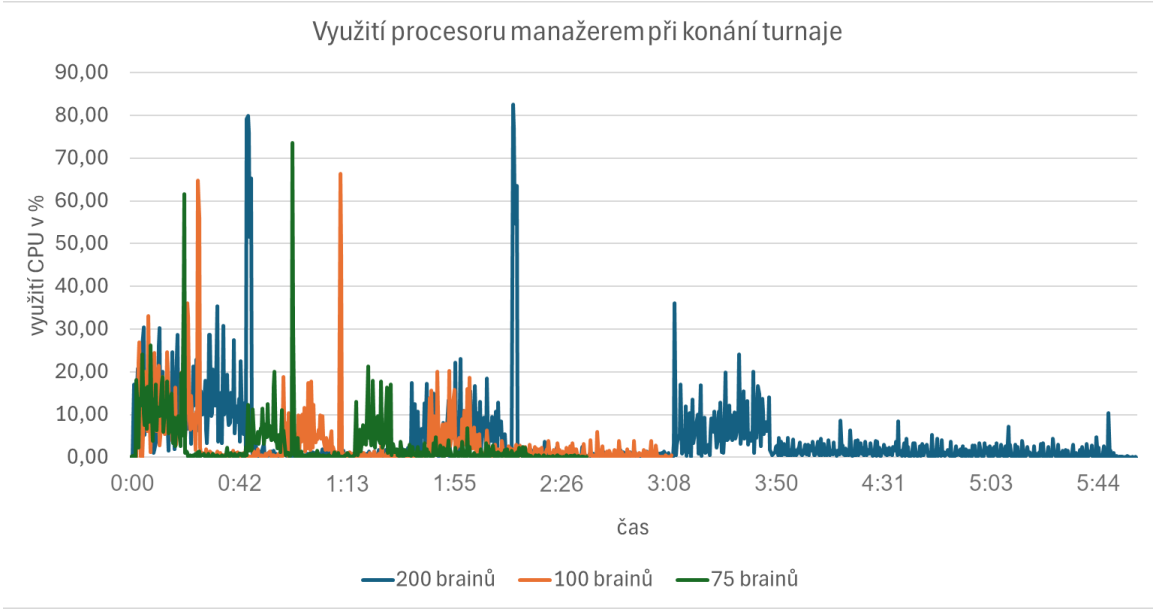
turnaje přisuzuji k malému časovému oknu každé instance `BrainCommunictaion` protože o časové okno s docker engine bojuje společně se všemi ostatními instancemi.

7.3 Testování náročnosti na procesor

Druhý typ testu naopak sleduje využití procesoru v čase konání turnaje. Stejně jako v případě paměti zde byl uspořádaný turnaj mezi umělými inteligencemi v prostředí každý s každým na dvě kola a na velikost herní desky 5x5.

Na získání dat jsem využil obdobný způsob jako v případě získávání využití paměti. Pro využití procesoru jsem využil program `top` a přes `awk` jsem si filtroval procentuální využití procesoru. Vzhledem k tomu, že testy se odehrávaly na virtuálním stroji, bylo využití procesoru prakticky nulové když server neběžel, to znamená, že hodnoty z grafu odpovídají skutečnému využití procesoru Manažerem.

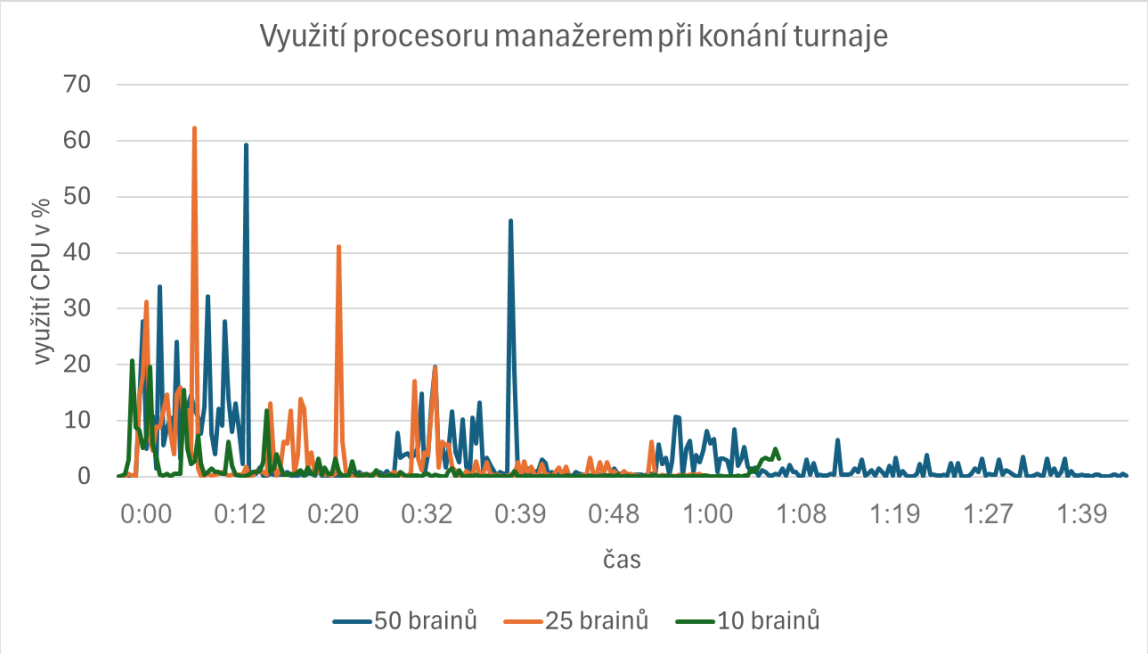
```
#!/bin/bash
while true; do
    top -bn1 | grep "Cpu(s)" | awk '{print $2 + $4}'
    sleep 0.25
done
```



Obrázek 7.3: Graf využití procesoru při pořádání turnaje pro vybrané sety brainů - horní polovina

Za zmínku stojí první fáze u všech testovacích sad, kde využití procesoru je mnohem více rozličné a rovnoměrné v porovnáním se zbytkem. Tento první nárazový zásah na výkonové nároky je způsobený již zmiňovaným dávkováním nasazováním kontejnerů, kvůli limitaci dávky na maximálně počet vláken fyzického procesoru dotnet runtime umí efektivně využít teoreticky všechny jádra optimálně a nedochází ke vzájemným konfliktům o časové okno zpracování. Jedná se také fakticky o část, kde probíhá kompilace zdrojových souborů, kdy

se takových malých "zdrojáků"začne kompilovat desítky to zcela jistě způsobí pozorované vytížení systému.



Obrázek 7.4: Graf využití procesoru při pořádání turnaje pro vybrané sety brainů - spodní polovina

Podivné špičky v průběhu turnaje jsou čistě způsobené hromadným zápisem databáze na disk. Při testování jsem některé sady zkusil pustit i ve větším množství na plotnovém disku (protože sady na grafech 7.3 7.4 běžely na NVME SSD). Ty sady, co potřebovaly zapisovat na databázi na plotnovém disku výrazně méně zatěžovaly procesor a naopak zcela saturovaly diskovou jednotku zápisy. U SSD jsem pozoroval značně zvýšenou aktivitu také, nicméně SSD je násobně rychlejší oproti plotnovým diskům. Jedině v těchto případech dokázal manažer využít procesor na maximum se všemi jádry.

Nevyužitý procesorový výkon mi v tomto kontextu dává smysl, protože manažer nevykonává téměř v žádném kroku výpočetně náročnou operaci, ale naopak má práci ve velkém množství orientovanou na IO[2] operace s rozličnými zdroji dat (docker, databáze, soketová komunikace mezi brainy), které budou vždycky v kontextu rychlosti procesoru velice pomalé i v případech, kdy vše pojede z SSD. Instalovaný procesor v mém počítači je v tomto kontextu velice rychlý procesor, který bych si dokázal představit i na menším serveru.

[2]Input/Output

Kapitola 8

Závěr

Při implementaci třetí verze Manažera se mi podařilo celkovou koncepci systému převést na moderní platformu. Díky webové technologii si Manažer nechává multiplatformní přístup vyzdvihovaný v předešlé práci, zároveň si ponechal paralelní zpracování turnajů. Manažer také centralizoval pohled na systém, který nyní splňuje plnohodnotnou pomůcku dostupnou odkudkoliv a komukoliv. Díky robustní architektuře je Manažer připravený na případná rozšíření. Nabízí se mimo jiné integrovat Manažera do systému IS VUT podobně, jako je integrovaný ISU Hub. Přestože Manažer nedisponuje všemi podporovanými hrami jako původní manažeři, svým návrhem na tyto nedostatky myslí, naopak dovoluje zcela změnit architekturu brainů. Vedle toho jsou nyní brainy jednoduše nasaditelné, centralizované a navzájem i vůči sobě dostatečně zabezpečené. Na této práci jsem si mohl vyzkoušet práci s paralelním zpracováním turnajů, vymyslet robustní řešení výlučného přístupu s robustním odchytáváním chyb z úloh na pozadí, ochytat si velké množství různých technologií a vyzkoušet si postavit projekt s komplexní architekturou zcela od začátku až do finální podoby.

Literatura

[1] What is docker? *Dockerdocs* [online]. [cit. 2024-10-28]. Dostupné z: https://docs.docker.com/get-started/docker-overview.

[2] About Wine. *WineHQ* [online]. [cit. 2024-10-27]. Dostupné z: https://www.winehq.org/about.

[3] Mechanisms to determine if software is running in a VMware virtual machine. *Broadcom* [online]. 1. května 2015 [cit. 2024-10-27]. Dostupné z: https://knowledge.broadcom.com/external/article?legacyId=1009458.

[4] *But, which flavor of ASP.NET?* [online]. 13. listopadu 2019 [cit. 2025-04-28]. Dostupné z: https://jonhilton.net/picking-the-right-flavor-of-asp-net/.

[5] Secure Docker-in-Docker with System Containers. *Nestybox Blog Site* [online]. 14. září 2019 [cit. 2024-10-28]. Dostupné z: https://blog.nestybox.com/2019/09/14/dind.html.

[6] Anton, S. What is a Double Elimination Tournament? *Leaguespot* [online]. 22. listopadu 2022 [cit. 2024-10-27]. Dostupné z: https://leaguespot.gg/blog/double-elimination-tournament.

[7] Binnie, C. Avoid Using Lazy, Privileged Docker Containers. [online]. 31. května 2017 [cit. 2024-10-30]. Dostupné z: https://insights.sei.cmu.edu/blog/virtualization-via-containers/.

[8] Brauner, C. Runtimes And the Curse of the Privileged Container. *Personal blog of Christian Brauner* [online]. 12. února 2019 [cit. 2024-10-28]. Dostupné z: https://brauner.io/2019/02/12/privileged-containers.html.

[9] Firesmith, D. Virtualization via Virtual Machines. *Carnegie Mellon University* [online]. 14. srpna 2017 [cit. 2024-01-16]. Dostupné z: https://insights.sei.cmu.edu/blog/virtualization-via-virtual-machines/.

[10] Firesmith, D. Virtualization via Containers. *Carnegie Mellon University* [online]. 25. září 2017 [cit. 2024-01-17]. Dostupné z: https://insights.sei.cmu.edu/blog/virtualization-via-containers/.

[11] Graziano, C. D. *A performance analysis of Xen and KVM hypervisors for hosting the Xen Worlds Project*. Ames, Iowa, 2011. Diplomová práce. Iowa State University. Dostupné z: https://dr.lib.iastate.edu/handle/20.500.12876/26405.

[12] Hanysz, A. Swiss Pairing. *Sensei's Library* [online]. 20. března 2024 [cit. 2024-10-27]. Dostupné z: https://senseis.xmp.net/?SwissPairing.

[13] Kenneth Hess, A. N. *Practical Virtualization Solutions: Virtualization from the Trenches*. 1. vyd. Pearson Education, 2009. ISBN 0137142978; 9780137142972.

[14] Kerrisk, M. selinux(8). *Man7* [online]. 29. dubna 2005 [cit. 2024-10-28]. Dostupné z: https://man7.org/linux/man-pages/man8/selinux.8.html.

[15] Kerrisk, M. namespaces(7). *Man7* [online]. 13. června 2024 [cit. 2024-10-28]. Dostupné z: https://man7.org/linux/man-pages/man7/namespaces.7.html.

[16] Kerrisk, M. cgroups(7). *Man7* [online]. 15. června 2024 [cit. 2024-10-28]. Dostupné z: https://man7.org/linux/man-pages/man7/cgroups.7.html.

[17] Kouřil, J. *Manažer stolních deskových her*. Brno, 2009. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

[18] Kusák, J. *Manažer stolních deskových her*. Brno, 2007. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

[19] Čížek, T. *Manažer pro deskové hry s paralelním hraním turnajů*. Brno, 2015. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

Příloha A

Protokol komunikace

Manažer posílá přes TCP protokol instrukce, které mají formu textu pomocí ASCII znaků. Při odpovědi brain zasílá vždy dotazovaný příkaz a příslušnou odpověď podle tabulky níže.

Zpráva manažera	**Odpověď brainu**	**Popis**
hello msg	ok error	Dotaz na zjištění připravenosti komunikace brainu. Kladná odpověď značí, že je brain připravený.
com msg	ok error	Dotaz verze na komunikační protokol
game <hra>	ok failed	Informování brainu o hrané hře. Pokud brain hru umí, odpoví kladně.
board <číslo>	ok error	Oznámení velikosti herní plochy
name msg	<jméno> error	Dotaz na jméno brainu. Jméno slouží pro identifikaci brainu v turnaji
timeout <číslo>	ok error	Oznámení o maximální čekací době manažera před odpovědí brainu. Čas v milisekundách.
player <číslo>	ok error	Informace za jakou barvu (figurek/kamenů apod) bude brain hrát.
new game	ok error	Oznámení začátku nové hry
move <tah>	<tah> error	Oznámení brainu tah protivníka. Výzva brainu pro prvotní tah se používá řetězec "start"
quit msg	-	Zpráva pro ukončení komunikace s brainem

Tabulka A.1: Protokol komunikace manažera s brainy