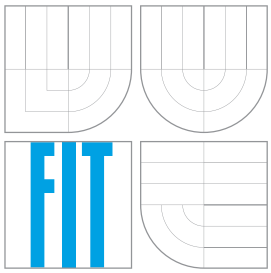
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SYSTÉM PRO KONTINUÁLNÍ INTEGRACI PROJEKTU K-WAVE

CONTINUOUS INTEGRATION SYSTEM FOR THE K-WAVE PROJECT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RADEK NEČAS

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JIŘÍ JAROŠ, Ph.D.

BRNO 2016

Zadání diplomové práce/17430/2015/xnecas18

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Nečas Radek, Bc.**
Obor: Bezpečnost informačních technologií
Téma: **Systém pro kontinuální integraci projektu k-Wave**
Continuous Integration System for the k-Wave Project
Kategorie: Softwarové inženýrství

Pokyny:

1. Seznamte se s technikami kontinuální integrace, automatického testování software na úrovni jednotek, systémy pro správu repositářů a technikami agilního vývoje.
2. Prostudujte dostupné softwarové produkty z dané oblasti. Zaměřte se především na produkty GitLab, Jenkins CI, CPPUnit. Uvažujte rovněž možnost distribuovaného spouštění testů na různých platformách.
3. Prostudujte simulační software k-Wave, vytvořte detailní workflow pro systematický vývoj a testování tohoto simulačního softwaru a navrhněte sadu testů k ověření funkčnosti.
4. Navržený scénář implementuje a otestuje v reálném prostředí s využitím dostupných superpočítačových technologií.
5. Vytvořte podrobnou uživatelskou dokumentaci.
6. Zhodnoťte dosažené výsledky a diskutujte přínos pro další vývoj projektu k-Wave.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 4 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Jaroš Jiří, Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

Kotásek

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

Abstrakt

Tato práce se zabývá zavedením metodik a nástrojů postupné integrace do reálného projektu k-Wave. Věnuji se především verzi psané v jazyce C/C++ s využitím knihovny OpenMP, která je určená pro běh na superpočítačích. Projekt, díky svým vlastnostem a potřebám, nezapadá mezi běžné projekty, do kterých se postupná integrace zavádí. Z toho důvodu musela být řada běžných postupů upravena a bylo nutné vymyslet některé vlastní. Práce popisuje kompletní sestavení prakticky použitelného řešení. Pokrývá jeho návrh, výběr potřebných nástrojů, zprovoznění běhového prostředí, konfiguraci a nastavení služeb, ze kterých se řešení skládá a softwarovou implementaci frameworku pro běh testů na superpočítačích včetně realizace některých regresních a unit testů. Realizace je postavena na službách Gitlab a Jenkins, které běží v oddělených Docker kontejnerech.

Abstract

The main goal of this thesis is to describe the implementation of continuous integration into the k-Wave project. The thesis focuses primarily on the version written in the C/C++ language with the usage of the OpenMP library which typically runs on supercomputers. Accordingly, many of popular workflows and approaches ought to be adapted, a few more created. The outcome of the thesis is a complete solution with real and practical usage. The author provides design, tools selection, runtime environment administration and configuration for each one of the used services. Software implementation of the basic framework is used in order to utilize running tests on the supercomputers. Furthermore, the implementation of chosen types of regression and unit tests are performed. Realisation is based on Gitlab and Jenkis services that are running on separated Docker containers.

Klíčová slova

k-Wave, softwarové inženýrství, kontinuální vývoj, postupný vývoj, kontinuální integrace, postupná integrace, agilní metodyky, agilní postupy, agilní vývoj, systém správy verzí kódu, automatické testování, regresní testování, superpočítače, Anselm, Git, Jenkins, Gitlab, GoogleTest, Docker, Git workflow, Gitflow

Keywords

k-Wave, software ingeneering, continuous integration, continuous delivery, agile, agile development, version control system, automated testing, unit testing, regression testing, supercomputers, Anselm, Git, Jenkins, Gitlab, GoogleTest, Docker, Git workflow, Gitflow

Citace

NEČAS, Radek. *Systém pro kontinuální integraci projektu k-Wave*. Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Jaroš Jiří.

Systém pro kontinuální integraci projektu k-Wave

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jiřího Jaroše, Ph.D. Další informace mi poskytli národní superpočítačové centrum it4i a firma CCV Informační systémy, s.r.o. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Radek Nečas
23. května 2016

Poděkování

This work was supported by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development and Innovations project „IT4Innovations National Supercomputing Center – LM2015070“

Obsah

Úvod

Dnešní software, je čím dál sofistikovanější a rozsáhlejší. Vytváří jej týmy s desítkami členů, kteří mohou být rozmístěny různě po světě. Vývoj softwaru sebou navíc nese specifika, která znemožňují používat klasické metody pro řízení projektů. Jde o vlastnosti jako špatnou měřitelnost a kontrolu práce jednotlivých členů týmu nebo neustálé změny požadavků zákazníka. Díky těmto a mnoha jiným vlastnostem spousta projektů končila neúspěchem. Vzniklo tedy mnoho podnětů ke zdokonalování procesu vývoje softwaru a jeho řízení. Jedním z nich jsou *agilní metodiky a techniky vývoje softwaru*. K dosažení jejich principů jsou doporučovány různé konkrétní postupy a nástroje, mezi které patří i *kontinuální integrace*, která je hlavním tématem následujících kapitol.

V této práci se zaměřím na vysvětlení základních principů kontinuální integrace, jejího zavedení a nástrojů, které jsou pro ni potřebné. Seznámím čtenáře s možnostmi, které jim kontinuální integrace může nabídnout a důvody, proč by ji mohli chtít použít. Popíši běžnou workflow při jejím používání. Dále popisuji integraci do projektu k-Wave, který běží na superpočítačích a pro který jsem musel navrhnout novou workflow. Požadavky na tento projekt a práci týmu se liší od těch běžných, pro které se kontinuální integrace zavádí. Práce tím ukazuje variabilitu popisovaných nástrojů a postupů včetně možnosti jejich použití pro nestandardní projekty. Nedílnou součástí je popis zřízení a správa serveru na kterém poběží vybrané nástroje prostřednictvím Docker kontejnerů. Praktickým výstupem práce je sestavení prostředí a programového frameworku pro zajištění a běh kontinuální integrace projektu k-Wave.

V první kapitole se seznámíme s pojmy z oblasti agilního vývoje, automatického testování, správy zdrojových kódů a kontinuální integrace. Popíšeme si pozici těchto nástrojů a jejich význam i důležitost v procesu vývoje softwaru na kterém se podílí větší počet vývojářů. Tyto informace budou obecné bez ohledu na použité nástroje, techniky a oblast vytvářeného softwaru a budou základem pro výklad v dalších kapitolách.

Druhá kapitola již popisuje konkrétní nástroje a služby z dané oblasti s vysvětlením základních principů jejich využití. Bude zde krátce uvedeno porovnání nejpoužívanějších na trhu. Uvedu zde výběr konkrétních nástrojů a zdůvodnění proč jsem se je rozhodl využít.

Třetí kapitola popíše projekt k-Wave a principy přidělování prostředků používaných na superpočítačích. Vybrané nástroje z předchozí kapitoly sloučíme do jednoho řešení a popíšeme si běžnou workflow práce s nimi. Poté si popíšeme proč tuto běžnou workflow využít nemůžeme a navrhneme možná řešení konkrétních problémů. Vybereme vhodná řešení a sestavíme kompletní workflow pro projekt k-Wave.

Čtvrtá kapitola je ryze praktická a implementační. Zabývá se vytvořením a správou serverového prostředí na kterém běží používané nástroje. Popisuji v ní konfiguraci nástrojů a řešení potíží se kterými jsem se během realizace projektu setkal. Konec kapitoly je věnován programové části řešení a popisuje vytvoření programového frameworku a realizaci regresních testů a testů jednotek, které ukazují způsob práce s vytvořeným frameworkem a lze je použít pro testování projektu.

Poslední pátá kapitola obsahuje zhodnocení práce a přínosů pro projekt k-Wave s ohledem na budoucí možná rozšíření.

Kapitola 1

Techniky a nástroje pro automatizaci vývoje

Tato úvodní kapitola se zaměřuje na základní pojmy z oblasti agilního vývoje, automatického testování a kontinuální integrace. Agilnímu vývoji se v této práci věnuji velmi zběžně a to především z důvodu zasazení dalších informací do kontextu vývoje softwaru.

1.1 Agilní vývoj softwaru

Obecně lze agilní přístupy použít i v jiné oblasti než vývoj softwaru. Nás ale bude zajímat přesně tato oblast. Popis si rozdělíme na dvě části *agilní metodyky* a *agilní techniky* [5].

Agilní metodyky popisují principy, které by se měli při vývoji dodržovat. Zabývají se spíše správou životního cyklu softwaru, řízením projektu, jeho plánováním, dělení práce, způsobem reakce na změny zadání atd. Vycházejí z *manifestu agilního vývoje softwaru* [1].

- Jednotlivci a interakce před procesy a nástroji.
- Fungující software před vyčerpávající dokumentací.
- Spolupráce se zákazníkem před vyjednáváním o smlouvě.
- Reagování na změny před dodržováním plánu.

Agilní techniky se naproti tomu přímo zabývají konkrétními technologiemi, postupy a nástroji používanými při vývoji softwaru. Jsou mnohem blíže vývojářům, kteří se s nimi setkávají denně při svoji práci. Patří sem např. *extrémní programování, automatické testování, refaktorizace či postupná integrace a vydávání* [5].

Základním cílem agilního přístupu je rychlá reakce na změny požadavků. Proto probíhá vývoj v krátkých iteracích. Na konci každé iterace je k dispozici prototyp, který se může předvést zákazníkovi. Zákazník se aktivně účastní vývoje a průběžně vyhodnocuje vytvářený produkt a upřesňuje své požadavky [6].

Vývoj je řízený hodnotou pro zákazníka (*value-driven development*). Každá nově implementovaná funkce nebo opravená chyba by pro něj měla mít jasný přínos. Tento fakt ovlivňuje jak fázi dekompozice problému, tak plánování implementace do jednotlivých iterací. Při návrhu se používají *vertikální řezy systému*. Při tomto postupu může nová funkce zasahovat do všech vrstev softwaru [6].

[1] Zdroj: http://www.agilemanifesto.org/iso/cs/.

1.2 Průběžná integrace

Tento pojem v sobě zahrnuje metodiky, postupy a nástroje sloužící pro zrychlení a zkvalitnění vývoje softwaru a jeho řízení, zjednodušení práce v týmu a zefektivnění práce jednotlivých programátorů. Při dodržování této metodiky se snaží vývojář postupně integrovat svoji práci s prací ostatních členů týmu. Snahou je maximální možné využití softwarových nástrojů a automatizace všech procesů souvisejících s vývojem softwaru a správou jeho životního cyklu. V různých zdrojích lze najít i alternativní názvy jako *postupná integrace* nebo *kontinuální integrace*. V angličtině se ustálil název *Continuous Integration (CI)*.

Hlavními cíli této techniky je zkvalitnění a urychlení vývoje softwaru, snížení chybovosti, zjednodušení a zefektivnění práce v týmu a zvýšení přehlednosti průběhu vývoje a aktuálního stavu projektu.

Sestavení jednotlivých částí projektu do výsledného celku je složitý proces náchylný na chyby. Zdrojové kódy musí být zkompilovány, musí se zkopírovat a instalovat různé knihovny a další pomocné nástroje, upravovat různá konfigurační nastavení apod. Při sestavování vzniká tzv. *build*. Každý build musí před integrací projít sadou automatických testů. Stejným způsobem se testuje integrovaný celek. Do procesu se často zařazuje *měření kvality kódu*, při kterém se sledují různé metriky kódu jako např. pokrytí projektu automatickými testy, dostatečné komentování kódu, dodržování pravidel psaní kódu aj. Proces bývá často integrován s nástroji pro řízení projektu, jako systém pro správu úkolů a další. CI klade velký důraz na vizualizaci výsledků. Většina nástrojů nabízí přehledný *dashboard*, kde jsou rychle k dispozici informace o stavu a průběhu vytvářeného produktu.

CI poskytuje jednotné *testovací prostředí*, které se může dynamicky měnit dle potřeb testování. Tento koncept je důležitý pro práci většího týmu. Vyhneme se situacím, kdy jednomu vývojáři build funguje a projdou mu všechny testy, kdežto ostatní si jej ani nepřeloží v důsledků odlišností běhového prostředí (různé knihovny, verze operačního systému, atd.). Během integrace se mnohdy spouští tisícovky různých testů. Proto umožňují CI nástroje distribuovat tento proces na další stroje principem *master-slave*. *Master* je většinou jeden a řídí celý build. *Slave* uzly se starají o běh jednotlivých testů a dalších pomocných výpočtů. Výsledky poté zasílají master uzlu, který je zobrazuje uživateli. Tento princip se často využívá i pro testování programů na různých běhových prostředí. Slave uzly poté provádí ty stejné operace, ale na různých operačních systémech, s různými verzemi knihoven, konfiguracemi apod.

Z popisu vyplývá, že proces je poměrně složitý a vyžaduje komplikované řízení a spolupráci řady různých nástrojů, o kterých se zmíním v dalších kapitolách.

1.3 Správa zdrojových kódu

Dnešní software se skládá z tisíců někdy i milionů řádků kódu, které vytváří desítky programátorů, kteří mohou pracovat napříč celým světem [2]. Při správě takhle rozsáhlého softwaru se musíme vypořádat s problémy správy zdrojových kódů, mezi které patří například:

- Všichni členové týmu musí mít okamžitě k dispozici aktuální verzi kódu.
- Každý vývojář musí mít možnost nový kód ihned zpřístupnit ostatním členům týmu.
- Musí být oddělena stabilní (distribuční) verze kódu od vývojové.

[2] Zdroj: http://www.informationisbeautiful.net/visualizations/million-lines-of-code/

- Pokud jeden z programátorů zavede do kódu chybu, tak by tato chyba neměla zastavit práci ostatním členům týmu.
- Možnost snadného návratu k předchozí verzi kódu.
- Minimalizace možnosti ztráty kódu, zamezení získání kódu neoprávněnými osobami a další.

Pro zajištění těchto a dalších podmínek existují *systémy pro správu verzí kódu (version control system - VCS)*. Na trhu se vyskytuje řada nástrojů, které splňují výše uvedené požadavky. Některé z nich popíšu v kapitole 2.2.

Systémy ukládají soubory projektu a jejich metadata do tzv. *repozitářů*. Dle způsobů práce s repozitáři si je můžeme rozdělit do dvou základních skupin

- **Centralizované VCS** - Existuje jeden centrální repozitář. Uložení kódu do systému znamená automatické sdílení kódu s ostatními uživateli, což může být nevýhodné při provádění experimentů s kódem. Centrální repozitář je většinou umístěn na serveru. Uživatelé u sebe mají pouze pracovní kopie. Při jakékoliv práci s VCS musí uživatelé komunikovat se serverem. Pokud není server dostupný nemůže uživatel provádět žádné operace. Centralizované řešení je taky hůře škálovatelné s rostoucí velikostí týmu. Tyto systémy jsou vhodné spíše pro organizace s menším počtem vývojářů, sídlících na jedné pobočce, kde jsou často v přímém kontaktu a nejsou kladeny tak vysoké nároky na nezávislost jednotlivých členů týmu.
- **Distribuované VCS (DVCS)** - Každý uživatel má lokální kopii repozitáře. Existuje centrální repozitář, který slouží pro synchronizaci a integraci práce mezi uživateli. Pokud není centrální repozitář k dispozici, tak může uživatel stále pracovat lokálně, pouze nemůže práci sdílet ostatním členům týmu. Toto řešení je komplikovanější než předchozí. Musí se řešit různé synchronizační konflikty při integraci do centrálního repozitáře. Tyto systémy jsou vhodné i pro velké týmy, kde jsou jednotlivý členové rozmístěni různě po světě a je kladen velký důraz na nezávislost jejich práce. Proto jsou často používány pro open-source projekty. Většina nových VCS se vydává touto cestou.

1.4 Automatické testování

V agilních metodách má každý člen týmu (programátor, tester i zákazník) určitou zodpovědnost za kvalitu výsledného produktu. Časté provádění testů je jednou z technik, které pomáhají tuto kvalitu zajistit a udržet ji během budoucích úprav a rozšíření kódu. Provádění testů je často opakovaná, rutinní činnost, která zabírá spoustu zdrojů týmu. Proto je v dnešní době snahou tuto činnost maximálně automatizovat.

Existují různé typy testů, které lze automatizovat různou mírou:

- **Unit testy** - Testování základních jednotek je základní a nejčastěji používaná metoda. *Základní jednotkou* se zde myslí nejmenší jednotka kódu, jejíž testování dává smysl (funkce, metoda, atd). Pro tento typ testů existuje řada knihoven či frameworků a jsou nedílnou součástí metod CI. Jdou velmi dobře automatizovat. Více v 1.4.1.
- **Integrační testy** - Jde o testování větších jednotek kódu a jejich integrace. Typické pro testy tohoto typu je využití komponent, nad kterými nemáme plnou kontrolu

a skutečných závislostí jako např. připojení k sítí, databáze, náhodné generátory, atd. Tyto testy se většinou provádí stejnými nástroji jako unit testy, ale obtížněji se automatizují. Někdy je obtížné určit hranici mezi unit testy a integračními testy.

- **Regresní testy** - Při tomto způsobu testování jde o odhalení softwarových chyb při úpravách kódu a *regresí*. Regrese je situace, kdy program stále funguje, ale běží pomaleji či využívá více paměti atd. Pro tyto testy se užívá opakované spouštění unit a integračních testů, případně se spouští programy s různými parametry či nastaveními a vyhodnocují se výkony jednotlivých běhů. Často se při regresních testech kontrolují výstupy různých verzí či nastavení programu.

- **UI testy** - Testování grafického uživatelského rozhraní lze také velmi dobře automatizovat. Existuje řada nástrojů pro UI testy webů či desktopových aplikací. Tyto testy se zaměřují na chování uživatelských rozhraní programů, tedy různá zobrazování, přechody mezi obrazovkami atd. V tomto projektu jsem se tímto testováním nezabýval.

Existuje řada dalších typů testů, které lze automatizovat jako například zátěžové testy, bezpečnostní testy a jiné. V této práci se jimi ale nebudu zabývat.

1.4.1 Unit testy

Provádění tohoto způsobu testování je tak častou a důležitou součástí tvorby softwaru, že vedlo až k zavedení metodiky *programování řízené testy(Test Driven Development, TDD)*. Při tomto způsobu testování se nejdříve píšou testy a až poté programuje. Na začátku vývoje všechny testy selžou a programátor při vytváření kódu postupně dosahuje plnění jednotlivých testů. TDD se v dnešní době dále rozvijí a vznikají z něj další postupy jako *Programování řízené chováním (Behaviour Driven Development, BDD)*, aj [7].

Vytváření Unit testů zjednodušují knihovny, které jsou k dispozici pro všechny běžně používané jazyky. Pro tvorbu kvalitních testů je důležitý kvalitní návrh, který by měl splňovat tyto principy [7]

- Měl by být plně automatický a opakovatelný. Proto tyto testy nevytváří textové či jiné výstupy, ale typicky obsahují `assert instrukce`, které v případě neúspěchu testů, jeho běh ukončí s chybou.

- Měl by být plně izolovaný. Každý test může být spuštěn kdykoliv a kýmkoliv. Nemělo by tety záležet na pořadí jednotlivých testů či nastavení prostředí (test by si jej měl případně nastavit sám).

- Každý test by měl být jednoduchý a přehledný. Upřednostňuje se tvorba více malých testů před jedním velkým. Test by měl také dávat jednoznačnou zpětnou vazbu pochopitelnou i po delší době nepoužívání testu. Měl by být jednoznačně pojmenovaný a každá chyba, odhalená testem, by měla být jednoznačně identifikovaná a popsaná.

Testování větších celků kódu probíhá typicky *zdola-nahoru*. Nejdříve se testují úplně nejzákladnější jednotky, poté až vyšší komponenty, které tyto základní používají ke své činnosti. Typické je použití *falešných komponent*, které simulují chování těch základních a podávají předem jasně definované výsledky, určené pro potřeby testování vyšší komponenty. Těmto komponentám se říká *fake* nebo *mock*. Tato technika je velmi často používaná, proto existuje řada různých knihoven, které tzv. *mockování* zjednodušují [7].

Důležité je také si uvědomit, že každý software není dobře testovatelný. Je potřeba uzpůsobit architekturu softwaru tak, aby byl lépe testovatelný a šel dobře *pokrýt testy*. Uplatňují se různé principy návrhových vzorů. Dobře testovatelný kód bývá většinou i dobře rozšiřitelný a robustní. Zvyšuje se tedy hodnota softwaru z pohledu softwarového inženýrství. V dnešní době potřeby dobré testovatelnosti značně ovlivňují návrh softwaru [7].

Kapitola 2

Softwarové nástroje

V předchozí kapitole jsem se snažil vysvětlit důležitost použití softwarových nástrojů při používání technik kontinuální integrace a agilních metodikách obecně. Popsal jsem třídy těchto systémů a jejich pozici v procesu vývoje softwaru. V této kapitole popíši některé z těchto nástrojů a jejich základní principy. Především se zaměřím na ty, které jsem použil prakticky pro tento projekt. Také zde popíši nástroje, které jsem použil pro vytvoření běhového prostředí projektu.

2.1 Správa serveru

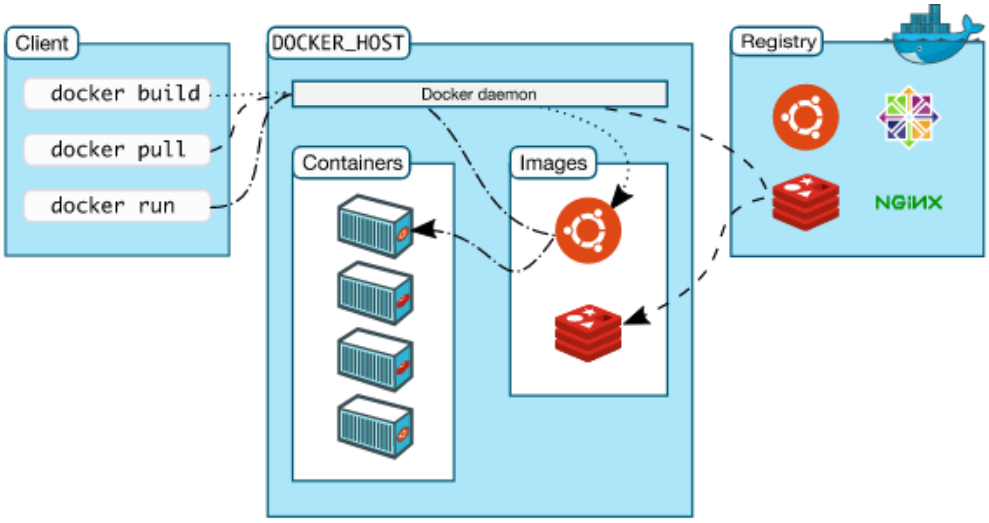
V této části stručně popíši `Docker`, hlavní nástroj, který jsem použil pro vytvoření a správu prostředí kontinuální integrace.

2.1.1 Docker

`Docker` je otevřená platforma pro vývoj, distribuci a běh aplikací. Hlavním cílem jeho tvůrců bylo oddělení aplikace od infrastruktury a schopnost zacházet s infrastrukturou jako spravovatelnou aplikací. Díky tomu lze snadno vytvářet různá aplikační prostředí, která lze bez problémů přenášet s celou jejich konfigurací, a lze je snadno rozšiřovat, či jiným způsobem modifikovat. V těchto prostředích, poté běží vaše aplikace, kterým je zajištěný bezpečný a izolovaný běh na `hostující stanici` - počítači, kde běží Docker jako takový. Lze jej použít i pro tvorbu distribučních prostředí, čehož jsem využil v tomto projektu.

Docker je založen na klient-server architektuře, kterou můžete vidět na obrázku 2.1 Architektura se skládá z těchto komponent:

- **Klient (client)** - Poskytuje uživatelské rozhraní, přes které uživatel zadává příkazy a skrze které se mu zobrazují výsledky. Požadované operace neprovádí sám, ale deleguje je na Docker deamona, se kterým komunikuje skrze sockety nebo RESTFull API.
- **Docker deamon** - Běží na hostující stanici, kde sestavuje, spouští a distribuuje Docker kontejnery.
- **Docker obrazy(images)** - Jde v podstatě o šablonu prostředí, určenou pouze ke čtení (např. Ubuntu distribuce s Gitlab serverem).Tyto image jsou často volně šířené komunitou prostřednictvím Docker hubu. Docker nabízí sadu nástrojů, pomocí kterých lze vytvářet nové image, upravovat stávající, publikovat image do hubu, atd.



Obrázek 2.1: Architektura Dockeru

- **Docker registry** - Veřejná či soukromá uložiště Docker image. Existuje tzv.`Docker hub`, oficiální úložiště se spoustou image, volně ke stažení.
- **Docker kontejner** - Podobají se klasickým adresářům nebo běhovým prostředím. Obsahují vše potřebné pro běh aplikací. Každý kontejner je tvořen z Docker image, dá se tedy říct, že je její instancí. Tvoří bezpečnou a izolovanou platformu, ve které může image běžet. Kontejnery mohou být spuštěny, zastaveny, přesunuty nebo smazány. Kontejner většinou obsahuje operační systém, uživatelské aplikace a další nastavení.

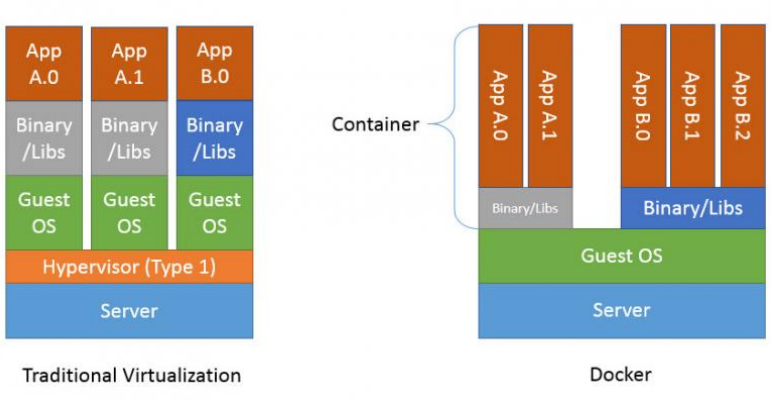
Narozdíl od *virtuálních počítačů* jsou kontejnery mnohem lehčím prostředím. Někdy se práce s kontejnery nazývá *virtualizací na úrovni operačního systému*. Kontejnery nemusí virtualizovat operační systém a mohou mezi sebou sdílet různé knihovny, jak je vidět na obrázku 2.2. Docker využívá komponent jádra operačního systému `Linux`. Nelze jej tedy přímo spustit na operačních systémech jiného typu. Stejně tak nemůžete vytvořit image s jiným než Linuxovým prostředím. Pro běh na jiných operačních systémech je potřeba nainstalovat virtuální počítač s Linuxovým operačním systémem, a v něm provozovat Docker.

Docker využívá těchto komponent jádra operačního systému Linux:

- `namespaces` - Pomocí nich zajišťuje izolované prostředí kontejnerů.
- `cgroups` - Slouží pro správu přidělování prostředků kontejnerům (CPU, paměť, ...).
- `unionFS` - Jednotný souborový systém s vrstvovou architekturou. Sjednocuje souborové systémy kontejnerů. Umožňuje snadné rozšiřování existujících image.

2.2 Nástroje pro správu zdrojových kódů

Nejvíce se v této části budu věnovat nástroji Git, který jsem zvolil pro tento projekt. Ostatní nástroje popíši spíše přehledově s jejich vlastnostmi v porovnání s Gitem.



Obrázek 2.2: Rozdíl mezi kontejnerem a virtuálním počítačem

2.2.1 Git

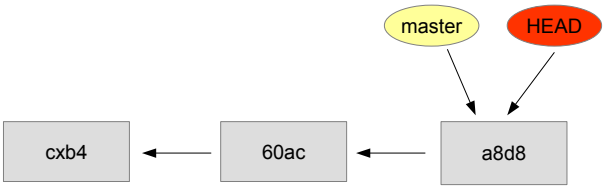
V dnešní době jeden z nejpoužívanějších nástrojů pro správu zdrojových kódů. Nástroj založila v roce 2005 komunita vývoje Linuxového jádra v čele s Linusem Torvaldsem tvůrcem Linuxu. Mezi základní principy tohoto nástroje patří rychlost, jednoduchý design, silná podpora pro nelineární vývoj (tisíce paralelních větví kódu) a schopnost zvládnout velké projekty jako např. Linuxové jádro. Na nástroji lze rozpoznat linuxovou filozofii v tom, že obsahuje sadu základních jednoduchých instrukcí, pomocí kterých si uživatel může sestavit komplexnější nástroj přesně dle svých požadavků.

Systém je plně distribuovaný, každá změna se ukládá ve formě *snímků (snapshotů)* v lokálním *repozitáři*, uložišti na počítači, kde git uchovává data a metadata souborů. Lokální repozitář může být snadno synchronizován se vzdáleným, sdíleným repozitářem, často zvaným *origin* nebo *centrální repozitář*. Existuje i možnost mít více origin repozitářů.

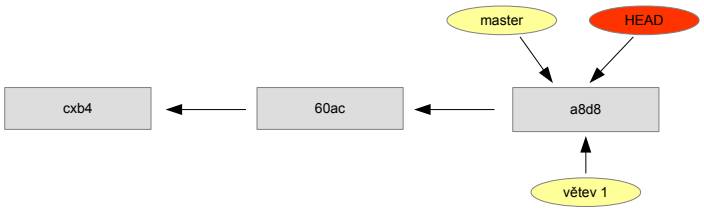
Uložení snímku do lokálního repozitáře se říká *commit*. Každý commit je jednoznačně identifikován pomocí svého *hashe*. Jednotlivé commity si Git uchovává v acyklickém grafu, kde každý uzel (commit) zná svého přímého předchůdce a může mít jednoho či více následníků (více v případě větví, viz dále). Na poslední, aktuálně uživatelem využívaný, commit ukazuje ukazatel zvaný *head*.

Systém umožňuje vytváření *větví*. Jde v podstatě o paralelní nezávislou linii kódu. Vždy existuje alespoň jedna výchozí větev, tzv. *master*. Vytvoření větve je velmi jednoduché. V podstatě se vytvoří pouze nový nezávislý ukazatel, směřující vždy na poslední commit v dané větvi. Při přepínání uživatele mezi větvemi se pouze jeho head ukazatel přepíná na ukazatele jednotlivých větví. Práce s větvemi je znázorněna na obrázku 2.2.1. Díky popisovanému principu je práce s větvemi velmi jednoduchá a flexibilní. Což je jedna z věcí, kterými se Git liší od ostatních nástrojů. Větve jsou často využívány při práci většího týmu. Každý vývojář si vytvoří svoji větev ve které provádí potřebné změny. Po dokončení své práce svoji větev sloučí s hlavní větví [10].

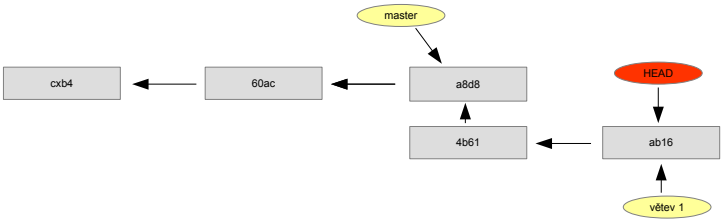
Uživatel může procházet historii stavů projektu, pozorovat jaké změny se prováděli, vracet se k předchozím verzím kódu a provádět nad nimi úpravy. Tyto změny pak může uložit jako aktuální verzi kódu, případně jako separátní větev. Při provádění takovýchto úprav, nebo při práci s více větvemi, musí uživatel často provádět tzv. *merge* neboli sloučení úprav (větví) do jedné. Při tomto procesu se Git musí vypořádat s řadou problémů mezi verzemi kódu, jako např. existence různých souborů nebo rozdílný obsah jednotlivých souborů. Merge tvoří často problém, především pokud více vývojářů upravuje stejný soubor.



Obrázek 2.3: Git s výchozí master větví



Obrázek 2.4: Git vytvoření nové větve



Obrázek 2.5: Git práce v nové větvi

Mohou vzniknout situace kdy jednotlivé větve kódu fungují, ale po jejich mergi výsledný kód nefunguje. Pro minimalizaci tohoto problému je vhodné, pokud každý vývojář upravuje jinou sadu souborů, za kterou je poté zodpovědný). Toto není mnohdy v praxi možné, proto se nasazují pokročilejší techniky pro řešení těchto problémů, které se jej snaží automaticky řešit, případně alespoň minimalizovat jeho dopad [10].

Git je používán jako VCS pro *Github*. Webovou službu pro správu zdrojových kódů. Github je jeden z nejpoužívanějších veřejných repozitářů a stává se v podstatě standardem pro uchovávání open-source projektů.

Nejčastěji jsou Gitu vytýkány vlastnosti jako delší učící křivka, komplikovanější a méně intuitivní ovládání skrze příkazovou řádku nebo nutnost pracovat s hashem pro identifikaci konkrétních commitů.

2.2.2 Mercurial

Tento nástroj vznikal současně s Gitem v roce 2005. Jde opět o distribuovaný systém pro správu kódu. Vlastnostmi a množinou funkcí téměř totožný s Gitem. Narozdíl od něj je ale navržen jako jeden monolitický software se striktněji předdefinovanými funkcemi a postupy použití. Dle uživatelů a tvůrců se dá Mercurial rychleji naučit a uživatelé jsou schopni dříve využít všech jeho možností, včetně různých *workflow*. Zastánci jiných nástrojů, především Gitu, kritizují jeho princip práce s větvemi, kdy není možné plně smazat pojmenovanou větev. To je problematické při práci více uživatelů, např. při tvorbě open-source projektů, kdy si jednotlivý uživatelé chtějí vytvářet větve pro experimenty. Tyto větve poté nejdou ze systému vymazat a zbytečně v něm zůstávají a snižují tak přehlednost ve verzích kódu. Neexistuje navíc žádná webová služba podobná Githubu, která by tento nástroj tak masivně podporovala [8].

2.2.3 Apache Subversion (SVN)

Jde o jeden z dřívějších VCS. Vznikal již od roku 2000, jako náhrada populárního nástroje CSV. V mnohém tento nástroj vylepšila. Dle kritiků jsou ale některé operace, jako např. vytváření větví, komplikovanější, pomalejší a těžkopádnější než u dříve popsaných nástrojů. Zveřejnění práce probíhá automaticky vždy při každém commitu, což vždy nevyhovuje způsobu práce v týmu. Centralizovaný přístup sebou nese horší škálovatelnost. Především u větších týmů, mohou vznikat problémy s dostupností serveru(u Git a Mercurial můžete provádět téměř cokoliv offline a vše uložit až je server k dispozici) atd. Výhodou je jednoduchost tohoto nástroje. Na trhu se také vyskytuje delší dobu, takže jej mohou mít některé týmy více zažitý [1].

2.2.4 Veracity

[1] Poměrně mladý projekt, který vznikl v roce 2011. Opět jde o distribuovaný systém se schopnostmi velmi podobnými Gitu a Mercurialu. Navíc ale přidává některé nové vlastnosti jako např.:

- Přímá integrace distribuované správy chyb a úkolů.
- Příma integrace distribuované wiki.

[1] Domovská stránka: http://veracity-scm.com

- Zavedení uživatelských účtu pro projekt. Díky této vlastnosti lze například zamykat soubory jen pro konkrétní uživatele apod.
- Nové hashovací algoritmy, integrovaný skriptovací jazyk (Javascript), podpora mobilních zařízení, aj.

Nástroj není prozatím příliš populární. Přináší ale řadu zajímavých nápadů a rozšíření.

2.2.5 Závěr

Kromě popsaných existuje i řada jiných, dnes často používaných nástrojů. Za zmínku stojí například *Team Foundation Server* od Microsoft, *Bazaar*, *Fossil*, aj. Většina nabízí velmi podobnou řadu funkcí. Objektivně nelze určit, který z nich je lepší. Většinou při výběru záleží na zkušenostech jednotlivých členů týmu s konkrétním VCS, na prostředí ve kterém je projekt vytvářen nebo na tom, zda je dané VCS podporované dalším softwarem, který tým používá. Já jsem pro projekt zvolil Git, kvůli jeho rozšířenosti, široké komunitě a rozsáhlé podpoře nástrojů třetích stran.

2.3 Hostování Git repozitářů

Jeden z prvních problémů, které musíte při správě zdrojových kódů pomocí Git řešit, je kam umístit origin repozitář. Při tomto úkolu musíte zajistit způsob komunikace s klienty, správu a řízení přístupu jednotlivých uživatelů a úkoly spojené s bezpečností. Na webu existuje řada služeb a produktů, které za vás tento problém vyřeší. Navíc obsahují spoustu dodatečných funkcí jako například správa úkolů, wiki stránky, fórum pro vývojáře i uživatele softwaru, aj.

Při výběru vhodného řešení musíte brát v potaz otázky:

- Rozhodnutí mezi veřejným či soukromým repozitářem. Možnost uchovávání dat na veřejném serveru.
- Nabízené služby, jako bezpečnost systému, správa uživatelů, vedení úkolů aj.
- Finanční náročnost. Existuje řada placených služeb. Spousta hostingů nabízí placená řešení.

2.3.1 Hosting jako služba

Jde o nejjednodušší a nejrychlejší řešení. Stačí vytvořit účet, projekt, přidat jednotlivé uživatele, popřípadě provést základní konfiguraci prostředí a můžete vše začít používat. Všechnu další práci od hardwaru po správu prostředí a zajištění bezpečnosti pro vás zajistí poskytovatel.

Existují služby zdarma i placená řešení. Některá řešení zdarma umožňují vytváření soukromých repozitářů. Hlavní otázkou tedy zůstává, zda si můžete dovolit umístit data na cizí server.

Nejznámější službou je Github s více jak 10 miliony aktivních repozitářů. Tato služba se v dnešní době stává v podstatě standardem pro uložení open source projektů. Další známé služby jsou například *Bitbucket, SourceForge, CodePlex, Visual Studio Online*, aj. Většina nabízí velmi podobnou sadu funkcí s možností placených rozšíření.

2.3.2 Vlastní hosting

Pokud chcete ukládat data na vlastním serveru je potřeba si zajistit vlastní hosting. Nejznámějším nástrojem této oblasti je *Gitlab*, který má k dispozici *Community edici (CE)* zdarma. Dále nabízí *Enterprise edici (EE)* s rozšířenými možnostmi, popřípadě i hosting jako službu na jejich serveru [3]. Konfigurace vlastního hostingu je popsána v kapitole 4.2.

Gitlab od verze CE nabízí službu *Gitlab CI*, která poskytuje základní možnosti pro zajištění kontinuální integrace. Více k těmto možnostem bude popsáno v kapitole 2.5.1.

2.4 Nástroje pro automatické testování

V této kapitole stručně popíši nástroje pro unit testy programů psaných v jazyce `C/C++`. Stručně uvedu další nástroje související s automatickým testováním.

2.4.1 xUnit

Unit testy se nejčastěji provádějí v jazyce, jakém je psaná samotná aplikace. Většina populárních jazyků má své knihovny. Java má `JUnit`, Python `pyUnit`, C# `NUnit`, aj. Souhrnně se tyto knihovny nazývají termínem `xUnit`. Jelikož je projekt k-Wave psán v `C/C++`, tak jsem experimentoval s knihovnami `CUnit`, `CppUnit` a `GoogleTest`.

Práce s unit testy je v `C/C++` náročnější než v moderních jazycích jako `Java` či `Python`. `C` ani `C++` nemá konstrukce pro reflexivní programování či anotace kódu. Je tedy potřeba napsat více kódu než jinde. Tyto nevýhody se snaží minimalizovat knihovna `GoogleTest`. Ta využívá jazyk Python pro automatizaci některých úkolů jako např. zavádění testů.

Jednotlivé knihovny se od sebe samozřejmě liší, ale základní princip použití zůstává velmi podobný. Často se používá i podobné názvosloví, které obsahuje pojmy jako:

- **Test Fixture** - Zabaluje jednotlivé testovací a další pomocné metody (funkce). Umožňuje použít stejná data i nastavení napříč všemi `test case`.
- **Test Case** - Testovací metoda (funkce). Testování se provádí pomocí `assert` příkazů.
- **Assert instrukce** - Podmíněné instrukce, které v případě nesplnění podmínky ukončí test case s chybou a případnou (nepovinnou) chybovou hláškou. Knihovny obsahují sadu těchto instrukcí. Lze porovnávat logické hodnoty, čísla, adresy do pamětí, objekty atd.
- **setUp() a tearDown() metody** - inicializační a úklidová metoda. Volaný vždy před/po spuštěním každého test case.
- **Test Suite** - Třída, která zabalí více Test Case, které mají být volány současně.
- **Test Runner** - Třída, která spouští jednotlivé test case. Nastavují se pomocí ní i objekty, které definují výstup testů.

Knihovna GoogleTest nepoužívá komponenty Test Suite ani Test Runnery, respektive je před uživateli skrývá. Více o implementaci unit testů pomocí této knihovny naleznete v kapitole 4.4.2.

Výsledky testů se běžně vypisují na standardní výstup. Pro potřeby tohoto projektu, bylo důležitou vlastností schopnost knihovny vypisovat výsledky testů do souboru. Pomocí výstupních XML souborů je zajištěno předání výsledků do nástrojů řízení kontinuální integrace, které poté reagují na výsledky testů a poskytují přehledné reporty.

2.4.2 Další nástroje pomocné při automatickém testování

Existují různé další nástroje, které usnadňují automatické testování. Časté jsou například různé *mockovací knihovny* nebo knihovny pro *dependency injection*. Vývojová prostředí často obsahují nástroje pro snadné spouštění testů, zobrazení jejich výsledků či měření pokrytí kódu testy. Existují i pokročilejší frameworky sloužící pro testování řízené chováním (BDD). Ty obsahují prostředky, jak formálně popsat chování testované jednotky a přiřadit ji sadu testů, která by měla toto chování potvrdit. Příkladem může být například `SpecFlow` v kombinaci s `WatiN` a `NUnit` pro jazyk `C#`. UI testování lze automatizovat například pomocí nástrojů `Selenium` (web) či `TestStack.White` (desktopové aplikace Windows).

2.5 Nástroje pro postupnou integraci

Postupná integrace (CI) je dnes velmi populární technikou. Díky tomu existuje řada nástrojů pro její zajištění od komerčních (placených) po open source projekty. Nástroje poskytují značně pokročilé funkce a současně kladou velký důraz na uživatelskou přívětivost. Hlavní podpora leží v oblasti mainstreamových jazyků jako `Java` nebo `C#`. Podpora jazyků `C/C++` není tak velká. Většina nástrojů umožňuje řídit projekty těchto jazyků, ale nenabízí pro ně takové možnosti jako pro dříve zmíněné.

Nástroje jsou většinou koncipované jako webové servery ovládané přes webové stránky, případně konfigurační soubory. Řada z nich navíc poskytuje veřejné API, pomocí kterého lze s nástroji komunikovat vzdáleně z jiných systémů. Pro potřeby tohoto projektu bylo důležité, aby nástroje splňovali tyto podmínky:

- Podpora jazyků `C/C++, make, cmake, automake, bash skriptů, python skriptů` atd.
- Distribuce buildů na více výpočetních stanic, podpora různých konfigurací pro jednotlivá prostředí.
- Podpora pro Git, možnosti pokročilejší práce s větvemi Gitu (merge větví po úspěšném buildu s testy).
- Podpora automatických a regresních testů (nástroje Cunit, CppUnit) včetně reportů.
- Podpora dalších nástrojů např. kontrola dodržování struktury kódu (style guide, coding rules), upozornění a výpisy překladače, jednoduché statické kontroly kódu, generování dokumentace (Doxygen).
- Upozornění na email (např. neúspěšný build). Vhodná je i možnost integrace se systémem pro správu úkolů.
- Možnosti rozšíření - např. možnost doprogramovat vlastní modul přidávající funkčnost, možnosti vlastní komunikace s CI serverem (např. přes API).
- Dobrá dokumentace, velká komunita.

V této kapitole popíšu především nástroj *Jenkins* a *Gitlab*, které jsem zvolil pro tento projekt. Ostatní nástroje uvedu pouze přehledově s porovnáním jejich vlastností.

2.5.1 Gitlab CI

Jde o rozšíření, které je přímo součástí Gitlabu o kterém jsem se zmínil již v kapitole 2.3.2. Toto řešení je výhodné pokud již používáte Gitlab pro hostování zdrojových kódů. V takovém případě vám postačí správa jednoho prostředí skrze jedno webové rozhraní a jedna sada uživatelských účtů. Budete mít k dispozici základní funkce jako například distribuovaný build, automatické spouštění testů, mergování kódu atd. Chybí ale pokročilejší podpora jazyka `C/C++` a možností skriptování.

Gitlab CI je většinou potřeba doinstalovat, popřípadě jej povolit během instalace. Skládá se ze dvou základních komponent [3]:

- **Koordinátor** - Zobrazuje všechny výsledky na webovém rozhraní a řídí samotný build.
- **Runner** - Gitlab CI je tvořeno tak, že veškerý build distribuuje. Build samotný vždy provádí jeden nebo více runnerů. Ty mohou běžet na více strojích nebo na jednom lokálním serveru společně s koordinátorem.

Základní nastavení se provádí přes webové rozhraní Gitlabu. Pro nastavení buildu se ale nevyhnete psaní `Yaml skriptů`. Pro potřeby tohoto projekty byly nabízené možnosti nedostatečné [3].

2.5.2 Jenkins

Zřejmě nejpoužívanější open source řešení pro kontinuální integraci, psané v jazyce Java. Je užíváno týmy různých velikostí pro projekty různorodých jazyků a prostředí jako např. `Java, .NET, Ruby, Python, Groovy, Grails, PHP, C/C++` a jiné. Mezi základní výhody tohoto prostředí patří jednoduchost použití, široká podpora jazyků či technologií a velká popularita, která sebou nese širokou komunitu a dobrou dokumentovanost. Velký důraz je kladen na rozšiřitelnost a možnost upravit si chování dle vlastních potřeb. K tomuto účelu je k dispozici široká řada existujících pluginů s možností dopsat si své vlastní.

Veškerou konfiguraci lze provést z webového rozhraní, přes které je k dispozici i repozitář pluginů. V základní verzi nabízí Jenkins pouze omezenou škálu funkcí. Instalaci pluginů se tedy nevyhnete téměř v žádném projektu. Webové rozhraní také zobrazuje veškeré informace o stavu projektu a průběhu jednotlivých buildů. Konfigurace Jenkins serveru pro tento projekt je popsána v kapitole 4.3.

Struktura Jenkins projektu

V této části budou popsány základní komponenty Jenkins serveru a jednotlivé kroky tzv. *buildu*.

Základními komponentami pro práci s Jenkins jsou:

- **Jenkins projekt** - Projekt v rámci kterého konfigurujete nastavení a zobrazujete informace na Jenkins serveru. Slovo Jenkins je uvedeno explicitně aby nedošlo ke zmatení s projektem, který jako tým vytváříte (nazývám jej vývojový projekt). Jedna instance Jenkins serveru může zpracovávat více vývojových projektů, stejně tak můžete mít pro váš vývojový projekt vytvořeno více Jenkins projektů např. z důvodů různorodé konfigurace či potřeb buildu.

- **Pohledy (View)** - Jednotlivé Jenkins projekty jsou slučovány pro přehlednost do pohledů. Pohledy mohou mít vlastní konfiguraci jako zobrazování ve formě seznamu, základní zobrazované informace atd.
- **Build proces** - Pro každý projekt je definován build proces, který bude blíže popsán dále. Při práci s Jenkins je důležité uvědomit si rozdíl mezi nastavením Jenkins serveru a jeho prostředí, a konfigurací jednotlivých projektů, při kterém je definován build proces.

Build proces

Build proces je srdcem práce s Jenkins. V něm jsou definovány nejdůležitější aspekty práce jako propojení s Git či jiným VCS, nastavení způsobu kooperace VCS, nastavení testovacího prostředí, sestavení programu, spouštění, vyhodnocení a zobrazení testů, spouštění dalších pomocných nástrojů a napojení na externí systémy a mnoho dalších. Proces probíhá v těchto základním krocích:

1. **Pre-build** - Tuto fázi lze doinstalovat pomocí pluginu. Slouží většinou k dodatečné konfiguraci testovacího prostředí. Tento krok bývá ale mnohdy automatickou součástí následujícího, proto tato fáze není často potřeba a není ve výchozí instalaci obsažena.
2. **Build** - Hlavní fáze, ve které se provádí kroky jako sestavení, překlad, testování, vyhodnocení a analýzy projektu. Sama se tedy skládá z mnoha kroků, které jsou řízeny konfigurací Jenkins. Nejdůležitějším a časově nejnáročnějším krokem bývá spuštění `build skriptu`. Tento skript si sám definuje několik fází. Jenkins podporuje spoustu populárních build nástrojů (jazyků) jako např. `Ant, Maven, Gradle, Rake, Visual Build` a mnoho jiných. Pro projekt k-Wave jsem použil nástroj `Make`, který je stále často používaný v prostředí nativních jazyků C/C++. V tomto kroku lze také spouštět klasické `Shelové` či `Python skripty`.
3. **Post-build** - Tato fáze slouží především k sesbírání výsledků předchozího kroku a jejich zobrazení ve webovém rozhraní. Můžou se zde provádět další akce jako merge větví kódu (pokud byl předchozí build úspěšný), notifikace v případě neúspěchu, úklid prostředí, aj.

Typy Jenkins projektů

Projekt je základní jednoutkou práce s Jenkins. Po instalaci jsou k dispozici základní typy projektů. Mnoho dalších lze doinstalovat formou pluginů. Po instalaci jsou k dispozici tyto projekty:

- **Freestyle projekt** - Základní typ projektu s kompletní možností konfigurace, sloužící pro potřeby řízení CI. Pomocí pluginů lze nainstalovat i projekty pro další populární build nástroje.
- **External job** - Tento projekt použijete pokud chcete použít Jenkins jen pro jeho Dashboard a zobrazování výsledků buildů. Celý proces je řízen a proveden jiným systémem, který pouze zašle Jenkins data k zobrazení.
- **Multi-configuration projekt** - Slouží pro projekty, kde potřebujete spouštět build vícekrát s různou konfigurací prostředí či podmínek. Nabízí stejnou funkcionalitu jako

Freestyle projekt. Navíc umožňuje definovat *konfigurační matici*. Tu si lze představit jako n-rozměrnou matici, kde každá dimenze udává proměnou. Velikost této dimenze udává počet hodnot, kterých bude proměnná nabývat, hodnoty v daném rozměru pak konkrétní hodnoty proměnné. Jenkins spustí buildy pro všechny kombinace proměnných. Buildy mohou být spouštěny paralelně nebo sériově dle potřeb projektu. Typické užití je např. spuštění stejného projektu na různých slave uzlech, s jinou verzí překladače či knihoven nebo různými proměnnými pro build skript [9].

Distribuovaný build

Jenkins umožňuje distribuovaný build postaven na principu *master-slave* popsaným v kapitole 1.2. Velmi často se používá pro multi-configuration projekty k distribuci velké zátěže a zajištění různých prostředí. Jenkins umožňuje více strategií, jak konfigurovat distribuovaný build v závislosti na operačním systému či síťové architektuře [9].

- Master spouští slave agenty vzdáleně přes ssh.
- Spuštění slave agentů manuálně na slave uzlu přes Java Web Start.
- Instalovat slave agenty jako Windows služby.
- Spouštění slave agentů manuálně z příkazové řádky ze slave uzlu.

Pro linuxové slave uzly se nejčastěji volí možnost spouštění přes ssh. Pro Windows se často volí varianta služby. Manuální spouštění jsou spíše doplňkovým řešením používaným pro testování popřípadě velmi krátké projekty [4].

Napojení na externí systémy a další možnosti Jenkins

Jenkins poskytuje většinu svých funkcí jako webovou službu prostřednictvím `Rest API`. Přes něj lze spouštět jednotlivé buildy, získávat jejich výsledky nebo naopak výsledky na Jenkins zasílat pro zobrazení (využíváno v External job projektech). Obdobný způsob komunikace umožňuje i řada jiných webových služeb. O některých z nich se zmíním v následující kapitole. Komunikace mezi službami poté probíhá prostřednictvím jejich API. Díky velkému rozšíření a komunitě Jenkins nástroje je napojení na spoustu služeb nachystané ve formě pluginů. Uživateli pouze stačí nakonfigurovat vlastnosti pluginu, který se postará o veškerou komunikaci a další operace.

Pluginy umožňují implementovat spoustu dalších funkcí přímo do Jenkins buildů. Často se používají různé statické analýzy, kontrola kvality kódu, tvorba dokumentace, notifikace na email nebo prostřednictvím IM zpráv, aj.

2.5.3 Další CI nástroje

Jenkins samozřejmě není jediným nástrojem na trhu. Existuje spousta dalších většinou komerčních nástrojů se stejným množství funkcí jako Jenkins. Jejich výhodou je především nabízení těchto funkcí formou služby. Uživatelé se poté nemusí starat o zajištění hardwaru a vytváření či správu prostředí. Tato možnost je obzvlášť výhodná v případě využití více slave uzlů. Dalším důvodem může být lepší spolupráce s produkty stejné společnosti, větší podpora a zaměření na konkrétní technologii, popřípadě jednodušší konfigurace na základě předdefinovaných profilů užití.

CruiseControl

Open source projekt s podobnou sadou základních funkcí jako Jenkins. Konfigurace probíhá prostřednictvím xml souborů. Není k dispozici webové konfigurační rozhraní, což činí konfiguraci projektů méně pohodlnou. XML lze tvořit a upravovat ve speciálních nástrojích upravených pro CruiseControl. Podporuje především jazyky Java a C#, pro které existují speciální verze projektu. Obsahuje velmi jednoduchý dashboard, učící křivka je delší než u Jenkins. Díky menší popularitě a komunitě nejsou k dispozici takové množství pluginů, které poskytují rozšířené funkce a komfortnější ovládání.

Apache Continuum

Obdobné řešení jako Jenkins. Opět omezení vznikající z menší komunity aktivních uživatelů.

Buildbot

Velmi zajímavě řešení nabízející jiný přístup než ostatní nástroje, které zde popisuji. Funguje jako *správce úloh*. Ukládá úlohy do fronty, spouští je když jsou k dispozici všechny zdroje a reportuje výsledky. Nabízí distribuovaný build řízený pomocí Python skriptů. V nich se dá využít všech možností jazyka, čímž jsou umožněny složité dynamicky generované konfigurace, přesně zaměřené na požadavky projektu a situace. Lze na něj nahlížet jako na *framework* se sadou předchystaných komponent.

Hlavním cílem je poskytnout nástroj bez jakýchkoliv předsudků ohledně struktury a workflow procesu kontinuální integrace. Uživatel si může vytvořit nástroj přesně zaměřený a přizpůsobený jeho konkrétním požadavkům.

Očekávanou nevýhodou je samozřejmě mnohem delší učící křivka a značné prodloužení zavedení systému do existujícího týmu. Možnosti reportů jsou také mnohem menší než u jiných nástrojů. Každou pokročilejší funkci si musí uživatel naprogramovat sám.

Github CI, Gitlab CI

Nástroje Github a Gitlab jsem již popisoval dříve. Oba nabízejí svá rozšíření pro kontinuální integraci, které jsou ale omezenější než u konkrétně zaměřených nástrojů. To lze především cítit v podpoře projektů jazyků `C/C++`, které nejsou tak masově rozšířené.

Atlassian Bamboo, Jetbrains TeamCity

Jde o ukázku komerčních řešení. Výhodou je nabídka řešení formou služby a silnější vazba na další produkty daných firem. Pro náš projekt by ale využití těchto nástrojů nepřineslo žádný zisk.

2.5.4 Závěr

V této kapitole jsem popsal základní nástroje nutné pro použití technik kontinuální integrace. Důraz jsem kladl především na Git a Jenkins, které jsem zvolil pro tento projekt. Hostování bude probíhat na Gitlab serveru, realizovaném pomocí nástroje `Docker`. Unit testy budou realizovány pomocí `GoogleTest`. Popisované principy jsou použitelné i u dalších nástrojů dané oblasti s mírnými úpravami. Další nástroje jsem spíše přirovnával ke

zvoleným, abych objasnil co mě vedlo k daným volbám a poskytl čtenáři základní přehled v dané problematice.

Kapitola 3

Návrh řešení pro projekt k-Wave

Nejdříve vás v této kapitole velmi stručně seznámím s projektem k-Wave a způsobem správy běhu úloh na superpočítačích. Poté budou uvedeny různé techniky kolaborace s využitím nástroje Git a typická workflow procesu spojité integrace. V poslední kapitole všechny tyto části spojíme dohromady a uvedu návrh řešení projektu k-Wave.

3.1 k-Wave nástroj běžící na superpočítačích

V této části vás stručně seznámím s projektem k-Wave a superpočítači. Komplexní popis by přesahoval rozsah této práce, proto se zaměřím převážně na nejpodstatnější vlastnosti, které ovlivnili návrh, zavedení a způsob použití nástrojů kontinuální integrace.

3.1.1 k-Wave

K-Wave je sada open source nástrojů pro akustickou a ultrazvukovou simulaci. Z pohledu kontinuální integrace jsou důležité tyto vlastnosti projektu a týmu:

- Existuje řada různých verzí produktu mezi které patří verze pro `MATLAB`, `C++`, implementace s využitím `OpenMP` nebo `MPI`, `CUDA` verze a další.
- Některé verze produktu jsou optimalizované pro běh velkých simulací, které jsou prováděny na superpočítačích.
- V současné době na projektu spolupracuje kolem patnácti členů týmu, kteří plní různé role. Členové pochází z různých zemí. Většina spolupráce a komunikace tedy probíhá vzdáleně.
- Je snaha zapojit do projektu studenty. Jejich práce musí být více kontrolována a ověřena.

V současné době není v projektu zaveden žádný způsob kontinuální integrace či automatického testování. Existují různé testovací skripty, psané především v jazyku `Matlab`, které poskytují textové výstupy, které musí projít manuální kontrolou. Testy jsou často založeny na porovnávání výsledků více verzí projektu. Referenčními výsledky, tedy těmi považovanými za správné, jsou výstupy matlabovské verze. Různé verze projektu jsou uchovávány v různých repozitářích, většinou s využitím Gitu. U verzí, které jsou určeny pro běh na superpočítačích je potřeba testovat širokou řadu různých konfigurací, verzí překladače a knihoven, běhových prostředí apod.

3.1.2 Superpočítač Anselm

V této práci jsem zaměřil na verzi projektu k-Wave, která je optimalizovaná pro běh velkých simulací na superpočítačích. Jako vzorový superpočítač mi sloužil Anselm [1]. Cluster se skládá z 209 výpočetních uzlů s 15 TB RAM a teoretickým výkonem kolem 9.3 4Tflops/s. Uzly jsou propojeny `Infiniband` sítí, vybaveny Intel Sandy Bridge procesory. Některé uzly jsou vybaveny NVIDIA Kepler GPU nebo Intel Xeon Phi MIC akcelerátory. Cluster běží na linuxovém operačním systému [2].

Nebudu zde popisovat architekturu superpočítače a způsob práce s ním. Spíše se zaměřím na aspekty, které ovlivňují návrh a způsob použití nástrojů postupného vývoje.

Typy uzlů

Existují dva základní typy uzlů [2]:

- **Login uzly** - Pouze tento typ uzlů má povolen externí přístup, většinou přes `ssh`. Nejsou určeny primárně pro výpočet, ale menší výpočty se na nich provádí. Na tyto uzly se přihlašují uživatelé, kteří spravují a spouští svoje *úlohy (joby)*, nastavují pro ně prostředí a sbírají získané výsledky.
- **Výpočetní uzly** - Na těchto uzlech probíhá samotný výpočet. Nemají umožněnou žádnou externí komunikaci. S těmito uzly tedy nelze komunikovat, ani sestavit program, který by přímo z nich zasílal data na externí server.

Plánování úloh

Uživatelé nemůžou přímo spouštět svoje úlohy. O tuto operaci si žádají systémový plánovač úloh, který zařadí požadavek do některé ze svých front. Existuje více typů front s různou prioritou. Jakmile je úloha na čele fronty a jsou k dispozici potřebné prostředky, je systémem spuštěna. Úloha běží dokud neskončí nebo dokud nevyprší požadovaný čas. Na Anselmu plní tuto roli `PBSPro` manažer [2].

Nastavení prostředí a komunikace s okolím

Uživatel si může upravovat prostředí pomocí konfiguračních souborů. K instalaci potřebných nástrojů slouží rozhraní aplikačních balíčků. Pomocí něj si lze dynamicky nahrát potřebné programy v konkrétních verzích. Uživatel si může vytvářet vlastní balíčky, tzv. *moduly* [2].

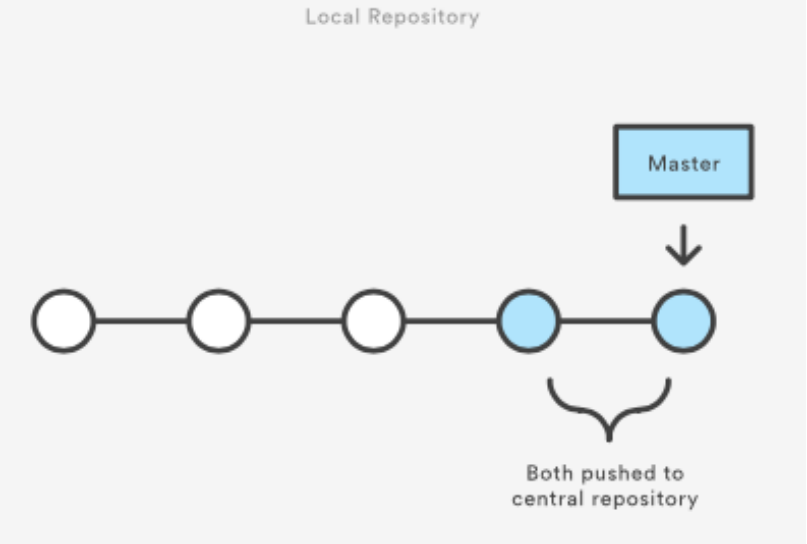
Připojení na Anslem je zajištěné pomocí SSH. Data lze zasílat přes `scp`, případně `sftp` Superpočítač bohužel neumožňuje žádné způsoby komunikace s externími systémy jako například pokročilejší notifikace o stavech úloh nebo veřejné API. Uživatel může zjistit stav úlohy skrze příkaz zadaný na login uzlu, popřípadě je mu při dokončení úlohy zaslán email. Doručení emailu ale není zaručené [2].

3.2 Git workflow

Pod Git workflow zde myslím organizaci práce s Git větvemi. Postupně zde vysvětlím některé populární možnosti od nejjednodušších způsobu po složitější.

Všechny postupy v této části včetně obrázků jsou převzaty z [11].

[1] https://www.it4i.cz



Obrázek 3.1: Centralizovaná workflow

3.2.1 Centralizovaná workflow

Tento způsob použití napodobuje chování centralizovaných systémů jako např. Subversion. V tomto případě existuje pouze jedna větev v origin repozitáři - *master*. Uživatel provádí změny u sebe na lokální stanici a jakmile uzná za vhodné uloží změny do master větve v origin repozitáři. Princip je znázorněn na obrázku 3.1. Existuje pouze jedna větev, která sdružuje verze kódu určené pro vývoj, testování i distribuci. Odlišení těchto verzí je možné pouze s využitím tagů.

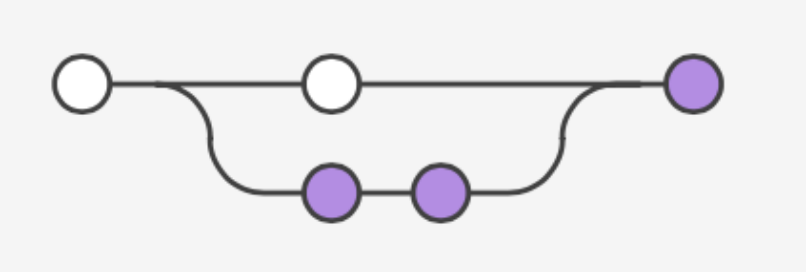
Tento koncept je vhodný pro malé týmy, obzvláště takové, které již mají zkušenosti s centralizovanými VCS a nechtějí přecházet na novou workflow. Výhodou je jednoduchost tohoto řešení. Nevýhodou tohoto přístupu je časté vznikání konfliktů při mergování kódu. S rostoucí velikostí týmu pravděpodobnost vzniku takových chyb rapidně roste. Tento systém také není vhodný pokud pomocí něj chceme zajistit distribuci výsledného softwaru.

3.2.2 Feature Branch workflow

Tato workflow obsahuje kromě master větve ještě tzv. `feature` větev. Při implementování nové funkce, popřípadě opravy kódu, se vytvoří nová feature větev. Každá funkce má tedy svoji vlastní nezávislou větev kódu. Tuto funkci často implementuje jeden vývojář. Větev je v takovém případě určena pouze jemu. Není to ale pravidlem a záleží hlavně na zvyklostech a praktikách týmu a způsobu dekompozice problému.

Tento způsob práce pomáha snižovat vznik situací, kdy si jednotlivý členové týmu pod rukama přepisují kód a zmenšuje dopady řešení konfliktů při mergování kódu. K této fázi dochází totiž až na konci vývoje nové funkce/opravy. Nová větev často vzniká i pro různé experimenty, případně alternativní způsoby implementace. Jakmile je ale jedna z alternativ vybrána, dochází ke sloučení větve. Feature větve by se tedy neměly používat pro trvalé alternativy programu ani různé odlišné části projektu (např. dokumentace a kód v různých větvích).

Princip je znázorněn na obrázku 3.2. Jde o velmi flexibilní způsob práce na projektu, který navíc uchovává jasně oddělené záznamy o práci na jednotlivých funkcích. Zanechává



Obrázek 3.2: Featured based workflow

týmů spoustu prostoru pro přizpůsobení ke svým potřebám a pracovním návykům. U větších projektů ale bývá vhodné přísněji definovat význam jednotlivých větví a práce s nimi.

3.2.3 Gitflow workflow

Toto řešení definuje více typů větví a striktně předepisuje jejich využití. Systém slouží současně pro spojitou distribuci kódu a je znázorněn na obrázku 3.3.

Existují tyto typy větví:

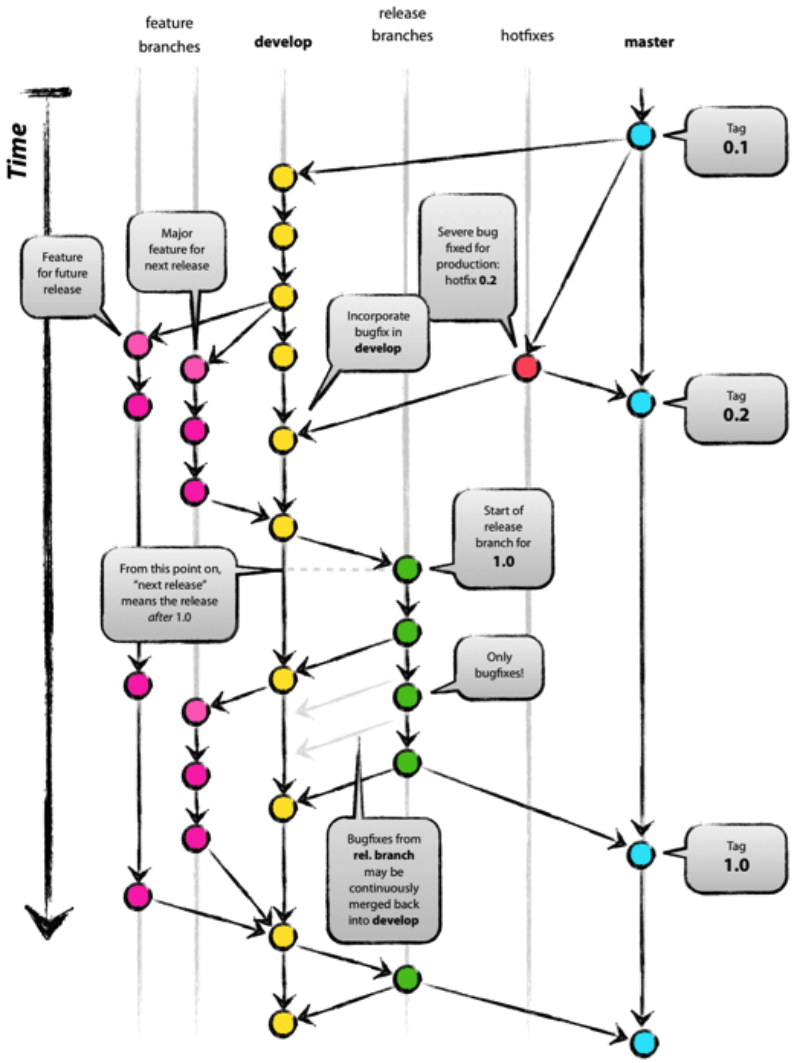
- **Develop** - Obsahuje vždy aktuální verzi kódu určenou pro vývoj. Každý projekt obsahuje právě jednu develop větev. Malé úpravy kódu jsou povoleny přímo v této větvi bez nutnosti vytvoření feature větve. Snahou je udržet tuto větev stále stabilní a aktuální, aby měli vývojáři k dispozici čerstvou verzi funkčního kódu.

- **Feature** - Plní podobnou funkci jako ve Feature Branch workflow. Každý vývojář si při implementaci nové funkce, popřípadě větší opravy, vytvoří svoji feature větev z develop větve. V systému tedy může existovat více takovýchto větví. Jakmile je práce ve větvi dokončena je snaha ji co nejdříve mergovat do develop větve, aby se udržovala develop větev co nejvíce aktuální. Před tímto mergem se často provádí sada automatických testů, což napomáhá udržení stabilní develop větve.

- **Release** - Jakmile je kód v develop větvi ve stavu vhodné pro vydání nové verze, vytvoří se tato větev, která je mezikrokem mezi develop a master větví. Tento krok je do systému přidán aby se před vydáním mohla provést zvláštní sada automatizovaných popřípadě ruční testů a revizí kódu před vydáním ostré verze. Jsou v ní povoleny rychlé opravy chyb. U větších chyb by měla realease větev zaniknout a z develop větve by měla vzniknout feature větev pro opravu. Při jejím dokončení dochází k mergi do master i develop větve současně.

- **Master** - Tato větev obsahuje jednotlivé verze programu určené k distribuci. S každým zápisem do této větve by měl vzniknout tag, udávající verzi. V této větvi nesmí vznikat žádné úpravy kódu ani z ní nesmí vznikat jiné větve kromě hotfix větví. Tato větev slouží striktně k přehlednému uchování vydaných verzí softwaru. Projekt může obsahovat maximálně jednu master větev. Je také hlavní větví při použití spojité distribuce kódu. Každá verze v této větvi se může automaticky distribuovat. Jednotlivé commity do ní mohou sloužit jako spouštěče těchto automatických událostí.

- **Hotfix** - Slouží pro rychlou opravu chyb, které se odhalili až v master větvi a také je jediným způsobem jak upravit kód přímo z ní. Po dokončení větve se její kód merguje do master i do develop větve, stejně jako v release větvi.



Obrázek 3.3: Gitflow workflow

Jak jde vidět systém je značně komplikovanější než předchozí a má přesně specifikované způsoby využití jednotlivých větví. Tato vlastnost mu bývá mnohdy vyčítána obzvláště u menších týmu. Zachovává ale velmi dobrou a přehlednou organizaci kódu a je výhodný pro automatické testování, které se provádí typicky před mergem feature větve a v release větvi.

3.2.4 Workflow pro k-Wave

Projekt počítá s využitím Gitflow workflow. Do budoucna se počítá s přidáním další větve, kterou jsem nazval `pre-develop`. Bude sloužit k manuální kontrole jednotlivých feature než dojde k jejich sloučení s develop větví. Tento merge bude prováděn manuálně, vybraným členem týmu. Na projektu se totiž mohou podílet členové, kterým ještě nebyla přiřazena dostatečná důvěra k tomu, aby mohli ovlivnit stabilní vývojovou větev, ze které čerpají kód další členové týmu (například studenti).

3.3 Typická workflow procesu kontinuální integrace

V této části popíši typickou workflow pro postupnou integraci s využitím popisovaných nástrojů Git, Gitlab a Jenkins, která je znázorněna na obrázku 3.4.

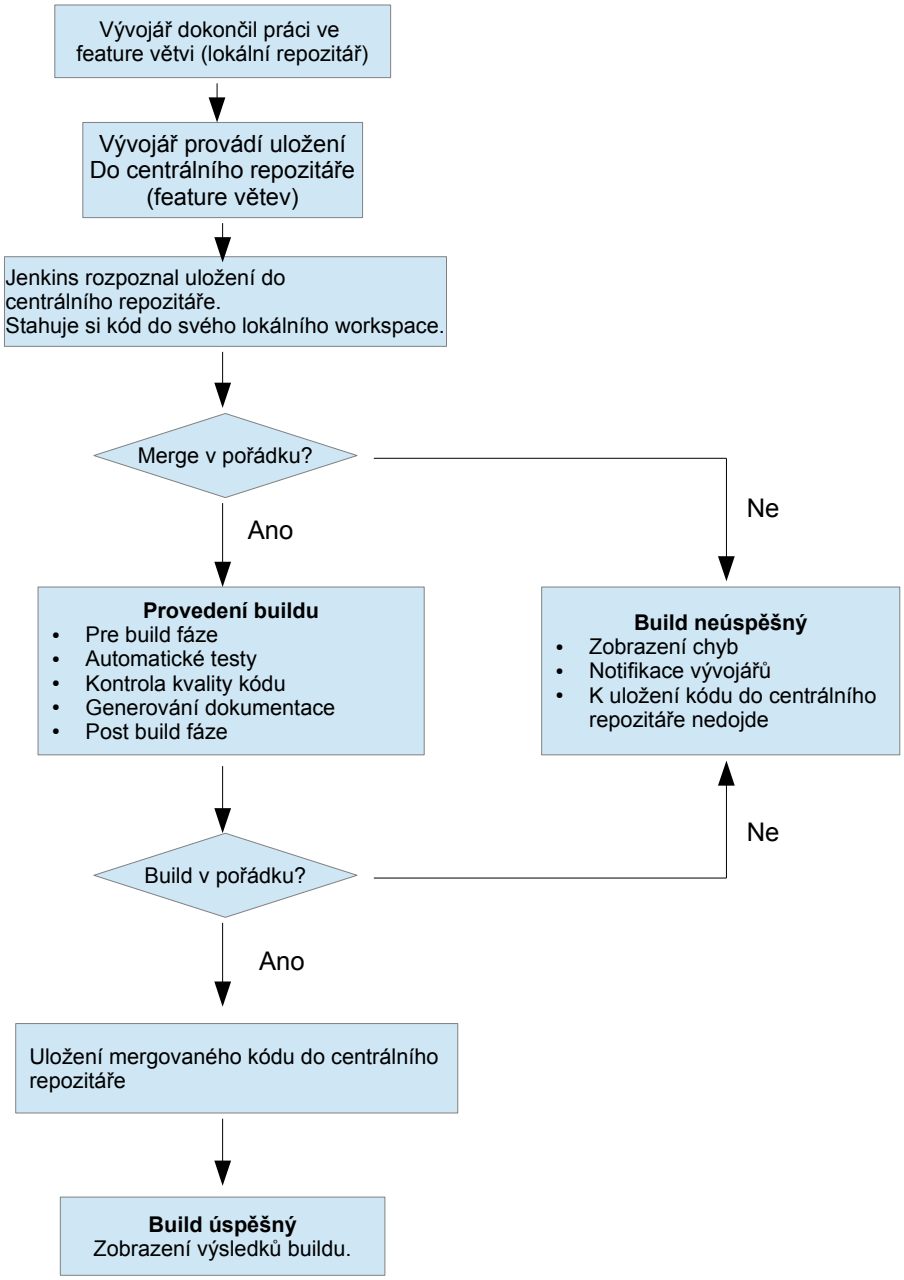
Z obrázku je princip jasný, zaměřím se zde na pár důležitých bodů této workflow. Pro Git je použita Gitflow workflow. Scénář popisuje běžné vytvoření nové funkce ve feature větvi. Jenkins je schopný automaticky rozpoznat změny této větve v centrálním repozitáři. Jakmile se o této změně dozví, stáhne si příslušnou feature větev do svého workspace a celý build probíhá nad touto lokální kopií. Proto lze bez starostí před vlastním buildem provést merge do develop větve (opět v lokálním workspace). Až nad takto sloučeným kódem se provádí testy a další operace. Pokud v procesu dojde k jakékoliv chybě, je ukončen s chybou. Notifikují se příslušní uživatelé a k uložení do centrálního repozitáře nedojde. Pokud je celý proces v pořádku, uložení proběhne a výsledky se zobrazí uživateli.

Tato workflow je nepřerušitelná. Spousta kroků musí být prováděna sekvenčně za sebou. Nelze ji tedy v průběhu pozastavit a pokračovat v ní na základě externí notifikace. Toto je jeden ze základních problémů při běhu programů na superpočítačích.

3.4 Návrh workflow pro k-Wave

Z popisu dřívějších kapitol lze vyčíst, že pro běh projektu k-Wave na superpočítačích podobných Anselmu, nelze použít běžnou workflow popisovanou v předchozí kapitole. Pro přehled zde shrnu všechny kritické body:

- Správa jobů na superpočítačích (nelze jednoduše spustit skript na výpočetním uzlu a čekat až doběhne).
- Není možná přímá komunikace výpočetních uzlů prostřednictvím internetu.
- Omezené možnosti notifikace o dokončení jobu superpočítače.
- Omezení login uzlu jako například omezené hardwarové a výpočetní prostředky. Navíc existuje možnost, že náš proces bude po dlouhém běhu násilně ukončen běhovým prostředím. Na druhou stranu je to jediný uzel, který nám je schopný zaslat data přes internet.



Obrázek 3.4: Běžná workflow procesu CI

- Softwarové prostředky a nastavení prostředí na superpočítači. Na superpočítači máme omezená oprávnění. Nemůžeme provádět systémová nastavení a instalovat můžeme pouze do lokálních adresářů.

3.4.1 Možná řešení problémů s workflow

V této části uvedu možná řešení jednotlivých problémů, které jsem navrhl na základě uvedené literatury a experimentování s nástroji.

Testování na login uzlu

Malé úlohy lze spouštět přímo na login uzlu bez plánovače jobů. Vhodné pouze pro základní a jednoduché projekty. Neotestujeme totiž běh v paralelním prostředí s potřebnou výpočetní silou ani specifické hardwarové požadavky jako GPU procesory atd. Můžeme ale otestovat nastavení prostředí, komunikaci se serverem a provést jednoduché testy.

Sekvenční test jedním Jenkins projektem

Jde o náročnější řešení, které probíhá v těchto krocích.

1. Konfigurace prostředí na superpočítači.
2. Zažádání o provedení testu na výpočetních uzlech. Test bude zařazen do fronty.
3. Spuštění kontrolního skriptu. Ten má za úkol monitorovat zda test, o jehož spuštění se žádalo v předchozím kroku, již doběhl.
4. Jakmile dojde k ukončení testu, skript se ukončí a může se přejít do post build fáze.
5. Vyhodnocení a zaslání výsledků Jenkins serveru.

Řešení je znázorněné na obrázku 3.5. Hlavní nevýhodou je, že nemáme jistotu, kdy se skutečně test spustí a build díky tomu může trvat opravdu dlouho (klidně hodiny). V takové situaci může dojít k násilnému ukončení buildu ze strany Jenkins serveru nebo prostředí superpočítače.

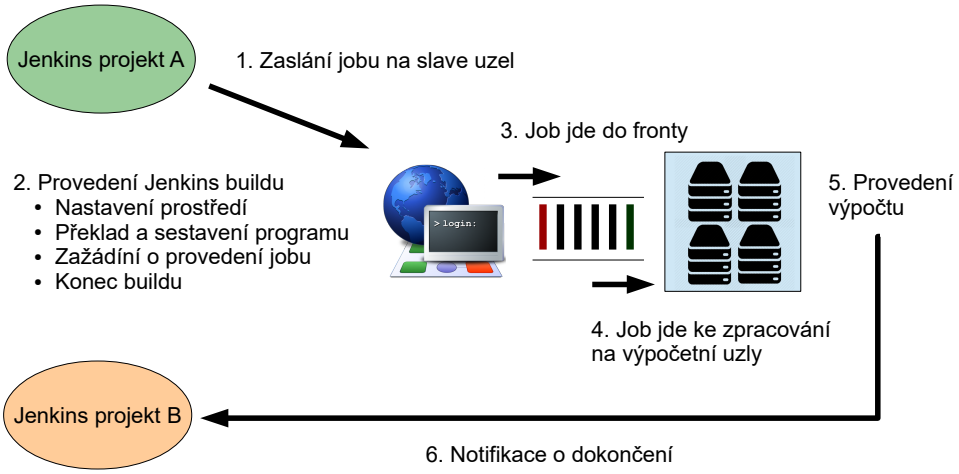
Paralelní test s více Jenkins projekty

Tento přístup již vyžaduje nakonfigurování dvou Jenkins projektů. Postupně se provádí:

1. Spuštění prvního Jenkins projektu.
2. Konfigurace prostředí na superpočítači.
3. Zažádání o provedení testu na výpočetních uzlech. Test bude zařazen do fronty.
4. Spuštění kontrolního skriptu v paralelním vlákně. První Jenkins projekt svůj build končí. Skript na superpočítači stále běží.
5. Ukončení testu. Skript notifikuje Jenkins server, který spustí paralelní Jenkins projekt. Možnost více způsobu notifikací. Nejjednodušší je notifikace přes Jenkins Rest API.
6. Tento druhý Jenkins projekt vyhodnotí a zobrazí výsledky uživateli.

Tento přístup je složitější na konfiguraci a méně příjemný a přehledný pro uživatele. Stále hrozí ukončení kontrolního skriptu na superpočítači. Přístup je znázorněný na obrázku Řešení je znázorněné na obrázku 3.6.

Obrázek 3.5: Sekvenční build s jedním Jenkins projektem



Obrázek 3.6: Build s více Jenkins projekty a notifikací

Paralelní test s notifikací emailem

Obdobné řešení jako předchozí, které ale využívá možností Anselmu zaslat upozornění o dokončení jobu emailem. Je možnost toto upozornění odchytnout a spustit jim druhý Jenkins projekt. Zaslání emailu ale není zaručeno.

3.5 Návrh řešení pro k-Wave

V této části shrnu údaje, které byly popsány v jednotlivých předchozích kapitolách. Pro projekt k-Wave jsem zvolil tyto nástroje a postupy:

- Git pro správu verzí zdrojových kódů s Gitflow workflow. Centrální repozitář bude hostovaný na Gitlabu nasazeném na soukromém serveru.
- GoogleTest pro tvorbu automatických testů jednotek.

- Jenkins pro řízení procesu kontinuálního vývoje a zobrazování výsledků.
- Jenkins buildy budou spouštěny manuálně přes web nebo automaticky při commitu do feature nebo release větve v origin repozitáři (push událost). Při dokončení některých testů může být proveden automatický merge do develop větve.
- Testování k-Wave bude probíhat na superpočítači Anselm. Použije se test na login uzlu a sekvenční test s jedním Jenkins projektem.
- Bude využito více typů Jenkins projektů. Pro jednodušší testy klasický Freestyle projekt. Pro testy s různou verzí konfigurace Multiconfiguration projekt.
- Buildy budou distribuované, spouštěné přes ssh.
- Do budoucna se počítá s využitím různých nástrojů pro generování dokumentace a kontrolu kvality kódu.
- Jenkins a Gitlab poběží jako Docker kontejnery.

Potencionálním rizikem zavedení metodik a postupů, popsaných v této práci, je dočasné snížení efektivity jednotlivých členů týmu, kteří se budou muset naučit novému stylu práce. Proto bude kontinuální integrace zaváděna inkrementálně po menších krocích, které budou vývojáři snadněji zavádět do své práce. Tento projekt poskytuje první kroky této integrace. Při návrhu a implementaci jednotlivých částí řešení jsem se snažil co nejméně vzdalovat zavedeným praktikám týmu. Proto jsem použil některé stávající testovací skripty a postupy. Některá řešení jsou proto komplikovanější, méně přívětivé, popřípadě není jejich architektura tak robustní jak by mohla být. Budou ale mnohem lépe zapadat do práce týmu, což je na začátku zavedení nové metodiky mnohem důležitější. Řešení je navrženo s ohledem na budoucí rozšíření a vylepšení.

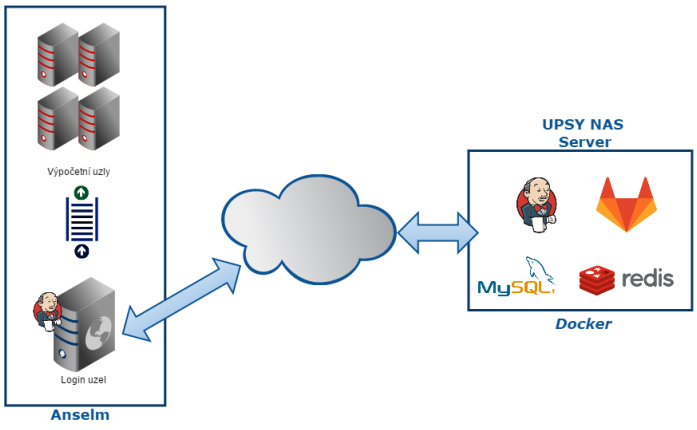
Kapitola 4

Implementace postupné integrace do projektu k-Wave

V této kapitole uvedu postup implementace postupů a nástrojů postupné integrace do projektu k-Wave. Nejdříve popíši zavedení a správu prostředí pro kontinuální integraci, dále instalaci a konfiguraci jednotlivých nástrojů. Popíši zde i realizaci některých regresních a unit testů, která byla součástí mého zadání.

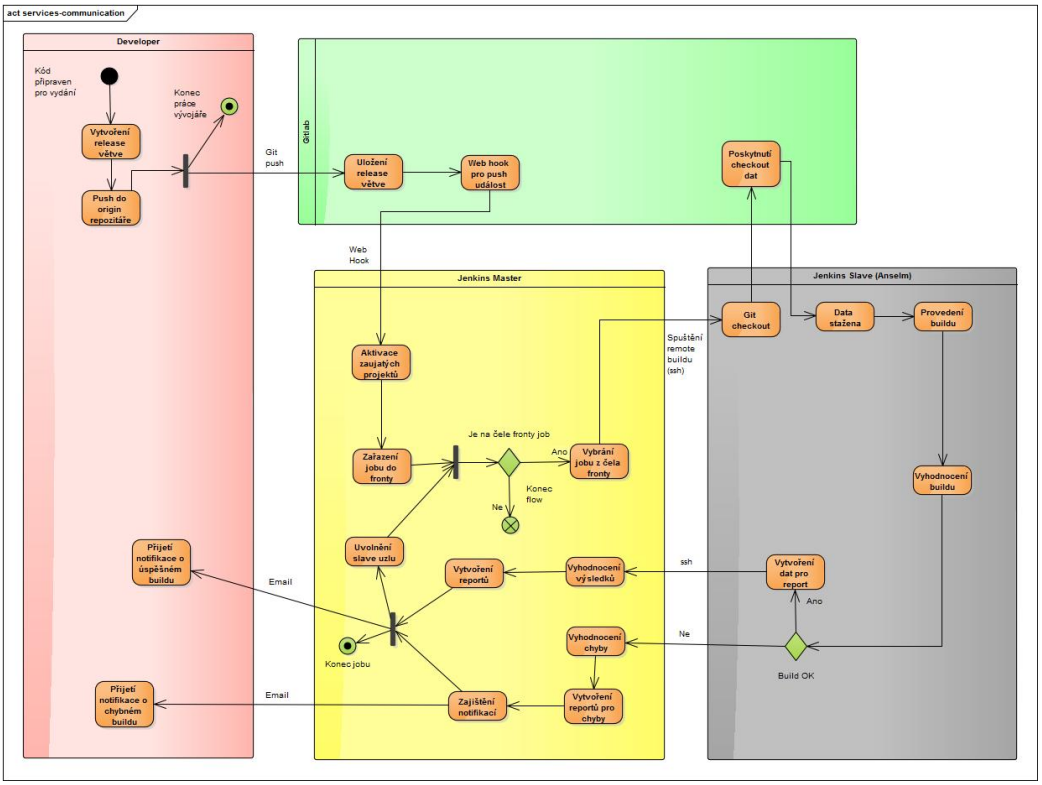
4.1 Prostředí kontinální integrace

Pod pojmem prostředí kontinuální integrace mám na mysli hardwarové a softwarové komponenty, nezbytné pro hladký běh kontinuální integrace projektu. Schéma lze vidět na obrázku 4.1. Automatické buildy se provádí vzdáleně s využitím master-slave architektury.



Obrázek 4.1: Architektura řešení

V této části budu popisovat administraci serveru na kterém běží používané služby, konfiguraci služeb samotných a nastavení kooperace mezi nimi. V příloze B je myšlenková mapa zobrazující nastavení, prováděná na jednotlivých službách. Služby, včetně komunikace mezi nimi, jsou také znázorněny na diagramu 4.2. Obrázek také znázorňuje obecný pohled



Obrázek 4.2: Komunikace služeb během Jenkins buildu

na kroky, které se při kontinuální integraci provádí. Pro nastavení znázorněného chování je potřeba nakonfigurovat:

- Samotný server na kterém služby běží. Toto nastavení je v kapitole 4.1.1.
- Nastavení Gitlab služby. Tato konfigurace zahrnuje zpřístupnění Git repozítáře, správu wiki a tzv. *issues*. Pro některé scénáře je třeba zajistit *web hook*, který inicializuje spuštění Jenkins buildu. Více v kapitole 4.2.
- Nastavení Jenkins služby, která pokrývá konfiguraci celého build procesu, komunikaci se slave uzly, kde se build provádí, vyhodnocení výsledků a vytváření reportů, notifikace uživatelů při úspěšném či neúspěšném buildu a mnoho dalších. Také je zde nastavena komunikace s Gitlab serverem, odkud si může Jenkins stahovat zdrojové soubory a případně uložit sloučené verze kódu z různých větví. Tomuto nastavení se věnuje kapitola 4.3.
- Dále je potřeba provést uživatelské nastavení pro přístup ke Git repozitáři a umožnit vzdálené připojení na slave uzlu. Tomuto nastavení se v této práci věnovat nebudu.

4.1.1 Server

Veškeré služby běží na jediném serveru, na obrázku popsaném jako *UPSY NAS Server*. Služby navíc nejsou přímo instalované na stanici, ale běží jako Docker kontejnery. Toto řešení sebou přináší jednu velkou výhodu. Docker se stává poměrně populárním nástrojem,

který je podporován na různých architekturách. Nám to umožnilo zprovoznit celé prostředí na NAS serveru od firmy Synology.

Zkratka NAS znamená *Network-attached storage*, tedy datové uložiště na síti. Nejdříve se NAS používal pouze ke sdílení dat. Časem se ale začal používat i pro poskytování dalších služeb, nejčastěji jednoduchých HTTP serverů. Zavedením podpory Dockeru ale možnosti použití značně rostou, což dokazuje i tato práce. Můžete na něm v podstatě zprovoznit jakoukoliv službu, pro kterou je k dispozici Docker image, popřípadě si vytvořit image vlastní. Omezují vás pouze hardwarové možnosti NAS serveru [12].

S hardwarovým omezením jsem se musel vypořádat i v tomto projektu. Většina služeb je psaná v jazyce Java a příliš nešetří zdroji, především operační pamětí. Tento problém se projevoval velmi pomalým během služeb. Po řadě testů a analýz jsme se s vedoucím práce rozhodli navýšit RAM paměť, což náš problém vyřešilo.

4.1.2 Nastavení prostředí

Synology NAS server lze ovládat přes webové rozhraní *Disk Station Manager (DSM)* nebo pomocí ssh. Instalace balíčků se provádí prostřednictvím `Centra balíčků`. Přes něj lze nainstalovat Docker a další aplikace. Většina aplikací se instaluje jako předkonfigurovaný Docker kontejner. Tento přístup jsem ale raději nepoužíval, jelikož zpřístupňuje pouze minimum možných nastavení.

Správu Dockeru jsem prováděl těmito dvěma způsoby

- **Ssh připojení** - Docker poskytuje sadu nástrojů, pomocí kterých se dá ovládat přes příkazovou řádku. Na NAS serveru to ale často způsobovalo nestabilitu prostředí. Proto jsem tento způsob používal spíše při řešení potíží.
- **Grafická aplikace** - DSM obsahuje grafického správce, přes kterého lze stahovat jednotlivé image, ty poté spravovat a vytvářet kontejnery s jejich konfigurací. Utilita poskytuje veškerá potřebná nastavení, ale potýká se s nestabilitou obzvláště při připojení konzole do Docker kontejneru. Obecně ale byla stabilnější než správa přes ssh.

Na každém kontejneru se dají konfigurovat tyto položky:

- **Porty** - Určuje jak se budou mapovat porty kontejneru na veřejné porty hostující stanice.
- **Svazky** - Soubory a adresáře uvnitř kontejneru lze mapovat na svazky hostující stanice. Data jsou poté k dispozici i když kontejner neběží a dají se snadno zálohovat či přenášet na jiné stanice.
- **Odkazy** - Pomocí odkazů lze propojovat jednotlivé kontejnery mezi sebou. Já tak propojoval například MySQL databázi s Gitlab kontejnerem. Vždy když se uvnitř Gitlab kontejneru zavolá příkaz mysql, spustí se přímo příkaz kontejneru MySQL.
- **Proměnné prostředí** - Pomocí nich se provádí konfigurace prostředí kontejneru. Každá image má většinou Github stránku s návodem jaké proměnné prostředí se mohou nastavovat a jakým způsobem.

Detailní popis nastavení naleznete v příloze A. Nastavení proměnných prostředí je potencionálním rizikem. Některé proměnné uchovávají citlivá data jako například přístupové

údaje. Nastavení kontejnerů lze vždy přečíst z Grafické konzole Dockeru. Proto je potřeba zabezpečit přístup na Docker DSM, případně vytvořit nové image, které budou mít tyto údaje v sobě přednastavené.

4.2 Nastavení Gitlabu

Následující podkapitoly obsahují popis nastavení Gitlab serveru. Spolu s využitím odkazovaných zdrojů lze kapitolu použít jako uživatelskou příručku správce serveru.

4.2.1 Nastaveni Docker kontejneru pro Gitlab

Vytvoření a nastavení Gitlabu bylo nejsložitější z použitých Docker kontejnerů. Oficiální Docker image se mi vůbec nepodařilo na NAS serveru zprovoznit. Z průzkumu a testování dostupných image mi nejvíce vyhovovala `sameersbn:gitlab`, která nabízí pokročilé možnosti nastavení prostřednictvím proměnných prostředí, dobrou dokumentaci a stabilní běh.

V této image se mi nepodařilo nastavit SMTP server, aby přijímal certifikát podepsaný vlastní certifikační autoritou, který používá email na fakultě informačních technologií. Je tedy potřeba použít emailovou službu s uznávaným certifikátem. V tomto projektu jsem použil Gmail. Tento emailový účet bude sloužit pro zasílání upozornění a přístupů uživatelům Gitlabu.

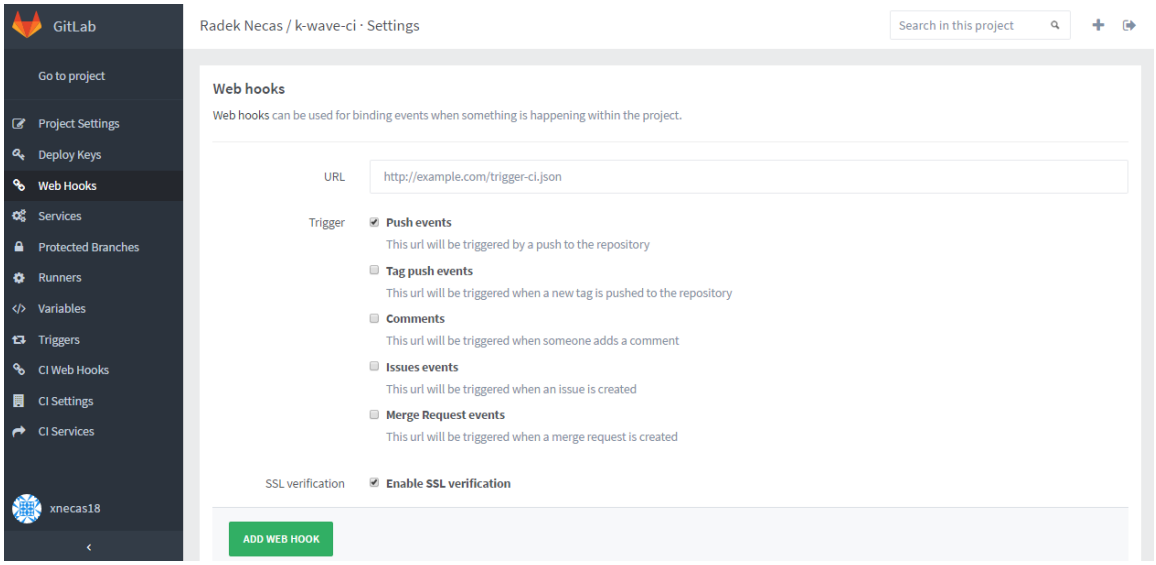
Problémy s nastavením image jsem zkoušel obejít i využitím kontejnerů s Linuxovou distribucí, do které bych si nainstaloval potřebné služby. Vyzkoušel jsem různé distribuce. Hlavním problémem byla složitá konfigurace a problémový běh komponent. Tyto image jsou velmi omezené oproti klasickým instalacím. Většina běžných instalačních postupů pro mě nebyla použitelná, potřebné služby a jejich části nechtěly startovat nebo se samovolně ukončovaly. Výkon tohoto řešení také nebyl vyhovující. Proto jsme se rozhodli vést služby v samostatných kontejnerech, což odpovídá doporučovaným konceptům práce s Dockerem.

4.2.2 Nastavení služby

Veškerou konfiguraci jsem prováděl přes webové rozhraní. Přístup ke Git repozitáři byl zajištěný přes HTTP, kvůli problémům se směrováním SSH přístupu. Důležitým krokem bylo nastavení portu. Login uzel na Anselmu umožňuje komunikovat pouze přes povolené porty. Z nabízených možností se dal použít pouze port 9418. K autentizaci bylo využito uživatelské jméno a heslo. Gitlab CI jsem pro projekt nepoužíval, proto jej nebylo potřeba konfigurovat. Dále se nastavení obešlo pouze s menšími úpravami, které zde nebudu popisovat.

Práce s issues

V současném řešení předpokládám pouze základní práci s Gitlab issues. Ty budou vytvářeny přes webové rozhraní, přes které mohou být i ručně dokončeny. Implementace podporuje i automatické dokočení issue na základě commitu či požadavku na merge v origin repozitáři. Pro využití této možnosti se používá speciální formát zprávy, která se připojí ke commitu. Pokud tedy bude chtít vývojář s commitem automaticky dokončit issue, musí přidat vhodný komentář. Komentář musí obsahovat anglický text, odpovídající následujícímu regulárnímu výrazu:
`((?:[Cc]los(?:e[sd]?|ing)|[Ff]ix(?:e[sd]|ing)?) +(?:(?:issues? +)?%issue_ref (?:(?:, *| +and +)?))+)`.



Obrázek 4.3: Nastevení web hook pro Gitlab

Takže například příkaz: `git commit -m "My commit (Fix #20, Fixes #21 and Closes group/otherproject#22). This commit is also related to #17 and fixes #18, #19 and https://gitlab.example.com/group/otherproject/issues/23."` ukončí issues 18, 19, 20 a 21 projektu, do kterého byl commit proveden a issues 22 a 23 v odpovídajícím projektu group/other/otherproject. Issue s číslem 17 zůstane neovlivněna, protože neodpovídá výrazu.

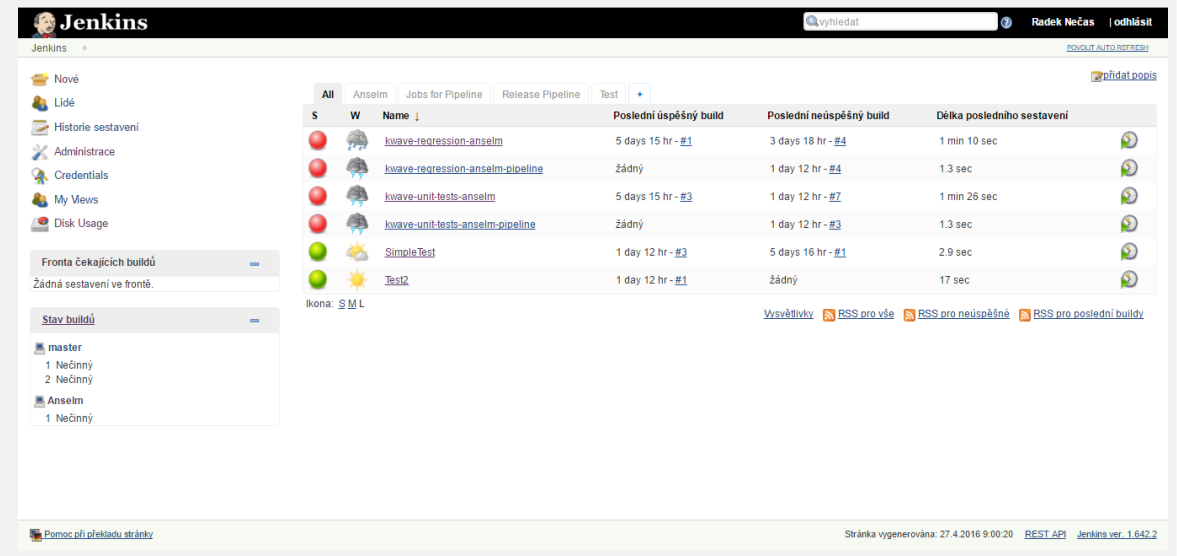
Automatické dokončení issue s využitím výše popsaných komentářů je k dispozici v základní konfiguraci Gitlab serveru. Není teda potřeba žádné zvláštní nastavení. V případě potřeby lze upravit regulární výraz dle vlastních potřeb. Výchozí nastavení je ale poměrně intuitivní, proto jsem úpravy neprováděl.

Propojení s Jenkins

Dle návrhu řešení je potřeba zajistit, aby commit do feature či release větve automaticky spustil odpovídající Jenkins buildy, které provedou sadu testů a dalších operací. Pokud vše projde v pořádku dojde v případě feature větve k automatickému mergi do develop větve. Pro zajištění takového chování je potřeba provést konfiguraci jak na straně Gitlabu, tak na straně Jenkins serveru. Nastavení Jenkins serveru bude probráno v kapitole 4.3.2.

Aktivním prvkem je při této spolupráci Gitlab. Ten inicializuje build na Jenkins serveru. K tomu využívá tzv. *web hook*. Jde o automaticky spouštěnou událost, která se postará o zaslání http zprávy určenému cíly. Zpráva sebou nese řadu různých informací ve formátu `json`, který je k vidění v příloze C.

Konfigurace web hooku je velmi jednoduchá. Provádí se v nastavení Gitlab projektu, kartě `Web Hooks`, která jde vidět na obrázku 4.3. Stačí zadat cílovou URL adresu a spouštěč akce (`trigger`). V našem případě použijeme adresu Jenkins serveru (ve tvaru http://jenkins-url/gitlab/build_now - viz. kap. 4.3.2) a jako trigger použijeme push událost (`Push events`). Jenkins se již postará o zbytek.



Obrázek 4.4: Jenkins dasboard

4.3 Nastavení Jenkins serveru

Nastavení Docker kontejneru pro Jenkins bylo bezproblémové. Proto jej zde popisovat nebudu a zaměřím se na nastavení služby jako takové. Toto nastavení jsem prováděl přes webové rozhraní.

Jenkins web používá jednotnou šablonu, s hlavním obsahem, levým bočním panelem, záhlavím a zápatím. Výchozí stránka se nazývá Dasboard a obsahuje základní informace o všech projektech. Hlavní část stránky zabírá tabulka s tzv. `pohledy`, které sdružují projekty do skupin a u každého zobrazuje základní informace. Dashboard lze vidět na obrázku 4.4.

V Jenkins lze provádět tyto kategorie nastavení:

- Globální nastavení Jenkins serveru, které obsahuje obecná nastavení využita serverem a napříč všemi buildy. Toto nastavení je přístupné z bočního panelu dashboardu .
- Nastavení Jenkins projektů, které je přístupné z bočního panelu stránky konkrétního projektu, která je zobrazena na obrázku 4.6. Na tuto stránku se lze dostat kliknutím na název projektu v pohledu Jenkins dashboardu.
- Nastavení pohledů, které slouží k lepšímu uspořádání projektů. Toto nastavení budu popisovat pouze v části zabývající se pipeline buildy (kapitola 4.3.2).

4.3.1 Globální nastavení Jenkins serveru

Toto nastavení obsahuje obecné vlastnosti platné pro službu samotnou nebo napříč všemi Jenkins projekty. Provádí se ze sekce `Administrace`, která je k dispozici z bočního panelu dashboardu (viz obr. 4.4). Obsahuje vlastnosti jako Nastavení zabezpečení, správu pluginů, monitorování zátěže a mnoho dalších. Já zde popíši pouze nastavení, která byla pro tento projekt nejdůležitější.

Zabezpečení

Zabezpečení serveru se týkají tyto položky:

- **Globální nastavení** - Umožňují nastavit metodu zabezpečení serveru. Je rozdělena do dvou částí `Security Realm` a `Autorizace (Authorization)`. Realm udává, jakým způsobem budou k dispozici uživatelské údaje. Nejzajímavější varianty jsou autentizace pomocí vlastní Jenkins databáze uživatelů a autentizace pomocí LDAP serveru. Já použil metodu vlastní Jenkins databáze uživatelů. Autorizace poté určuje jak budou uživatelé autorizování. Zde jsou vhodné varianty přihlášení uživatelé mohou provádět vše nebo využití zabezpečovací matice, kde nastavíte každému uživateli konkrétní oprávnění (lze rozšířit i o možnost upřesnění projektu).
- **Správa oprávnění** - Zde se nastavují ověření, která použije Jenkins při práci s dalšími systémy. Možné je použití uživatelského jména a hesla, uživatelského jméno a SSH klíče, či certifikátů. Pro projekt jsem použil různé varianty dle možností externího systému. Autentizace pro Anselm slave uzel probíhala pomocí uživatelského jména a SSH klíče, komunikace s Gitlabem pomocí uživatelského jména a hesla.
- **Správa uživatelů** - Zde můžete spravovat uživatele, přiřazovat jim SSH klíče, API tokeny přístupová hesla atd.

Možnosti zabezpečení se dají dále rozšiřovat pomocí různých pluginů.

Správa uzlů

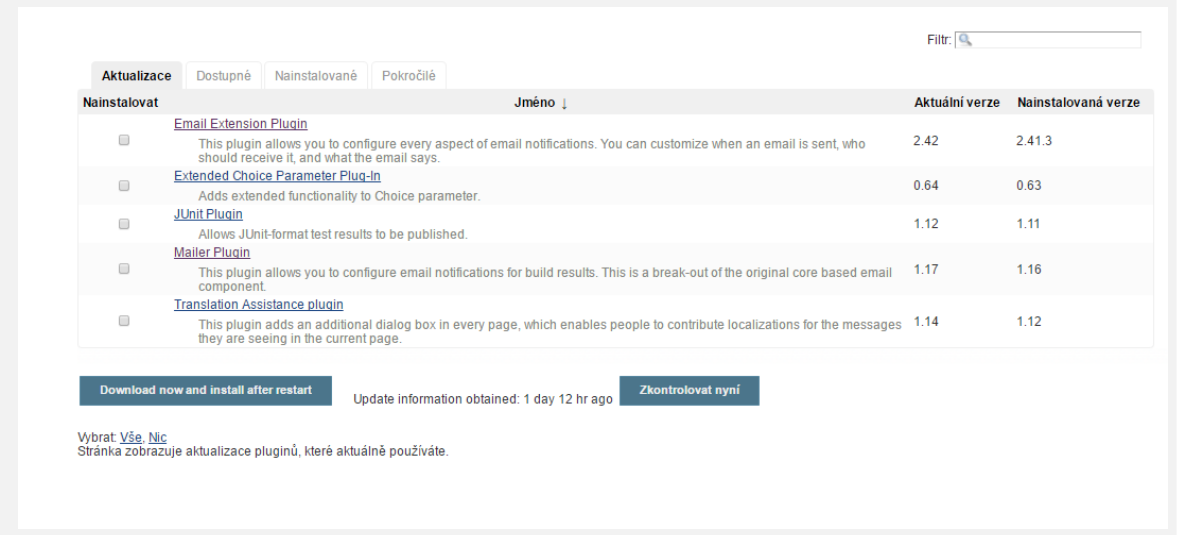
V tomto nastavení lze spravovat vzdálené uzly, které se používají při distribuovaných buildech (viz. 2.5.2). Při vytváření nového uzlu lze provést sadu nastavení, které poté ovlivní možnosti další správy. Já používal jednoduché uzly, které jsem udržoval online jak bylo možné a nebo je spouštěl online až při spuštění buildu, který je používal, a po jeho skončení odpojil (spouštění a odpojení je automatické). Důležitou vlastností je možnost přiřazovat uzlům `label`. Pomocí něj můžete sdružovat uzly do skupin. Ukázkovou skupinou může být skupina uzlů s operačním systémem Windows a další skupina s Linuxem. Při konfiguraci buildu se poté udává právě tento label. Můžete tedy s jedním buildem spustit naráz 10 uzlů s Linuxem, ale různou konfigurací. V tomto projektu jsem neměl k dispozici tolik výpočetních stanic, proto jsem vždy měl nastavený zvláštní label pro každý uzel. V praxi bývá ale tato možnost velmi často využívaná.

Z tohoto nastavení lze uzly i monitorovat, spouštět na nich vzdáleně příkazy, vypisovat jejich konfigurace, zjišťovat jejich historii atd.

Správa pluginů

Práce s pluginy je jednou z hlavních výhod Jenkins serveru. Samotná instalace obsahuje pouze základní sadu možností. Veškeré pokročilejší funkce se systému dodávají formou pluginů. Jelikož jde o velmi rozšířený open source projekt, existuje opravdu mnoho pluginů, poskytující různé funkce od jednoduchých možností úpravy zobrazení informací po velmi pokročilé pluginy, které umožňují a řídí spolupráci s externími systémy.

Důležité pluginy pro tento projekt jsou sepsány v příloze D. Jenkins obsahuje správce pluginů, který umožňuje jejich snadné vyhledávání, instalaci, aktualizaci či mazání. Správce lze vidět na obrázku 4.5. Vlastnosti konkrétních pluginů se pak nastavují dle jejich typu. Pluginy, které ovlivňují přímo jednotlivé buildy se nastavují v nastavení buildů, globální pluginy v globálním nastavení atd.



Obrázek 4.5: Jenkins správce pluginů

Monitorování zátěže

Monitorování zátěže je důležitou součástí správy každého serveru. Pro Jenkins lze toto monitorování rozdělit do dvou kategorií:

- Obecné monitorování zátěže. Pro tento projekt bylo velmi důležitý kvůli omezeným výpočetním možnostem platformy (Docker kontejner na NAS serveru). Prováděl jsem jej s využitím pluginu `Monitoring`, který umožňuje sbírat řadu charakteristik a prezentovat je formou tabulek či grafů.
- Vytížení disku. To je velmi důležitou charakteristikou. Moderní agilní přístupy provádí řadu velmi rozsáhlých automatických testů a dalších operací. Každá z nich může ukládat velké množství dat. Snadno poté může dojít k vyčerpání kapacity disku. Pro toto monitorování je vhodný plugin `disk-usage`. Ten umí zobrazit trendy vytížení disku celého serveru i pro jednotlivé projekty.

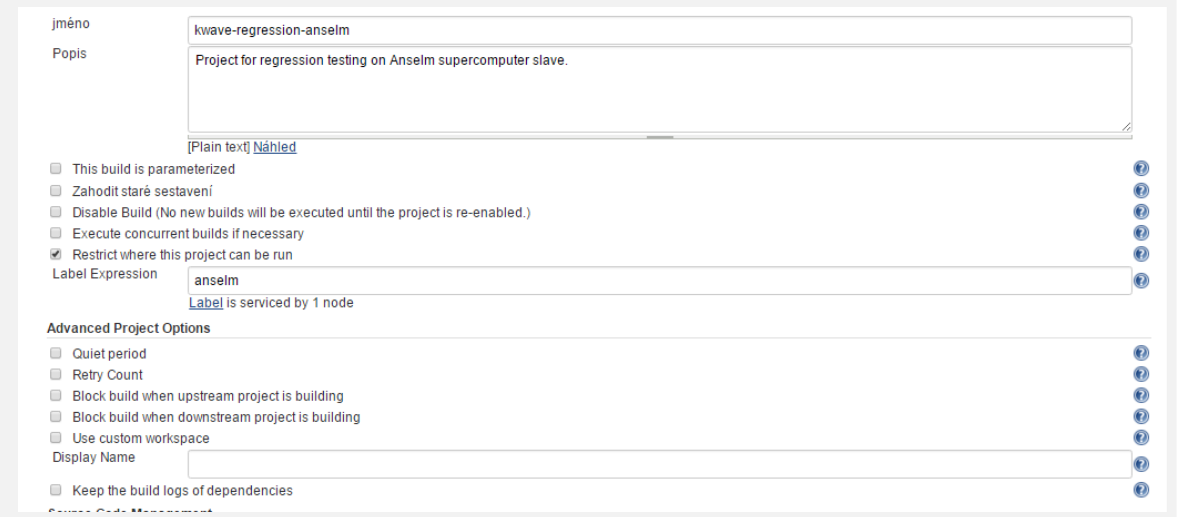
4.3.2 Nastavení Jenkins projektů

V této části popíši nejpodstatnější nastavení Jenkins projektů. Tato nastavení jsou přístupná přes položku `Nastavit` z bočního panelu stránky projektu, která je zobrazena na obrázku 4.6.

Mezi důležité položky nastavení řadím

- Správa zdrojového kódu (Git)
- Spouštění buildů (Manuální nebo automatické při uložení kódu)
- Build akce
- Post-build akce

Dále lze nastavovat název či popis projektu, uchovávání historie buildů a řadu dalších vlastností, které lze rozšiřovat pomocí pluginů.



Obrázek 4.6: Stránka Jenkins projektu

Git a Gitlab

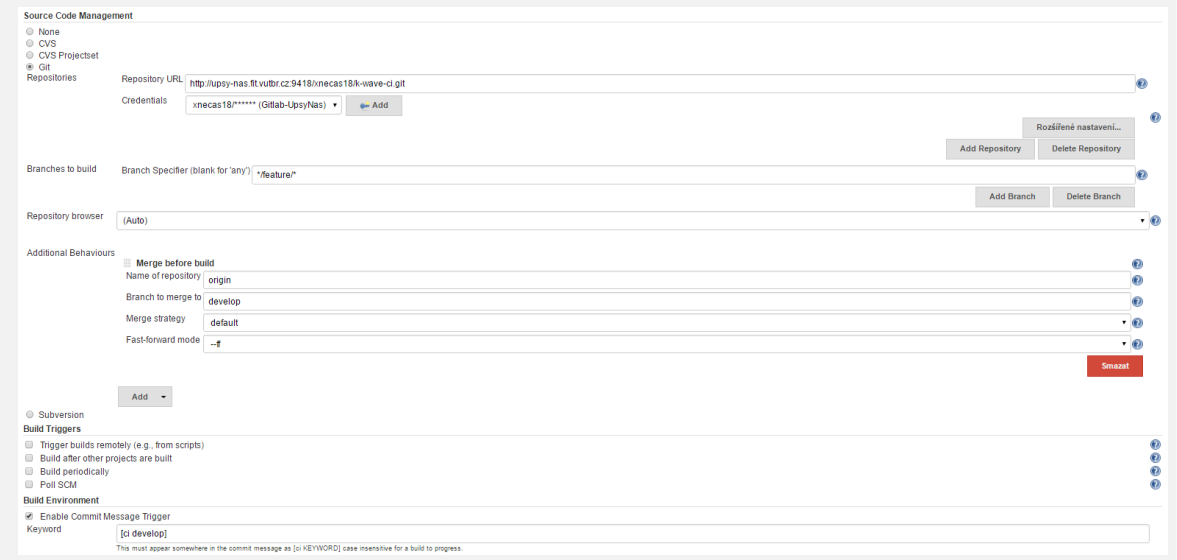
Požadavky na práci s Gitem jsou popsány v kapitole 3.5, potřebná nastavení na straně Gitlabu pak v kapitole 4.2.2. Práce s Gitem je odlišná pro feature a release větve.

Konfigurace Jenkins serveru je již obtížnější. Celá se provádí v nastavení konkrétního Jenkins projektu a skládá se z těchto kroků:

- V části `Source Code Managment` zaškrtneme možnost Git a nastavíme URL adresu repozitáře s potřebnými oprávněními viz 4.7. Udáme jaká větev repozitáře nás zajímá (feature nebo release). Toto nastavení umí pracovat se zástupnými znaky. Využijeme zde konvence pro pojmenovávání větví, kterou používá gitflow (podobá se cestám adresářů). Například upřesněním větve ve tvaru `*/feature/*` zajistíme, že nás zajímá libovolná feature větev.

- Pro feature větve potřebujeme zajistit, že před samotným buildem dojde ke sloučení s develop větví. Tohoto docílíme přidáním rozšířeného nastavení (`Additional Behaviours`) s názvem `Merge before build`. Vytvoří se tím nová sekce nastavení, která je také k vidění na obr. 4.7. Doplníme cílový repozitář a větev se kterou se bude merge provádět. Plugin umožňuje slučovat data z různých repozitářů a větví, což zpřístupňuje možnost realizovat rozličné scénáře. Automatické testy v release větvích sloučení kódu nevyužívají.

- Zajištění automatického spuštění se běžně provádí v části `Build Triggers`, kterou ale necháme nevyplněnou. Lze v ní nastavit pouze postupy jako například periodické spouštění buildu, opakované dotazování do git repozitáře apod. V projektu ale potřebujeme automatické spouštění, které je vyvoláno git repozitářem, respektive Gitlabem. K tomu jsem využil `Gitlab Hook Plugin`. Ten využije Gitlab Web hook, který je vyvolán vždy, když dojde uložení do origin repozitáře. Hook je zaslán na adresu ve tvaru: `http://jenkins-server/gitlab/notify_commit`. Jak jde vidět není v adrese vůbec uveden konkrétní Jenkins projekt. Ten se určí až dle obsahu hooku, který obsahuje sadu informací, mezi které patří o jakou událost se jedná (commit), jaké Git větve se událost týká, kdo ji provedl a spousta dalších. Ukázka zprávy je v příloze



Obrázek 4.7: Nastavení správy souborů na Jenkins

C. Jenkins projde tento obsah a spustí ty projekty, které zajímá odpovídající větev. Pokud tedy na Gitlabu dojde k uložení dat do konkrétní větve, vždy se na Jenkins spustí všechny buildy, které se o větev zajímají. Pokud potřebujete konkrétnější identifikaci buildů, je potřeba použít jiný postup. V projektu jsem využil dalšího pluginu s názvem `Commit Message Trigger Plugin`. Ten zajistí to, že když zpráva commitu nebude obsahovat specifikovaný text, nedojde provedení build akcí a projekt bude označen jako *NOT BUILD, NOT FAILED* nebo *SUCCESS*. Nastavení se provádí v sekci `Build Environment` a jde vidě také na obrázku 4.7.

- U testování feature větve chceme mít k dispozici i možnost automatického uložení sloučeného kódu do develop větve v origin repozitáři. Toto nastavení se provádí v sekci `Post-build actions` pomocí nástroje `Git Publisher`, který je součástí Git pluginu. V nástroji je opět třeba nastavit název repozitáře a větve, kam se bude kód ukládat. Je možnost přiložit i vlastní poznámku. V našem případě ale stačí zaškrtnou políčko `Merge Results`, protože používáme sloučení kódu před buildem. Nastavení lze vidět na obrázku 4.7, více informací o nastavení post-build akcí je v kapitole 4.3.2.

Další plugin pro spolupráci Jenkins - Gitlab je `gitlab-plugin`. Ten umožňuje spouštění přímo konkrétních Jenkins projektů. Při jejich větším počtu, ale může docházet ke značným prodlevám spuštění buildu (desítky minut).

Pro práci s Gitem je potřeba nainstalovat `Git plugin` a nastavit cesty ke spouštěcím souborům. U vzdálených uzlů se dá nastavení specifikovat pro každý uzel zvlášť. U Anselmu byla situace trošku komplikovanější. Výchozí verze Gitu, která je na superpočítači nainstalovaná nepodporuje autentizační metody, které používá Jenkins. Bylo nutné dohodnout instalaci novější verze na server. Nová verze je ale k dispozici formou modulu, který se musí zavést. To je v klasickém scénáři Jenkins buildu problém, protože modul pro Git musí být zaveden pouze pokud se použije Anselm uzel a musí být zaveden ještě před tím, než se začne stahovat kód. Zavedení tedy nemůže být součástí skriptu pro build, ale musí být provedeno dříve. V projektu jsem vyzkoušel více řešení, ale nakonec jsem se rozhodl pro použití pluginu `pre-scm-buildstep`. Ten přidá možnost provedení kroků ještě před

Obrázek 4.8: Jenkins spuštění bash skriptu

stažením kódu. Nastavuje se v části `Build Environment`.

Konfigurace build akcí

Tato část je jádrem práce s Jenkins. Existuje celá řada pluginů, které umožňují správcům provést velmi rychlé nastavení běžných úloh. Jak jsem ale uváděl v kapitole 3.4 tento projekt je poměrně atypický, proto jsem tyto nástroje nemohl použít a většinu funkcí jsem musel obstarat pomocí shell skriptů, které popíši blíže v kapitole 4.4. Z pohledu nastavení Jenkins serveru byla tato část tedy velmi jednoduchá. Obsahovala pouze jeden krok a to spuštění potřebného bash skriptu. Nastavení lze vidět na obr. 4.8. Shell skripty řídily většinu části build flow popsané v kapitole 3.3. Blíže je popíši v kapitole 4.4.

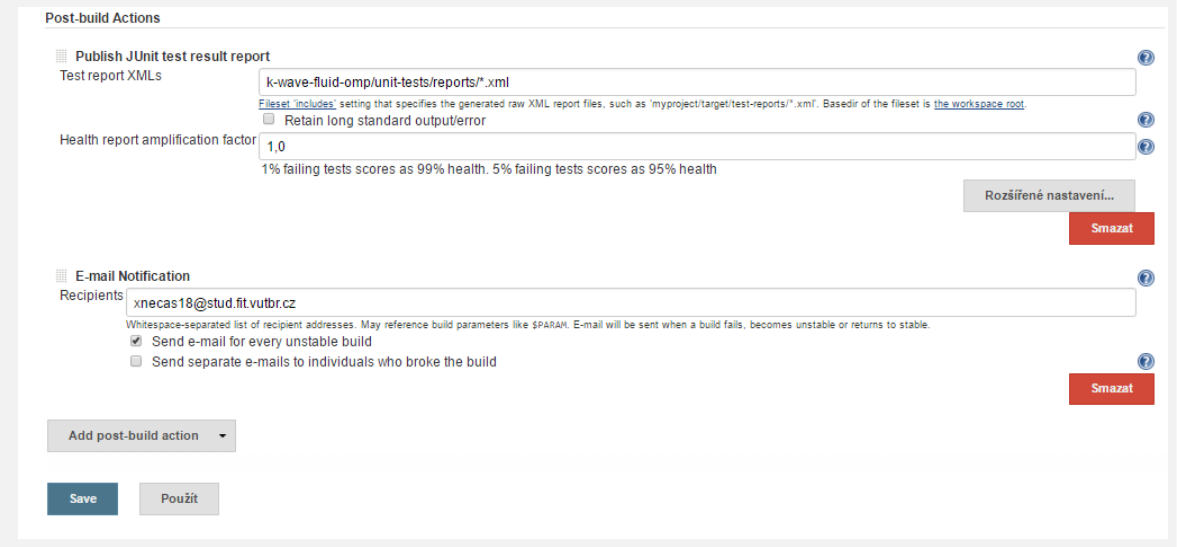
Konfigurace post build akcí

V této sekci se nastavují kroky, které se provedou až po proběhnutí buildu samotného. Typicky se zde provádí vytváření různých reportů, záloha artefaktu či automatický deployment. V této práci jsem vytvářel reporty pro unit testy a poté vlastní reporty pro regresní, případně výkonnostní testy. Některé z použitých technik vyžadovali zálohování artefaktů.

Při použití vhodné knihovny pro unit testování je vytvoření reportů velmi jednoduché a lze vidět na obrázku 4.9. Využijeme pluginu `JUnit`. V něm nastavíme adresář s xml výstupy testů a tzv. `health faktor`. Ten udává jak moc bude vyhodnocení výsledků testů přísné. Výchozí nastavení s hodnotou 1.0 znamená, že pokud selže 10% testů, bude celková úspěšnost testů 90%. Nejpřísnější hodnota 10.0 říká, že již při selhání 10% testů bude brán celý test jako neúspěch.

Nejdůležitější aspektem je ale vytvoření výstupního XML souboru v odpovídajícím formátu. Knihovna `GoogleTest` tento formát dodržuje. Většina jiných xUnit testovacích frameworků ale používá vlastní formát. Pro něj je nutné provést transformaci XML do odpovídajícího formátu. Tento krok lze naprogramovat vlastnoručně, případně použít plugin `xUnit plugin`, který umí výstupy vybraných knihoven (např. `CppUnit`) provést automaticky. Výše popsané knihovny automaticky stáhnou výstupní XML ze slave uzlů a vypíší příslušný report.

Zajímavější částí je tvorba vlastních reportů, které jsem chtěl zobrazovat na stránce projektu. Existuje více způsobů jak těchto reportů dosáhnout. Já zvolil řešení s využitím pluginu `Summary Display`. Vytvoření reportů se poté skládá z těchto kroků:



Obrázek 4.9: Report unit testů

1. Na slave uzlu se vytvoří výstupní XML soubory v požadovaném formátu. Tento soubor jsem vytvářel pomocí vlastní Python knihovny, kterou popíši v kapitole 4.4.1.

2. Po úspěšném dokončení testů se musí XML soubory dopravit na master uzel. K tomu lze použít archivaci artefaktů. Já si vystačil se základní možností, která je v Jenkins přístupná. Pro komplikovanější scénáře lze ale využít pluginu `ArtifactDeployer`. Pomocí artifaktů jsem distribuoval i další textové výstupy z regresních testů, potřebné pro manuální vyhodnocení.

3. Na master uzlu se ze souborů vytvoří potřebný report.

V nastavení Jenkins projektu je potřeba určit reportovací soubory a lze nastavit, zda se mají reporty zobrazit i na stránce projektu, ne jen výsledků buildu, jak lze vidět na obr. 4.10. Na obrázku lze vidět chybová hláška, která říká, že daný adresář není na serveru k nalezení. To je na začátku tvoření projektu u podobných konfigurací typické. Výstupní adresář se na serveru vytvoří až po prvním spuštění buildu.

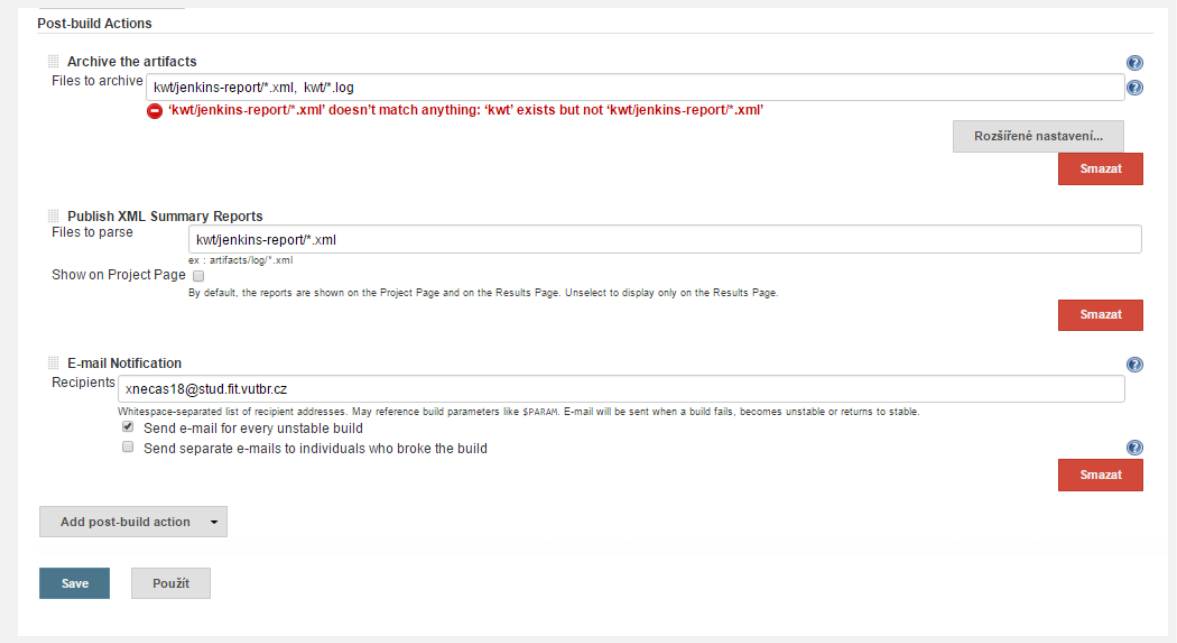
Multikonfigurační projekty

Tento typ projektů bude v dalších fázích pro projekt k-wave velmi důležitý. Pro začátek by mohla mít konfigurační matice dvě dimenze

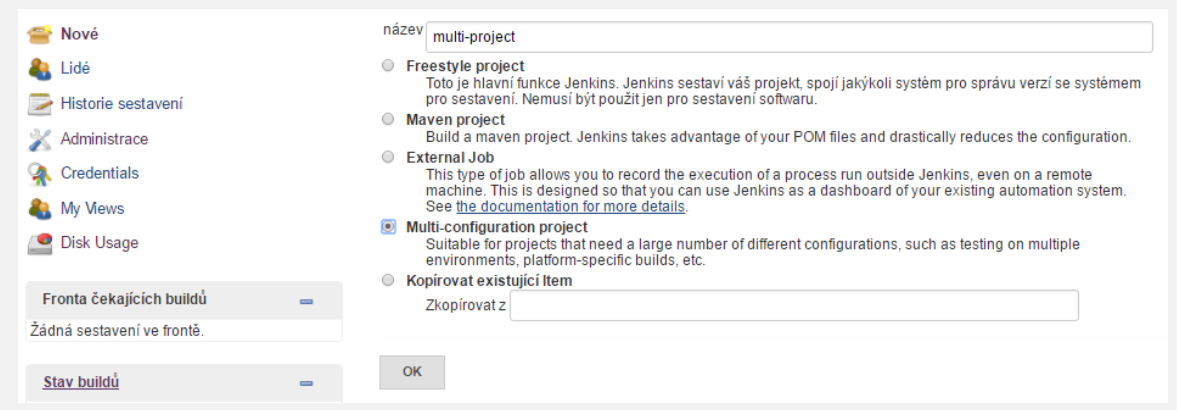
- **Slave uzel** - Základní scénář bude počítat s tím, že testy poběží na superpočítačích Anselm a Solomon. Do budoucna mohou přibýt další.

- **Překladač** - V současné době se testuje překladač `gcc` a `icpc` od firmy Intel. Každý vyžaduje trošku jiné nastavení a jiné verze pomocných knihoven. Ohled na různé volby překladačů byl brán i při volbě frameworku pro unit testy. `GoogleTest` funguje bez problémů s oběma překladači.

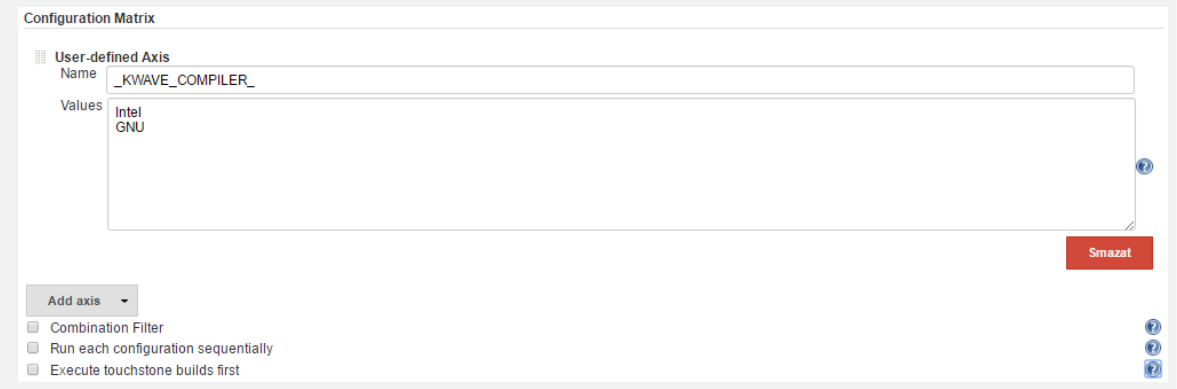
Časem mohou přibýt další dimenze matice jako například verze knihoven, optimalizace překladu, atd.



Obrázek 4.10: Vlastní reporty v Post build sekci



Obrázek 4.11: Nový multikonfigurační projekt



Obrázek 4.12: Nastavení multikonfiguračního projektu

Toto nastavení vyžaduje vytvoření zvláštního typu projektu, tzv. `Multi-configuration project`. Tato volba se vybírá v prvním kroku tvorby nového projektu a lze vidět na obrázku 4.11. V nastavení projektu, poté přibude nová sekce `Configuration Matrix`. Ta umožňuje provádět tyto akce:

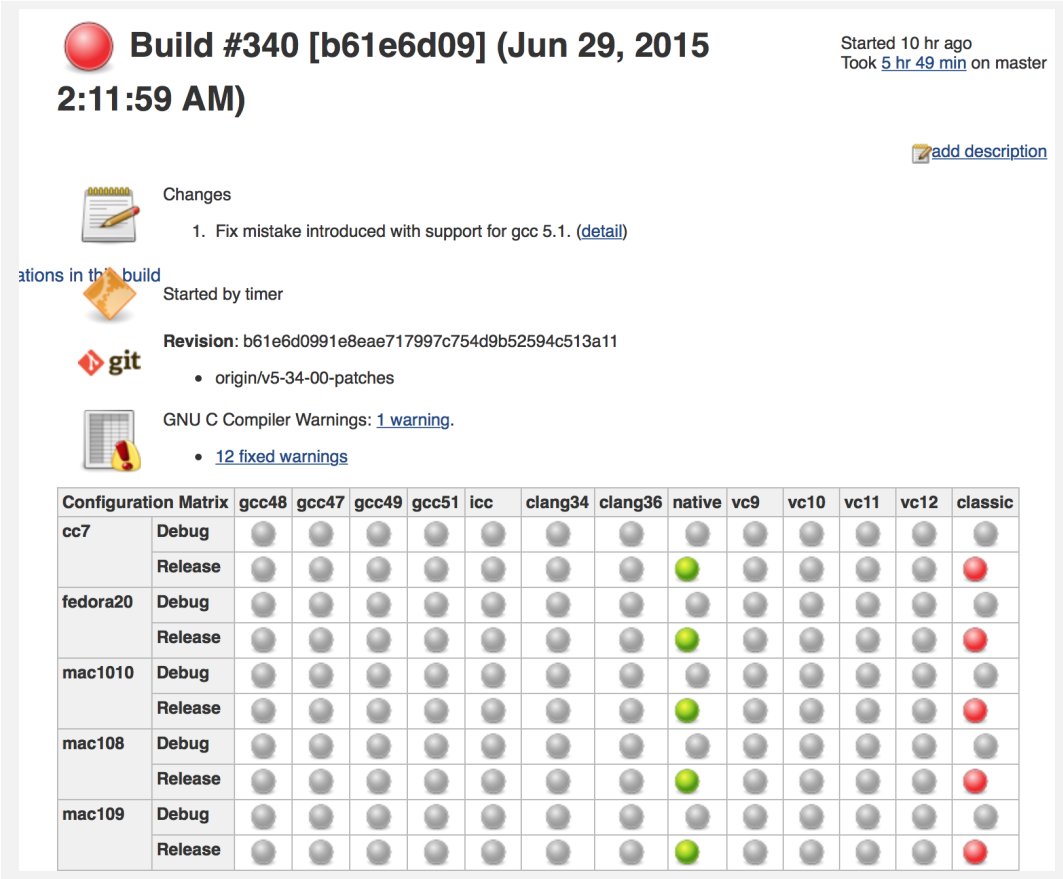
- **Přidání dimenze matice (Add axis)** - Umožní přidat další proměnnou, která ovlivní build a nastavit její povolené hodnoty. V základní instalaci jsou k dispozici možnosti `JDK` (verze Javy), `Label expression (Combination Filter)` (změna labelu, který určuje např. jaká skupina slave uzlů se použije), `Slaves` (přímé určení slave uzlu) a `User-defined axis` (uživatelsky definovaná proměnná).
- **Nastavení kombinačních filtrů** - Ve výchozím stavu tento typ projektu automaticky spustí všechny možné kombinace hodnot jednotlivých dimenzí konfigurační matice. Ne všechny kombinace ale musejí dávat smysl. V tomto nastavení lze pomocí filtrů nastavit, které kombinace nechceme použít.
- **Sekvenční běh konfigurací (Run each configuration sequentially)** - Pokud je tato volba zaškrtnutá bude Jenkins spouštět konfigurace sekvenčně za sebou. Ve výchozím nastavení je provádí paralelně vždy pokud je to možné. Tato volba se hodí například když jednotlivé kombinace přistupují ke sdílenému zdroji atd.
- **Upřednostnění konfigurací (Execute touchstone builds first)** - Umožňuje pomocí filtrů nastavit sadu konfigurací, které se mají spustit jako první. Pokud jejich výsledek splní nastavenou podmínku, dojde ke spuštění zbylých konfigurací.

Při testování možností jsem přidával dimenze pro Label expression (určovali slave uzel) a User-defined axis, pomocí které jsem si nastavoval hodnoty proměnných shellu. Na základě těchto proměnných se zvolil vhodný překladač a celková konfigurace překladu. Ostatní možnosti nastavení jsem nepoužíval. Nastavení proměnných pro shell lze vidět na obrázku 4.12.

Multikonfigurační projekt poskytuje i upravenou verzi reportů výsledků buildů. Ukázka reportů větší konfigurační matice je na obrázku 4.13 [1].

Multikonfigurační projekt jsem v této první fázi kontinuální integrace pro k-wave nezařadil. Především z důvodů časové náročnosti řešení. To by vyžadovalo získání přístupů na

[1] Převzato z webové stránky `https://root.cern.ch/how/how-use-root-jenkins`.



Obrázek 4.13: Ukázka reportu multikonfiguračního projektu

více superpočítačů a sjednocení nastavení překladu projektu pro více různých překladačů. Vytvořil jsem ale testovací projekty, které tuto možnost ověřovali a popsal způsob použití. Pomocné skripty (více v kapitole 4.4) již obsahují konstrukce pro práci s tímto typem projektů. Tím jsem poskytl základy pro jejich zařazení v dalších fázích.

Kooperace Jenkins projektů

U jednoduchých buildů si vystačíte s jedním Jenkins projektem. Pro složitější scénáře už je vhodné rozdělit práci do více projektů. Toto řešení sebou nese lepší přehlednost a škálovatelnost řešení. Musíme ale začít řešit kooperaci projektů, která se často popisuje jako *workflow více Jenkins projektů*.

Pro přehlednost připomenu, že se jedná již o třetí typ workflow popisovaný v této práci. První byla *Git workflow*, která popisovala práci s Gitem a jeho větvemi. Druhá pak *Jenkins workflow*, která popisovala běh konkrétního Jenkins projektu. Workflow více Jenkins projektů popisuje běh a spolupráci Jenkins projektů, z nichž každý implementuje Jenkins workflow, a které se vzájemně ovlivňují. Pokud například jeden projekt selže už nemá smysl spouštět další. Projekty si berou data z Git repozitáře a jsou automaticky spouštěny na základě commitů do konkrétních větví (feature nebo release), což je určeno na základě Git workflow. Jak jde tedy vidět jednotlivé workflow se navzájem ovlivňují a při návrhu řešení je třeba brát ohled na každou z nich. Pokud budu dále v této kapitole mluvit o workflow budu mít na mysli právě workflow nad více Jenkins projekty.

Obrázek 4.14: Jenkins pipeline view

Na Jenkins serveru lze implementovat různé workflow, od jednoduchých po složité. Pro jejich využití je většinou potřeba nainstalovat potřebný plugin. Já jsem si zvolil řešení pomocí `Build Pipeline Pluginu`. Jde o jednodušší typ workflow, která umožňuje vytvořit sekvenci zřetězených Jenkins projektů. Při spuštění této pipeline se v řadě za sebou spouští nejdříve první projekt, po něm druhý, atd. až do konce pipeline, jak je znázorněno na obrázku 4.14.

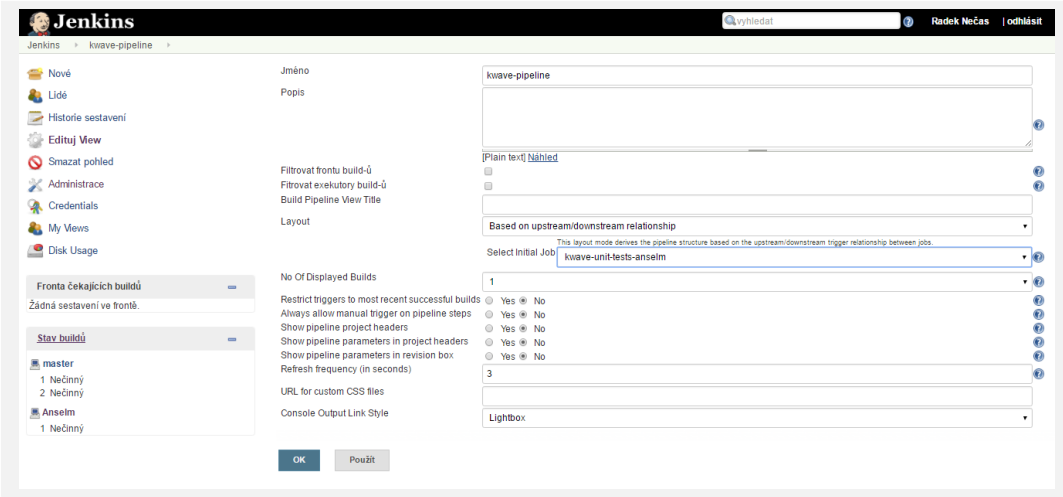
Vytvoření pipeline vyžaduje tyto kroky:

- **Vytvoření Pipeline View** - Jde o zvláštní typ pohledu, který se zobrazuje na výchozím dashboardu Jenkins webu. Po nainstalování pluginu přibude do seznamu pro vytvoření nových pohledů položka s názvem `Build Pipeline View`. Po jejím výběru je k dispozici nastavení, které lze vidět na obrázku 4.15. Na této stránce stačí zadat název pipeline a zvolit výchozí projekt. Ostatní položky stačí ponechat ve výchozím nastavení. V tomto pohledu poté bude zobrazena přehledná grafická prezentace pipeline s možností přechodu na detailní informace konkrétního jobu. Výsledný pohled lze vidět na obrázku 4.14.

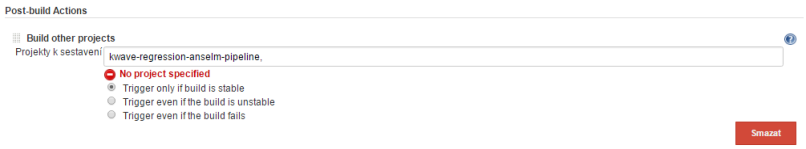
- **Zvolení výchozího projektu** - Výběr se provádí na stránce s nastavením Pipeline View (viz předchozí bod) pod položkou `Select Initial Job`. Jde o výchozí projekt, který se bude v pipeline spouštět jako první.

- **Propojení projektů v pipeline** - Tato konfigurace se musí provést v nastavení Jenkins projektu v sekci `Post-build Actions`. V ní se zaškrtne položka `Build other projects` a zadá se název projektu, který bude v pipeline následovat za projektem, který právě konfigurujete. Takto je třeba provázat všechny projekty v pipeline až na poslední, za kterým už žádný není. Nastavení lze vidět na obrázku 4.16. V nastavení lze zvolit, za jakých podmínek se má další projekt spustit (pokud aktuální projekt projde/selže/vždy). Lze zvolit i více projektů.

V této fázi zavedení kontinuální integrace jsem vytvořil pouze jednoduchou pipeline sestavenou z unit testů a regresních testů. Ta se bude spouštět vždy když dojde ke commitu do release větve origin repozitáře. Do budoucna můžou do pipeline přibývat další položky, popřípadě lze zvolit komplikovanější workflow.



Obrázek 4.15: Nastavení Jenkins pipeline view



Obrázek 4.16: Propojení projektů v Jenkins pipeline

4.3.3 Řešení potíží s Jenkins serverem

V této části velmi stručně popíši nástroje a postupy, které lze použít při řešení potíží s Jenkins serverem (master uzlem) a distribuovaným buildem na slave uzlech, popřípadě pro monitorování stavu systému. Veškeré nástroje a postupy lze provádět z webového rozhraní Jenkins serveru.

Potíže na master uzlu

Pro řešení potíží na master uzlu lze použít níže popsané nástroje, které jsou k dispozici z nastavení Jenkins serveru.

- **System Log** - Tento nástroje umožňuje zobrazit všechny systémové logy Jenkins serveru.
- **Disk Usage** - Tento nástroje je k dispozici jako plugin stejného názvu. Po jeho instalaci přibude v nastavení Jenkins serveru položka `Disk Usage`. Ta zobrazuje využití disku v rámci jednotlivých projektů.
- **Monitor of Jenkins master** - Tato položka poskytuje velmi rozsáhlé reporty ohledně vytížení master uzlu. Data jsou k prezentovaná formou tabulek i grafů. K dispozici jsou statistiky týkající se používání vláken, množství dotazů na server, vytížení procesoru, paměti a mnoho dalších. K aktivaci položky je potřeba nainstalovat plugin `Monitoring`.

Tyto nástroje jsem použil při řešení potíží s výkonem Jenkins serveru.

Potíže na slave uzlu

Problémy na slave uzlech se řeší obtížněji, jelikož často nemáme k dispozici potřebné nástroje a oprávnění pro jejich monitorování. Možnosti různých výpisů jsou taky poměrně omezené. Při řešení potíží v tomto projektu jsem používal tyto nástroje

- **Nastavení uzlů** - V nastavení Jenkins serveru je k dispozici položka `Manage Nodes`, která slouží pro správu slave uzlů. Při výběru konkrétního uzlu jsou poté k dispozici nástroje pro správu uzlu jako například zobrazení systémových informací o prostředí slave uzlu, Groovy konzole pro spouštění jednoduchých příkazů na slave uzlu, log a monitorování slave uzlu, které je součástí pluginu `Monitoring` o kterém jsem mluvil v předchozí části.
- **Výstup konzole buildu** - Při přepnutí na stránku s konkrétním buildem je v bočním menu k dispozici položka `Console Output`. V této konzoli lze živě vidět výpisy všech akcí prováděných na slave uzlu. Záznamy zůstávají zachované i pro zpětné prohlížení.
- **Pracovní prostor projektu** - Umožňuje nahlédnutí do adresáře projektu. Na slave uzlu si můžete vytvořit vlastní mechanizmus logování do souboru, který zde lze prohlížet.

Existuje řada dalších nástrojů, které usnadňují řešení potíží s během na slave uzlech. Pro tento projekt ale byli většinou nevhodné. Nejčastěji jsem využíval možnosti zobrazení slave konzole a pracovního prostoru, do kterého jsem si zapisoval vlastní logy.

4.4 Pomocné skripty

Jádrem každého Jenkins automatického buildu je samotná definice kroků, které se mají provést na testujícím prostředí. Požadované způsoby testování projektu, jeho běhového prostředí a použité platformy znemožňovalo použití většiny nástrojů pro automatizaci řízení běhu a dalších činností, které se zde provádí. Většinu úkonů jsem tedy realizoval pomocí skriptů. Jelikož budou testy běžet na linuxovém serveru zvolil jsem jako základní jazyk `Bash skript`. Pro dílčí kroky jsem ale použil i jiné jazyky jako `Python` či `Awk`.

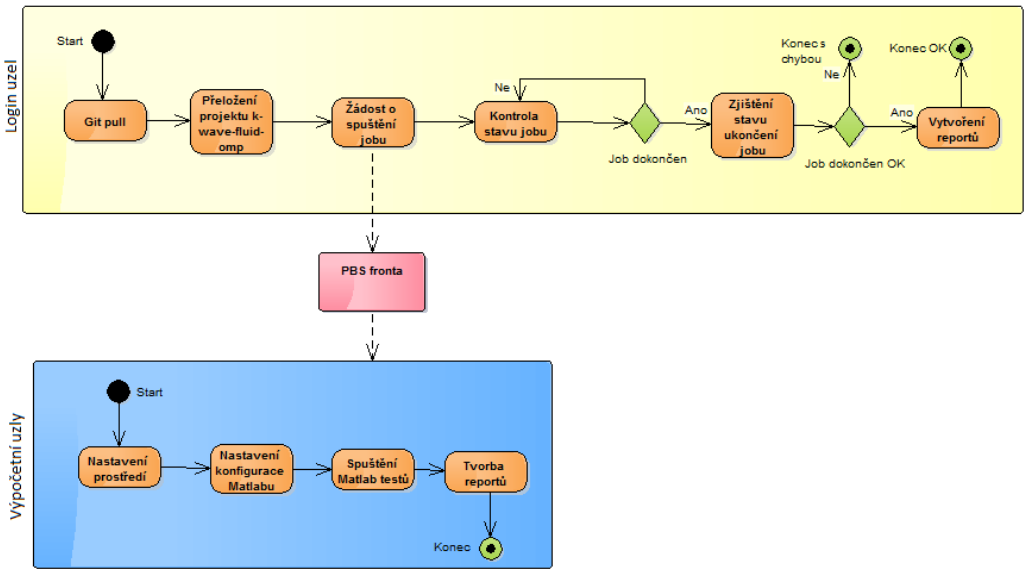
4.4.1 Regresní testy

Vlivem úpravy kódu, aktualizací knihoven či běhového prostředí, může docházet k různým nepřesnostem ve výpočtu. Při takovýchto chybách je nutné provést novou konfiguraci, použít jinou verzi knihovny, případně upravit výpočetní kód. K nepřesnostem dochází i při přechodu na stanice s rozdílnými architekturami. Takovéto regrese v kódu má odhalit tento typ automatických testů. Testy porovnávají výsledky `omp` verze oproti `Matlab` verzi a kontrolují, zda odchylka nepřekročí určitou mez.

Návrh a implementace testů

Tento typ testů vyžaduje běh na výpočetních uzlech superpočítače. Použil jsem návrh *sekvenčního testu s jedním Jenkins projektem*, který je popsaný v kapitole 3.4.1. Celý běh je ovládaný bash scriptem a je znázorněný na schématu 4.17.

Jednotlivé kroky jsou ze schématu poměrně jasné. Zopakuji zde pouze práci s joby na Anselmu, kterou jsem již popsal v kapitole 3.4. Žádostí o spuštění jobu vznikne požadavek, který se zařadí do některé ze systémových front. Jelikož systém nemá sofistikovanější



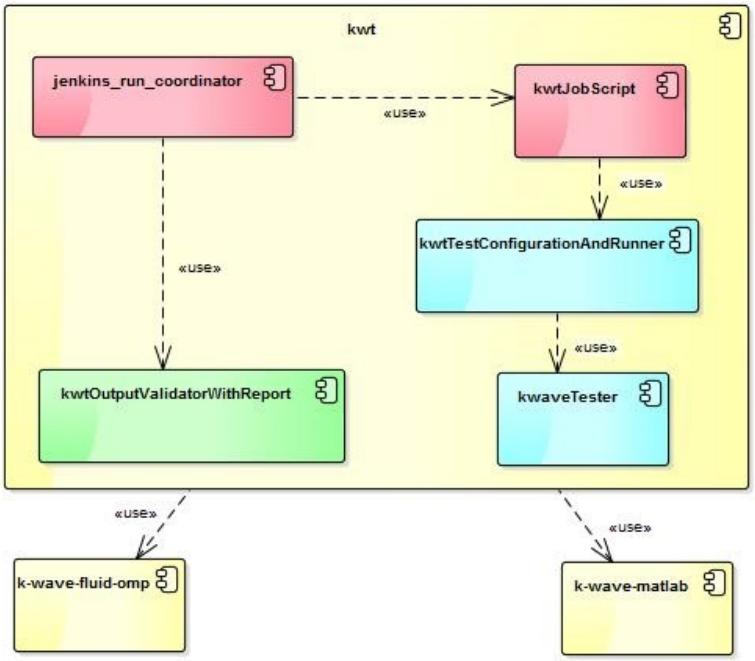
Obrázek 4.17: Diagram aktivit regresního testu

možnosti notifikace o ukončení jobu, je nutné kontrolovat stav opakovaným dotazováním. Jakmile se zjistí, že je job ukončen, je potřeba zjistit stav jeho ukončení, tzn. zda doběhl v pořádku nebo nastala chyba. Nakonec se můžou vyhodnotit výsledky, ve formě reportů, které vznikly na login uzlu a systém je automaticky uložil do svazku, odkud byl kód testu spuštěn.

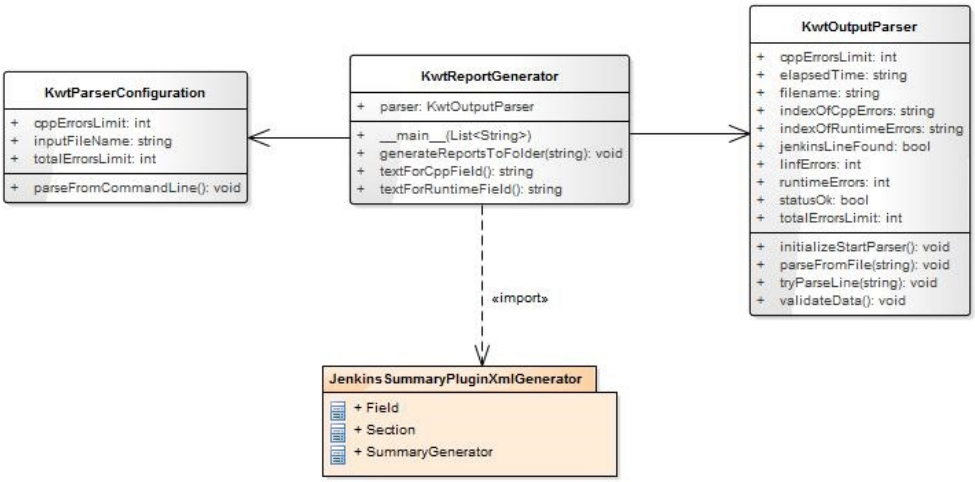
Jednotlivé komponenty řešení jsou k vidění na obrázku 4.18. Kromě verzí projektu pro OMP a Matlab je na schématu k vidění komponenta `kwt`, která se skládá z dalších komponent, které provádí testy. Barevné odlišení znázorňuje použité technologie.

Uvedu zde stručný popis komponent:

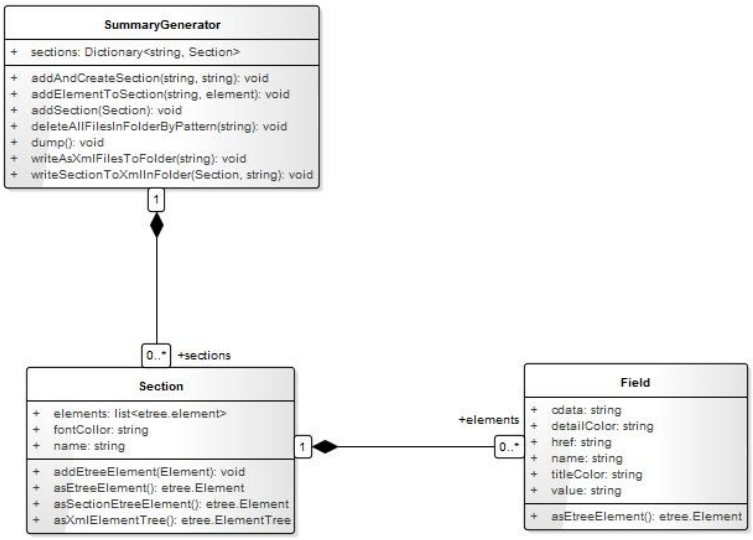
- **jenkins_run_coordinator** - Bash skript, který řídí celý běh testů, provádí některé kroky a sdružuje většinu nastavení testu. Tato komponenta využívá a řídí všechny ostatní.
- **kwtJobScript** - Bash skript s konfiguracemi a příkazy pro běh na výpočetních uzlech. V podstatě jen spustí matlab skript, ve kterém je definovaný samotný test.
- **kwtTestConfigurationAndRunner** - Jde o Matlab skript, který nakonfiguruje testovací prostředí a spustí skript s testy. Převzal jsem jej ze stávajícího řešení projektu a pouze upravil pro potřeby automatického testování. Konfigurace je poměrně rozsáhlá, proto její popis nezařazuji do této práce.
- **kwaveTester** - Matlab skript, který provádí samotné testování dle konfigurace, kterou nastaví `kwtTestConfigurationAndRunner`. Skript byl opět převzatý. Pouze jsem upravil tvorbu výstupních dat, pro vhodnější automatickou analýzu výstupu.
- **kwtOutputValidatorWithReport** - Tato komponenta se stará o analýzu výstupů `kwaveTesteru`. Na jejím základě, a s využitím předané konfigurace, poté vytvoří výstupní XML reporty ve formátu vhodném pro `Summary Display` plugin Jenkins serveru. Ten na základě těchto souborů vytvoří vlastní reporty na stránce projektu.



Obrázek 4.18: Komponenty kódu regresních testů



Obrázek 4.19: Diagram tříd komponenty kwtOutputValidatorWithReport



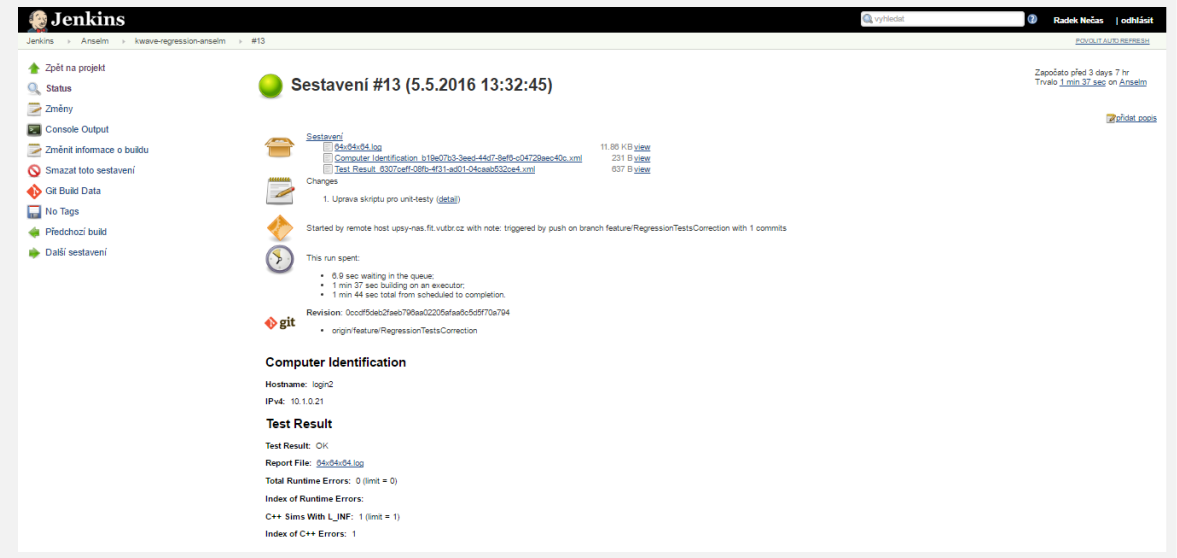
Obrázek 4.20: Diagram tříd balíčku JenkinsSummaryPluginXmlGenerator

`kwtOutputValidatorWithReport` je komponenta psaná v jazyce Python 3 s využitím objektově orientovaného programování. Jednotlivé třídy lze vidět na obrázku 4.19. Třída `KwtReportGenerator` řídí celý proces. Pomocí třídy `KwtOutputParser` analyzuje výstup testů a porovnává jej dle konfigurace zadané pomocí `KwtParserConfiguration`. Pokud jsou odchylky v zadaných mezích, jsou použity třídy z balíčku `JenkinsSummaryPluginXmlGenerator` pro generování XML reportů, na základě kterých `Summary Plugin` vytvoří report na stránce Jenkins projektu. Třídy tohoto balíčku jsou k vidění na diagramu 4.20.

Třída `SummaryGenerator` řídí celý běh vytváření výstupních XML souborů. K tomu využívá zbylé třídy. Název tříd odpovídá komponentám pro `Summary Plugin`. Jejich kompletní seznam lze najít v dokumentaci k pluginu. Odkaz na stránku s dokumentací je uveden v příloze D. Současná implementace využívá pouze dvě z nich a to `Section` a `Field`. Report se zobrazí na stránce s výstupem buildu a lze vidět ve spodní části obrázku 4.21.

V jednoduchosti lze říct, že report se skládá ze dvou sekcí (Section) - identifikace počítače a výsledku testu. Ty obsahují další komponenty. V reportu používám pouze jednu z nich, tzv. pole (Field), které obsahuje dvojici název a hodnota. Pro každou sekci musí vzniknout zvláštní XML soubor, který obsahuje jednotlivé pole sekce. Implementace není nijak složitá a pro její pochopení doporučuji nahlédnout do zdrojových kódů.

Testování dokončení jobu na výpočetních uzlech je součástí komponenty `jenkins_run_coordinator`. Skript v cyklu testuje stav běhu jobu pomocí příkazu `qstat`, který umožňuje získat informace o jobech přihlášeného uživatele. Výstup tohoto příkazu se vyhodnotí pomocí skriptu psaném v jazyce `awk`. Výstup příkazu a možné stavy jsou popsány v příloze F. Nevýhodou tohoto řešení je to, že výstup příkazu `qstat` není plně standardizovaný a na některých stanicích se může lišit. Bude potom nutné upravit toto testování a zajistit mechanismus přepínání jeho verzí.



Obrázek 4.21: Report nástroje Summary Plugin

Konfigurace testů

Z předchozí kapitoly lze vyčíst, že jsem se snažil použít co nejvíce komponent, které už tým projektu kwave používá. Šlo mi především o to, abych týmu co nejvíce zjednodušil přechod na použití systému kontinuální integrace. Používání a nastavení existujících nástrojů jim v prvním kroku může tento přechod zpříjemnit.

Tento přístup značně ovlivnil návrh a implementaci regresních testů, společně s potřebami nastavení prostředí na superpočítačích. Důsledkem je komplikované nastavení testů, které je potřeba provést ve více zdrojových souborech a které vůbec nepoužívá možnost konfigurace Jenkins projektu. Tuto vlastnost beru jako největší negativum tohoto řešení a v dalších fázích nasazení kontinuální integrace jej doporučuji upravit. Po konzultaci s vedoucím projektu jsem ale usoudili, že výhody použití známých nástrojů v této fázi převažují výše popsané nevýhody.

Nastavení testů se dělí do těchto částí:

- Největší část konfigrace jsem sloučil do komponenty `jenkins_run_coordinator`. V ní lze nastavit cesty k jednotlivým skriptům a nástrojům, nastavit parametry porovnání výsledků (akceptované odchylky), nastavit výstupní soubory a nakonfigurovat prostředí pro login uzel.

- Nastavení `PBS` jobu je potřeba provést v komponentě `kwtJobScript`. Jde o nastavení jako např. určení fronty do které se job zařadí, identifikace projektu pro potřeby účtování, maximální doba, po kterou může job běžet, počet výpočetních uzlů, atd. Toto nastavení probíhá pomocí speciálně formátovaných komentářů na začátku skriptu. Tento přístup je běžně používán při práci s `PBS`. Dále komponenta obsahuje načtení modulů pro výpočetní uzly.

- Nastavení regresních testů je prováděno uvnitř `kwtTestConfigurationAndRunner`. Je opravdu velmi komplexní a obsahuje celé konfigurační matice s desítkami různých příznaků. Možnosti konfigurace jsou dobře popsány v komentářích zdrojového souboru.

Kvůli velkému rozsahu je nebudu v této práci popisovat. Tento způsob nastavení je týmů dobře známý, protože jej využívá ve svém stávajícím řešení.

Kompletní export nastavení regresních testů z posledního bodu nelze kvůli velkému rozsahu vyřadit do konfigurace Jenkins projektu. Možným řešením je určit mnohem užší množinu možných konfigurací. Její hodnoty reprezentovat výčtem, který už lze konfigurovat v Jenkins projektu například prostřednictvím konfigurační matice.

4.4.2 Unit testy

Při realizaci tohoto úkolu mi šlo především o navržení a vytvoření metod pro automatické testování. Také jsem vytvořil systém řízení běhu testů na superpočítači a základní strukturu kódu pro testování s využitím frameworku `GoogleTest`. Cílem nebylo poskytnout pokrytí velké části kódu, ale vytvořit základní prostředí do kterého se budou jednotlivé testy komponovat a ukázat cestu po které se mohou vývojáři při implementaci testů vydat.

Popis běhu testu

Návrh a implementace těchto testů je velmi podobná regresním testům. Jenkins build se opět vykonává vzdáleně na slave uzlu, kterým je superpočítač. Zde se testy naplánují pro provedení na výpočetních uzlech. Na login uzlu dochází k opakovanému dotazování na stav jobu. Jakmile je job na výpočetních uzlech hotový, dozví se to skript na login uzlu, který vytvoří potřebné reporty, které jsou zaslány zpět na Jenkins master uzel. Jediným významným rozdílem je to, že místo regresních testů se na výpočetních uzlech vykonávají unit testy. Taky není potřeba starat se o vytvoření reportů pro Jenkins server. Potřebné XML soubory na slave uzlu vytvoří knihovna pro unit testy, v případě tohoto projektu `GoogleTest`. Formát těchto souborů přímo odpovídá požadavkům Jenkins serveru není tedy potřeba řešit XML transformace jako u jiných frameworků.

Implementace

Veškeré soubory potřebné pro testy jsou k dispozici v adresáři `unit-tests` projektu `k-wave-fluid-omp`. Zde se kromě frameworku samotného nachází i adresář `src`, který obsahuje veškerý zdrojový kód testů. Jde o zdrojové kódy v jazyce `C++`. Pro každou testovanou komponentu kódu by měl kvůli přehlednosti vzniknout nový zdrojový soubor.

V následujícím popisu očekávám, že je čtenář seznámen s pojmy popsanými v kapitole 2.4.1. Každý zdrojový soubor může obsahovat třídu, která definuje `Test Fixture` a slouží pro sdílení dat mezi více `Test Case`. Také umožňuje volání inicializačních a úklidových metod `SetUp()` a `TearDown()`, které jsou volány před a po každém `Test Case`. Tato třída musí dědit od `::testing::Test` a členy třídy musí být `public` či `protected`. Kromě SetUp() a TearDown() metod lze provést inicializaci a úklid nastavení testů i v konstruktoru a destruktoru třídy. Tato možnost je preferovaným způsobem. Jednotlivé `Test Case` se vytváří pomocí makra ve tvaru znázorněném na následujícím kódu.

```
TEST_F(test_case_name, test_name) {
... test body ...
}
```

Zde musí platit, že `test_case_name` musí odpovídat názvu třídy, která definuje příslušnou `Test Fixture`. V těle tohoto makra jsou poté k dispozici všechny členské objekty a metody `Test Fixture`. Makro se současně postará o zaregistrování všech `Test Case`.

Není tedy nutné provádět tuto registraci manuálně jako u ostatních frameworků. Při registrování dochází ke sloučení `Test Case` pro stejnou `Test Fixture`, což zpřehledňuje výstup testů. Názvy parametrů musí být platné identifikátory jazyka `C++` bez podtržítek. Výsledný název testu se sestaví z obou parametrů. Přístup práce s `Test Fixture` je poměrně neobvyklý, protože jednotlivé `Test Case` nejsou implementovány jako metody třídy, ale jako makra volaná mimo třídu a k vazbě s třídou dochází přes první parametr makra.

Pokud není potřeba využít možností `Test Fixture`, lze definovat testy pomocí následujícího makra, které se od předchozího liší pouze v názvu:

```
TEST(test_case_name, test_name) {
... test body ...
}
```

Spuštění všech registrovaných testů se provádí zavoláním makra `RUN_ALL_TESTS()`, které vrací hodnotu 0 v případě úspěchu, jinak vrací 1. Tuto hodnotu lze přímo použít jako návratovou hodnotu funkce `main`. Volání tohoto makra, dokonce ani vytváření `main` funkce není potřeba provádět manuálně. Framework obsahuje knihovnu `gtest_main.a`, kterou stačí přilinkovat k projektu. Knihovna poskytuje velmi jednoduchou implementaci `main` funkce s následujícím kódem

```
GTEST_API_ int main(int argc, char **argv) {
    printf("Running main() from gtest_main.cc\n");
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Kompletní ukázku velmi jednoduchého testu lze vidět v příloze E.

Pro zajištění spolupráce s Jenkins bylo nutné nastavit výstup do XML souboru. Toto nastavení zajistíme spuštěním testu s přepínačem `-gtest_output="xml:${TEST_REPORT_DIR}"`, kde `${TEST_REPORT_DIR}"` udává cestu do adresáře, kam budou reporty umístěny.

Knihovna `GoogleTest` je distribuována současně s unit testy a je nutné zajistit její přeložení na cílovém počítači. Překlad a spouštění testů jsem řídil s využitím nástroje `GNU Make`. Knihovna šla bez problému přeložit pomocí `gcc` i `icpc` překladače.

Kapitola 5

Zhodnocení výsledků práce

Hlavní přínos práce vidím ve vytvoření základní kostry řešení, na které může tým dále rozvíjet kontinuální integraci a automatizaci opakovaných úkonů. Zprovoznil a nakonfiguroval jsem serverové prostředí včetně nástrojů a jejich kooperace. Z poměrně rozsáhlé oblasti různých nástrojů, postupů a metodik jsem vybral vhodné kandidáty a pomocí nich vytyčil cestu kterou se může tým vydat. Podařilo se mi navrhnout a implementovat řešení atypických problémů projektu, které může být dále používáno a můžou na něm být vystavěny rozsáhlejší konstrukce. Vytvořil jsem základní sadu regresních, unit testů a Jenkins projektů, která může být použita pro testování projektu a ukazuje, jak by se mělo s vytvořeným řešením pracovat.

Dále je ukázáno jak lze automaticky spouštět různé testy na různých stanicích s různými konfiguracemi, což je pro projekt velmi důležité, protože to bývá velmi častým zdrojem chyb. Navíc díky velkému množství různých kombinací jde o časově náročnou práci, která se musí opakovaně provádět nejlépe při každé úpravě kódu nebo změně běhového prostředí či konfigurace. To by mělo vývojářům ušetřit spoustu času, který mohou věnovat další práci na projektu. Mělo by tedy dojít ke zvýšení efektivity práce.

Další výhodu vidím ve zkvalitnění způsobu správy zdrojových kódů. Ta umožní lepší sdílení kódu většími týmy, možnost automatického spouštění testů a dalších úkonů, snížení vzniku kolizí při slučování kódu a zkvalitní procesu vydávání nových verzí. Navíc lze beze strachu o porušení kódu projekt zpřístupnit dalším vývojářům ve které není kladena taková důvěra. Mohou se na něm tedy snadněji podílet například studenti fakulty. Tato vlastnost by se opět měla projevit na zvýšení efektivity práce.

Návrh řešení předpokládá inkrementální zavádění kontinuální integrace. Je implementována pouze první fáze, při které je brán velký ohled na již používané nástroje a postupy týmu, aby se dal proces snadněji integrovat do jejich každodenní práce. Do budoucna by měla práce přinést projektu k-Wave především již zmiňované zvýšení efektivity, rychlejší proces vývoje nových funkcí a oprav chyb. Zjednoduší se rozšiřování projektu pro běh na nových stanicích s různými architekturami, knihovnami, překladači a jejich verzemi. Při dostatečném pokrytí projektu automatickými testy se budou lépe řešit refaktorizace a optimalizace kódu, což je pro projekt tohoto typu velmi důležité. Je možné zavést nástroje pro měření kvality kódu a zajistit propojení se systémy pro správu řízení projektu. Díky tomu by se dosáhlo mnohem větší přehlednosti o stavu projektu, průběhu jeho vývoje a celkové kvalitě kódu. Dosáhlo by se efektivnějšího a přehlednějšího řízení týmu. Obecně se projektu zpřístupní možnosti provázání s řadu externích systémů, které přináší zvýšení kvality práce týmu.

Práce má řadu přínosů i mimo projekt k-Wave. Z nich bych chtěl zmínit především to, že

ukazuje, jak lze i pro poměrně nestandardní projekty a běhová prostředí řešit problematiku kontinuální integrace, a že je tato metodika pro podobné projekty řešitelná. Navíc kompletně s využitím open source technologií, běžících na NAS serveru v Docker kontejnerech. Jde tedy o technologicky zajímavé a finančně nenákladné řešení. Také práce poskytuje přehled v dané oblasti a příslušných technologiích, což může být využito pro studijní účely.

Závěr

V této práci jsem čtenáře seznámil s technikami používanými v agilních metodykách vývoje softwaru. Zaměřil jsem se na praktické nástroje používané ve vývoji, především na kontinuální integraci, způsoby správy verzí zdrojových kódů, automatického testování a kontrolu kvality kódu. Uvedli jsme si nejpoužívanější nástroje na dnešních trhu se stručným porovnání jejich schopností. U vybraných nástrojů jsme se blíže seznámili s principy jejich funkce a způsoby využití. Byla předvedena workflow použití těchto nástrojů, kterou lze aplikovat na většinu mainstreamových projektů, popřípadě ji použít jako odrazový můstek pro navržení vlastní.

Seznámili jsme se s projektem k-Wave a způsobem správy zdrojů a úloh na superpočítačích. Představil jsem problémy se kterými jsem se musel vypořádat při zavedení kontinuální integrace. Obdobné problémy mohou vznikat u všech projektů této kategorie, tedy takových, které jsou zaměřeny na výkon a běží na superpočítačích. Uvedl jsem možná řešení těchto problémů a vybral vhodné kandidáty pro tento projekt.

Práce popisuje zřízení kontinuální integrace pro reálný projekt v celém rozsahu její implementace. Od návrhu řešení, výběru vhodných nástrojů, zřízení a správy běhového serveru, konfiguraci jednotlivých nástrojů a zajištění jejich spolupráce, po programovou část. Ta obsahuje vytvoření frameworku pro běh testů na superpočítači a ukázku prakticky použitelných regresních a unit testů.

Návrh řešení je tvořen s ohledem na praktické zapojení do reálného projektu k-Wave, který spravuje více jak desetičlenný tým. Proto byla integrace postupů a nástrojů rozdělena do více fází. Programová část implementace se snaží maximálně využít existující nástroje a postupy týmu. Což by mělo také zpříjemnit její praktické zavedení. Práce pokrývá první fázi integrace. Poskytuje základní kostru řešení a sadu ukázkových testů a konfigurací, které můžou být prakticky použity. Byl kladen důraz na dobrou dokumentaci řešení formou této práce a wiki stránek projektu. Ta by měla usnadnit použití řešení a realizaci dalších fází.

Hlavní přínosy práce jsou pospány v kapitole 5. Práce se dá rozšířit v mnoha směrech. Postupně by mělo docházet k lepšímu pokrytí kódy unit testy, především v oblasti výpočetních kernelů. Počítá se zavedením výkonnostních testů, které budou součástí build pipeline pro release větev. Do projektu se zapojí více slave uzlů (superpočítačů) a využijí se multikonfigurační projekty, které budou spouštět testy na různých superpočítačích s různými kombinacemi konfigurací. Dále lze přemýšlet o zavedení metodyk pro kontinuální distribuci výsledného softwaru.

Vlastní skupinu rozšíření tvoří propojení Jenkins serveru s externími nástroji. Pro lepší monitorování kvality kódu lze zajistit integraci s nástrojem `SonarQube`, jehož použití jsem testoval v rámci této práce. Kvalitnějšího vedení projektu a monitorování stavu práce lze dosáhnout spojením se sofistikovanějšími nástroji řízení projektů (např. `Jira`). Lepší komunikaci členů týmu a rychlejší notifikace na chyby jednotlivých buildů lze realizovat propojením s nástroji jako `Slack` či `HipChat`.

V neposlední řadě lze upravovat a optimalizovat stávající řešení. Důležitým krokem je sjednocení konfigurací, které by vyřešilo roztříštěnost současného řešení. Bylo by vhodné zkvalitnit workflow Jenkins buildu běžícího na superpočítači. K tomu ale bude zapotřebí zajistit podporu na straně správců superpočítače, například zprovozněním API pro správu jobů na výpočetních uzlech. V budoucnu by bylo vhodné zprovoznit služby na běžném serveru místo Docker kontejnerů, především s ohledem na bezpečnost, výkonost a paměťovou náročnost řešení.

Literatura

[1] Apache Subversion Community Guide (aka "HACKING"). https://subversion.apache.org/docs/community-guide/.

[2] Get started with it4innovations. https://docs.it4i.cz/get-started-with-it4innovations.

[3] van Baarsen, J.: *GitLab Cookbook*. Birmingham, UK: Packt Publishing, první vydání, 2014, 171 s., iSBN 978-1-78398-684-2.

[4] Berg, A. M.: *Jenkins Continuous Integration Cookbook*. Birmingham, UK: Packt Publishing, první vydání, 2012, 343 s., iSBN 978-1-849517-40-9.

[5] Chromatic, J. S.: *The Art of Agile Development*. USA: O'Reilly Media, první vydání, 2007, 431 s., iSBN 978-0-596-52767-9.

[6] Millett, J. B. M. B. S.: *Pro Agile .NET Development with Scrum*. New York, USA: Apress, první vydání, 2011, 381 s., iSBN 978-1-4302-3534-7.

[7] Osherove, R.: *The Art of Unit Testing*. Shelter Island, NY, USA: Manning Publications Co., druhé vydání, 2014, 396 s., iSBN 9781617290893.

[8] O'Sullivan, B.: Mercurial: The Definitive Guide. http://hgbook.red-bean.com/read/, 2009.

[9] Smart, J. F.: *Jenkins: The Definitive Guide*. USA: O'Reilly Media, první vydání, 2011, 405 s., iSBN 978-1-449-30535-2.

[10] Straub, S. C. B.: *Pro Git*. USA: Apress, druhé vydání, 2014, 574 s., iSBN 978-1484200773.

[11] team, A.: Git tutorial. https://www.atlassian.com/git/tutorials.

[12] Wikipedia: Network-attached storage — Wikipedia, The Free Encyclopedia. 2016, [Online; accessed 18-April-2016].
URL `https://en.wikipedia.org/w/index.php?title=Network-attached_storage&oldid=714553602`

Přílohy

Seznam příloh

Příloha A

Nastavení Docker kontejnerů

Tato příloha obsahuje popis nastavení docker kontejnerů.

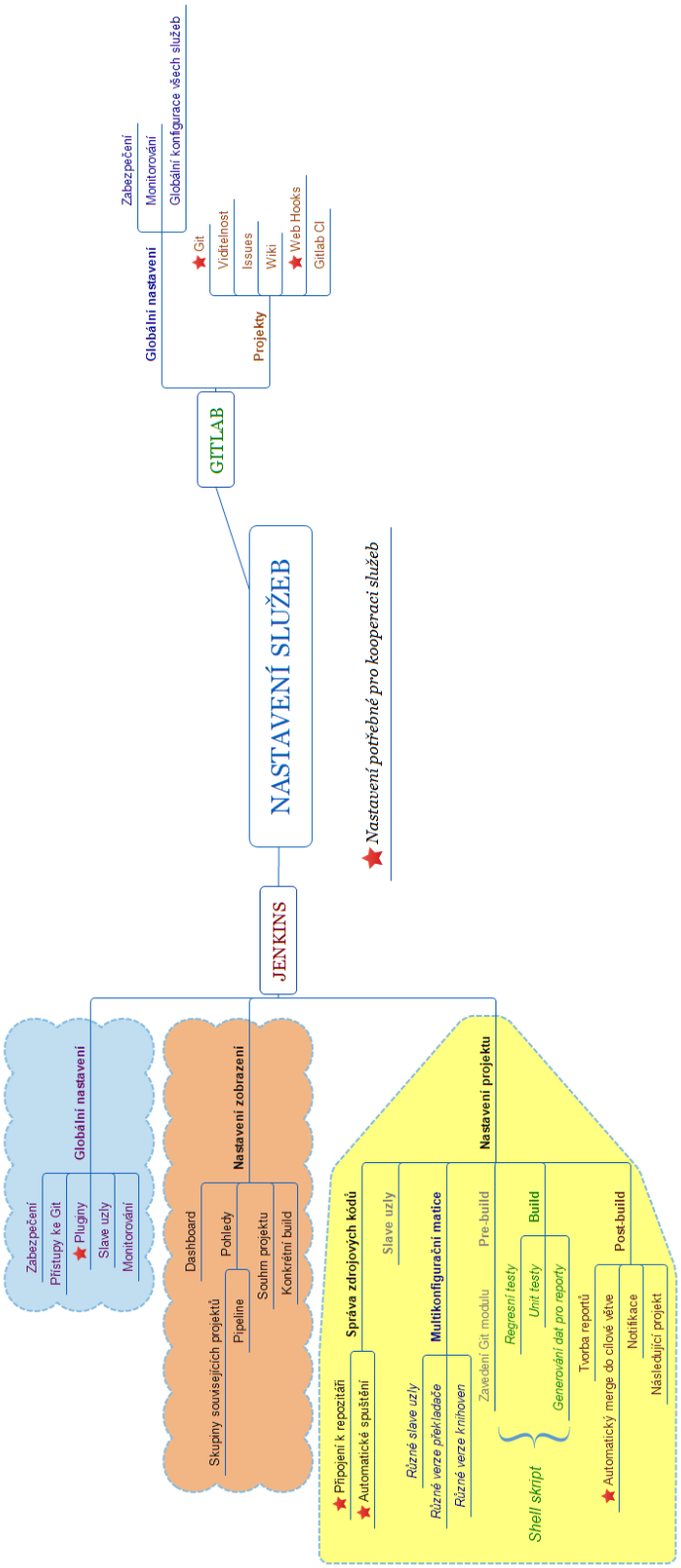
Služba na UPSY NAS	Docker image	Porty • port stanice • port kontejneru	Svazek • adresář stanice • adresář kontejneru	Odkazy • docker kontejner – název služby	Proměnné prostředí (Uvádím pouze názvy proměnných. Hodnoty nastavené dle konfigurací služeb a účtů)
Redis	redis:latest		• /docker/redis • /var/lib/redis		
MySQL	mysql:latest		• /docker/mysql • /var/lib/mysql		• MYSQL_ROOT_PASSWORD
Gitlab	sameersbn/gitlab:latest	• 10022 • 22 • 9418 • 80	• /docker/gitlab • /home/git/data	• Redis – redisio • MySQL - mysql	• GITLAB_SSH_PORT • GITLAB_PORT • GITLAB_HOST • SMTP_PASS • SMTP_USER • GITLAB_SECRET_DB_KEY • DB_NAME • DB_PASS • DB_USER
Jenkins	jenkins:latest	• 5055 • 50000 • 9090 • 8080	• /docker/jenkins • /var/jenkins/home		

Obrázek A.1: Nastavení docker kontejnerů

Příloha B

Nastavení služeb

Tato příloha obsahuje myšlenkovou mapu, která znázorňuje nastavení provedená na jednotlivých službách.



Obrázek B.1: Nastavení služeb

Příloha C

Gitlab push events hook formát

Tato příloha ukazuje formát Gitlab web hook zprávy, která vzniká při každém push do repozitáře. Gitlab obsahuje řadu jiných web hooks, které zde nejsou ukázány. Ukázkový výpis byl zkrácen pro přehlednější zobrazení na stránkách této práce.


```
{
    "object_kind":"push",
    "before":"95790bf891e76fee5e1747ab589903a6a1f80f22",
    "after":"da1560886d4f094c3e6c9ef40349f7d38b5d27d7",
    "ref":"refs/heads/master",
    "checkout_sha":"da1560886d4f094c3e6c9ef40349f7d38b5d27d7",
    "user_id":4,
    "user_name":"John Smith",
    "user_email":"john@example.com",
    "user_avatar":"https://s.gravatar.com/avatar/d4c74594d840",
    "project_id":15,
    "project":{
        "name":"Diaspora",
        "description":"",
        "web_url":"http://example.com/mike/diaspora",
        "avatar_url":null,
        "git_ssh_url":"git@example.com:mike/diaspora.git",
        "git_http_url":"http://example.com/mike/diaspora.git",
        "namespace":"Mike",
        "visibility_level":0,
        "path_with_namespace":"mike/diaspora",
        "default_branch":"master",
        "homepage":"http://example.com/mike/diaspora",
        "url":"git@example.com:mike/diaspora.git",
        "ssh_url":"git@example.com:mike/diaspora.git",
        "http_url":"http://example.com/mike/diaspora.git"
    },
    "repository":{
        "name":"Diaspora",
        "url":"git@example.com:mike/diaspora.git",
        "description":"",
```

```
        "homepage":"http://example.com/mike/diaspora",
        "git_http_url":"http://example.com/mike/diaspora.git",
        "git_ssh_url":"git@example.com:mike/diaspora.git",
        "visibility_level":0
    },
    "commits":[
        {
            "id":"b6568db1bc1dcd7f8b4d5a946b0b91f9dacd7327",
            "message":"Update Catalan translation to e38cb41.",
            "timestamp":"2011-12-12T14:27:31+02:00",
            "url":"http://example.com/mike/diaspora/commit/b65687",
            "author":{
                "name":"Jordi Mallach",
                "email":"jordi@softcatala.org"
            },
            "added":[
                "CHANGELOG"
            ],
            "modified":[
                "app/controller/application.rb"
            ],
            "removed":[

            ]
        },
        {
            "id":"da1560886d4f094c3e6c9ef40349f7d38b5d27d7",
            "message":"fixed readme",
            "timestamp":"2012-01-03T23:36:29+02:00",
            "url":"http://example.com/mike/diaspora/commit/da1567",
            "author":{
                "name":"GitLab dev user",
                "email":"gitlabdev@dv6700.(none)"
            },
            "added":[
                "CHANGELOG"
            ],
            "modified":[
                "app/controller/application.rb"
            ],
            "removed":[

            ]
        }
    ],
    "total_commits_count":4
}
```

Příloha D

Použité Jenkins pluginy

V této příloze je uveden seznam některých zajímavých pluginů, které nejsou ve výchozím stavu nainstalované.

Název pluginu	**Zdroj**	**Popis**
Build Pipeline Plugin	https://wiki.jenkins-ci.org/display/JENKINS/Build+Pipeline+Plugin	Vytvoření pipeline z více Jenkins projektů
Commit Message Trigger Plugin	https://wiki.jenkins-ci.org/display/JENKINS/Commit+Message+Trigger+Plugin	Kroky buildu se provedou pouze pokud commit zpráva obsahuje specifikovaný test.
Cppcheck Plug-in	https://wiki.jenkins-ci.org/display/JENKINS/Cppcheck+Plugin	Možnost jednoduché statické analýzy kódu
Custom Tools Plugin	https://wiki.jenkins-ci.org/display/JENKINS/Custom+Tools+Plugin	Instalace vlastních nástrojů před spuštěním buildu.
disk-usage plugin	https://wiki.jenkins-ci.org/display/JENKINS/Disk+Usage+Plugin	Monituruje spotřebu místa buildů.
Doxygen Plug-in	https://wiki.jenkins-ci.org/display/JENKINS/Doxygen+Plugin	Automatické generování dokumentace.
Email Extension Plugin	https://wiki.jenkins-ci.org/display/JENKINS/Email-ext+plugin	Pokročilé možnosti nastavení notifikačních emailů.
Git plugin	https://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin	Možnost použít Git jako VCS.
Gitlab Hook Plugin	https://wiki.jenkins-ci.org/display/JENKINS/Gitlab+Hook+Plugin	Zajišťuje spolupráci s Gitlabem tím, že napodobuje chování Gitlab CI.

Green Balls	https://wiki.jenkins-ci.org/display/JENKINS/Green+Balls	Výchozí indikátory pro úspěšný běh jsou modré. Tento plugin je změní na intuitivnější zelené.
Mailer Plugin	https://wiki.jenkins-ci.org/display/JENKINS/Mailer	Další rozšíření notifikačních emailů.
Monitoring	https://wiki.jenkins-ci.org/display/JENKINS/Monitoring	Monitorování zátěže Jenkins serveru
pre-scm-buildstep	https://wiki.jenkins-ci.org/display/JENKINS/pre-scm-buildstep	Provedení kroků před spuštěním SCM
SSH Slaves plugin	https://wiki.jenkins-ci.org/display/JENKINS/SSH+Slaves+plugin	Správa slave uzlů přes ssh.
Analysis Collector Plugin	https://wiki.jenkins-ci.org/display/JENKINS/Analysis+Collector+Plugin	Různé statické analýzy kódu.Závislý na dalších pluginech Checkstyle, Dry, FindBugs, PMD, Task Scanner a Warnings.
Summary Display Plugin	https://wiki.jenkins-ci.org/display/JENKINS/Summary+Display+Plugin	Umožňuje vytvářet vlastní reporty na stránkách buildů. Reporty jsou popsány v XML souborech, které mohou vytvářet slave uzly.
xUnit plugin	https://wiki.jenkins-ci.org/display/JENKINS/xUnit+Plugin	Možnost získávání reportů z různých xUnit testovacích frameworků (patří mezi ně i CppUnit)

Příloha E

Ukázka unit testu

Tato příloha obsahuje ukázku velmi jednoduchého testu vytvořeného pomocí frameworku `GoogleTest`.

```
namespace {

    // The fixture class
    class FFTWComplexMatrixTest: public ::testing::Test {
    protected:
        FFTWComplexMatrixTest()
            : realOnes4x4(TDimensionSizes(5,11,7)),
            realOnes4x4_2(TDimensionSizes(5,11,7)),
            fftwComplex(TDimensionSizes(3,11,7))
                {
            for (size_t i=0; i < realOnes4x4.GetTotalElementCount();
                i++)
            {
                realOnes4x4[i] = 1.0f;
                realOnes4x4_2[i] = 1.0f;
            }
        }

        virtual ~FFTWComplexMatrixTest() {
            // You can do clean-up work that doesn't throw
            // exceptions here.
        }

        // If the constructor and destructor are not enough for
        // setting up
        // and cleaning up each test, you can define the following
        // methods:

        virtual void SetUp() {
            // Code here will be called immediately after the
            // constructor (right before each test).
```

```
    }

    virtual void TearDown() {
        // Code here will be called immediately after each
        // test (right
        // before the destructor).
    }

    // Objects declared here can be used by all tests in the
    // test case for Foo.

    TRealMatrix realOnes4x4;
    TRealMatrix realOnes4x4_2;
    TFFTWComplexMatrix fftwComplex;
};

TEST_F(FFTWComplexMatrixTest, TestCreateReal4x4) {
    EXPECT_EQ(realOnes4x4.GetTotalElementCount(), 16);
    for (size_t i=0; i < realOnes4x4.GetTotalElementCount();
        i++)
    {
        EXPECT_EQ(realOnes4x4[i], 1.0) << "Element " << i <<
          " has unsuported value: " << realOnes4x4[i];
    }
}
```

Příloha F

Ukázka výstupu nástroje qstat

Tento nástroj slouží k získání stavu jobu, který je umístěný do fronty nebo je zpracováván dávkovacím serverem.

Ukázka spuštění příkazu a jeho výstupu

```
$ qstat -x $jobid
Job id            Name             User             Time Use S Queue
----------------  ---------------- ---------------- -------- - -----
1270160.dm2       kW-bench         xnecas18         00:03:56 F qexp
```

Význam většiny sloupců je jasný z jejich názvů. `Time Use` udává dobu běhu na výpočetních uzlech, což je důležitý údaj, jelikož výpočetní čas těchto uzlů je na superpočítačích účtovaný. Parametr `Queue` udává frontu procesu. Anselm nabízí více front s různými prioritami. Pro tento projekt byl nejdůležitější položkou sloupce s názvem `S`, který popisuje aktuální stav procesu. Právě hodnota tohoto sloupce opakovaně kontrolována částí skriptu, který se staral o čekání na dokončení výpočtu na výpočetních uzlech. Na různých systémech se mohou poskytované stavy lišit. Na superpočítači Anselm byly k dispozici tyto hodnoty stavů procesu:

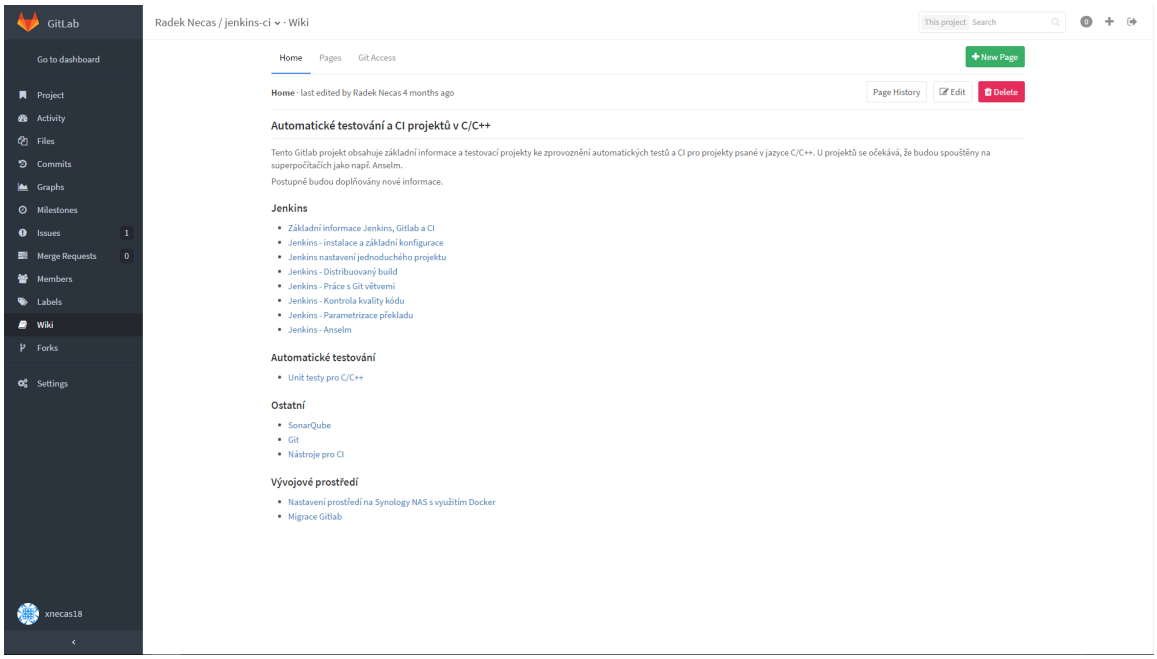
- **E - Exiting**: Proces dokončil běh (s chybou či bez) a systém provádí úklid po běhu jobu.
- **H - Held**: Job je pozastaven (jedním či více příkazů held).
- **Q - Queued**: Job čeká ve výpočetní nebo směrovací frontě na přiřazení prostředků. Není pozastaven.
- **R - Running**: Job je ve výpočetní frontě. Byly mu přiřazeny výpoečetní prostředky a bylo odstartováno jeho provádění.
- **S - Suspend**: Jobu byli přiděleny prostředky, běžel, ale byl pozastaven. Přidělené zdroje jobu zůstávají, ale nevyužívá je.
- **T - Transiting**: Job je směrován na výpočetní prostředky nebo přesouván na novou destinaci.
- **W - Waiting**: Job není pozastaven, ale atribut **Execution_Time** ještě nebyl dosažen.
- **F - Finished**: Job byl ukončen.

Skript kontroluje výstup příkazu, dokud stav nenabývá hodnoty E nebo F. Při spouštění skriptů typicky docházelo k této sekvenci stavů procesu: Q -> R -> R -> ... -> R -> E.

Příloha G

Ukázka wiki stránek

V této příloze je ukázána úvodní stránka wiki, kde jsou zpracovány informace ohledně nastavení prostředí pro tento projekt. Tato wiki je dalším zdrojem obsahujícím nastavení, správu a používání řešení, které je popsáno v této práci.



Obrázek G.1: Úvodní wiki stránka

Příloha H

Obsah přiloženého CD

V této příloze je popsán obsah přiloženého cd a také struktura zdrojového kódu, který je na CD také přiložen.

Níže je uveden obsah CD ve formě popisu jednotlivých adresářů.

- `src` - Zdrojové soubory projektu.
- `wiki` - Obsahuje zálohu wiki, která poskytuje další informace ohledně nastavení a používání řešení, které je popsané v této práci.
- `jenkins` - Obsahuje zálohu nastavení použitých Jenkins projektů.
- `doc` - Obsahuje zdrojové soubory pro tuto práci, která byla vytvářena v systému LaTeX.

Následuje popis adresářové struktury zdrojových kódů, umístěných ve složce `src`.

- `k-wave-fluid-omp` - kWave ve verzi OMP. Obsahuje podadresář `unit-tests`, ve kterém jsou umístěny všechny potřebné soubory pro unit testy, včetně použitých knihoven. Více o testech v kapitole 4.4.2.
- `k-wave-matlab` - kWave ve verzi pro Matlab. Tato verze se bere jako referenční pro regresní testy. Obsahuje řadu pomocných Matlab skriptů. Soubory v tomto adresáři jsem nijak neupravoval.
- `kwt` - Adresář se všemi soubory pro regresní testování. Více v kapitole 4.4.1.
- `Readme.md` - Obsahuje stručný popis projektu a zde popsané adresářové struktury.