Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Implementácia distribuovaného build systému

Bakalárska práca

2016
Peter Poláčik

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Implementácia distribuovaného build systému

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Michal Rjaško, PhD.

Bratislava, 2016
Peter Poláčik

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Peter Poláčik
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Implementácia distribuovaného build systému
Implementation of distributed build system

Cieľ: Navrhnúť a implementovať systém na distribuované kompilovanie programov na viacero serverov. Za účelom zrýchlenia vytvárania tzv. build programu, systém na základe súboru makefile rozdelí prácu pre kompilátory na viacerých severoch.

Vedúci: RNDr. Michal Rjaško, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Dátum zadania: 26.10.2015

Dátum schválenia: 27.10.2015 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie

Chcel by som sa poďakovať môjmu školiteľovi, RNDr. Michalovi Rjaškovi, PhD., za ochotu, pomoc, pripomienky a nápady, ktoré mi pomohli pri písaní tejto práce. Ďakujem aj Dáši Kotíkovej, Jurajovi Mašlejovi a Jakubovi Novákovi za psychickú podporu a všetkým ostatným, ktorí mi poskytli cenné rady.

Abstrakt

Bakalárska práca sa zaoberá implementáciou aplikácie, ktorá slúži na distribuovanie práce kompilátorov na viacero počítačov za účelom zrýchlenia procesu kompilácie projektu. Práca popisuje teoretickú prípravu, architektúru a návrh systému, použité nástroje a knižnice, a implementáciu.

Kľúčové slová: kompilácia, distribúcia úloh, Go, Linux

Abstract

The bachelor's thesis is concerned with implementation of application, which distributes compiler work among multiple computers in order to accelerate the project compilation process. The thesis describes theoretical preparations, architecture and design of the system, used tools and libraries, and the implementation.

Keywords: compilation, workload distribution, Go, Linux

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam listingov

Úvod

Kompilácia zdrojových súborov je každodenná súčasť práce na softvérovom projekte. Opakované zadávanie rovnakých príkazov môže pri nepozornosti spôsobenej rutinou spôsobiť zlyhanie. Preto je potrebné takéto procesy automatizovať. V spoločnostiach s prístupom k serverovej farme je možné tento výkon, pokiaľ ho nevyužívajú používatelia použiť na urýchlenie tohto procesu.

Existujúce riešenia ako GNU Make, Apache Ant alebo Bazel poskytujú automatizáciu procesu na počítači používateľa. Avšak pre spoločnosti s prístupom k serverovým farmám a s rozsiahlymi projektami je výhodné použiť nevyužitý výkon serveru na kompiláciu. Niektoré veľké firmy majú vlastné systémy ktoré toto umožňujú, ale neexistuje jednoduché open-source riešenie, ktoré by vznikajúce spoločnosti mohli využiť.

Preto sme sa rozhodli implementovať distribuovaný build systém, ktorý bude jednoduchý na použitie a výkonný aj pri rozsiahlych softvérových projektoch. Ako vhodný programovací jazyk sa javí Go, vďaka vstavanej podpore paralelizmu, a ako platformu sme zvolili operačný systém Linux/Unix, kvôli rozšírenosti na serveroch ako aj pracovných staniciach programátorov.

Cieľom tejto práce je poskytnúť teoretické základy za systémom a tiež oboznámiť čitateľa so štruktúrou a logikou aplikácie.

V kapitole 1 popíšeme požiadavky a návrh systému. V kapitole 2 vysvetlíme dôvody výberu použitých technológií a v kapitole 3 uvedieme teoretické základy návrhu.

Kapitola 4 tvorí jadro práce a popisujeme v nej implementáciu aplikácie. Vysvetlíme fungovanie a logiku systému, popíšeme jednotlivé moduly a služby, a uvedieme ukážky zaujímavých časti zdrojového kódu.

V kapitole 5 poskytneme dokumentáciu inštalácie, nastavenia a používania systému, spoločne s príkladmi popisov kompilovanych projektov. Na záver, v kapitole 6 porovnáme vytvorený systém s existujúcimi riešeniami.

Kapitola 1

Špecifikácia a návrh systému

V tejto kapitole popíšeme požiadavky na systém a navrhneme riešenie.

1.1 Úvod do problematiky

Majme projekt skladajúci sa z väčšieho množstva zdrojových súborov, ktoré treba skompilovať (alebo pre webové aplikácie minifikovať) predtým, ako z nich vytvoríme výslednú aplikáciu. Keďže postup kompilácie je presne daný a častokrát opakovaný, je veľmi výhodné ušetriť čas potrebný na spúšťanie všetkých príkazov za sebou použitím nejakého systému, ktorý to zautomatizuje. Okrem automatizácie chceme určite ušetriť aj čas pri takzvanej rekompilácii, aby sa po zmene niekoľkých súborov nemusela znova kompilovať celá aplikácia. Niekoľko existujúcich systémov spomíname v časti 1.3.

V prípade spoločnosti pracujúcej na projektoch obsahujúcich rádovo stovky až tisíce súborov je takýto systém nevyhnutný. Avšak aj pri použití automatizácie a použitia výsledkov predchádzajúcich behov môže rekompilácia, najmä pri väčších zmenách, trvať dlho, čo znižuje nielen produktivitu zamestnancov, ale aj zisky spoločnosti. Preto niektoré korporácie využívajú na kompiláciu (ktorá je pri veľkom počte nezávislých súborov inherentne vysoko paralelizovateľná) nielen pracovné stanice zamestnancov, ale aj ich serverové farmy.

Avšak narážame na problém. V čase písania tejto práce sú všetky takto distribuované riešenia uzavreté, neprístupné verejnosti, iba spoločnosti, ktorá ho vyvinula a drží si ho pod svojou strechou. Toto je veľká nevýhoda pre začínajúce start-up organizácie, ktoré, aj keď majú k dispozícii servery, ktoré by takto mohli využiť, nemôžu si dovoliť vývoj takéhoto riešenia pre vlastnú potrebu.

1.2 Formulácia problému

Sformulujme teda problém na základe predchádzajúcej časti a pridajme ďalšie požiadavky:

- Automatizácia kompilácie a linkovania zdrojových súborov do výslednej aplikácie.
- Využívanie predchádzajúcich výsledkov pri ďalších behoch.
- Distribúcia kompilácie na servery, pokiaľ sú dostupné.
- Sprístupnenie existujúcich behov a výsledkov celej organizácii.
- Rozšíriteľná podpora rôznych jazykov a postupov.
- Minimalizovanie potreby ručnej konfigurácie.
- Podpora VCS[1], najmä Git.[2]
- Vydané pod open-source licenciou.

Pozrime sa podrobnejšie na niektoré požiadavky.

Sprístupnenie existujúcich behov a výsledkov celej organizácii

Ak dvaja zamestnanci pracujú na projekte potrebujúcom kompiláciu tisícky súborov, nie je dôvod aby obaja, pri prvom pokuse skompilovať takýto projekt, museli čakať niekoľko hodín pred tým ako môžu ďalej pracovať. Preto plánujeme vytvoriť distribuované úložisko objektov vytvorených kompiláciou na serveroch, ktoré budú takto dostupné všetkým používateľom daného systému. Toto všetko za účelom čo najväčšieho ušetrenia výpočtového výkonu a času spotrebovaného kompiláciou.

Rozšíriteľná podpora rôznych jazykov a postupov

V prvej verzii tohto systému plánujeme implementovať podporu pre niekoľko najpoužívanejších jazykov. Avšak uvedomujeme si, že takýto systém by mal byť nezávislý na použitých nástrojoch a preto jeho súčasťou bude aj jednoduchý spôsob pridávania podpory pre iné platformy.

Minimalizovanie potreby ručnej konfigurácie

V prípade existujúcich riešení je častokrát až zarážajúce množstvo konfigurácie, ktorú treba vykonať pri použití takého systému v projekte. Náš systém by mal tieto záležitosti čo najviac zjednodušiť.

[1] Version Control System
[2] `https://git-scm.com`

Podpora VCS

Veľké množstvo projektov závisí na súboroch, knižniciach, ktoré sa nenachádzajú v ich repozitári, ale sú dostupné pod nejakým VCS. Náš systém by mal, pri vhodnej špecifikácii dostupnosti týchto závislostí v konfigurácii, byť schopný automaticky stiahnuť a použiť pri kompilácii takúto závislosť, bez potreby akejkoľvek činnosti od používateľa.

1.3 Existujúce systémy

V tejto časti popíšeme niekoľko open-source build systémov. Vo všetkých prípadoch ide ale iba o lokálnu kompiláciu.

GNU Make

GNU Make[3] je najstarší a stále pravdepodobne najpoužívanejší build systém. Využíva súbor nazvaný Makefile, ktorý obsahuje popis pravidiel na vytváranie súborov na základe ich prípony. Na zabránenie opakovanej kompilácii porovnáva časy posledného zápisu do súborov (zdrojového a výsledného), čo ale nezabraňuje opakovanej kompilácii napríklad v prípade obnovenia zdrojového súboru zo zálohy. Tento systém podporuje paralelnú kompiláciu, ale len v obmedzenej miere a na lokálnom počítači. Na zjednodušenie písania pravidiel v Makefile vzniklo niekoľko systémov, ktoré tvorbu Makefile automatizujú. Medzi najpoužívanejšie patria GNU Autotools a CMake.

Apache Ant

Apache Ant[4] je build systém, ktorý adresuje niekoľko problémov s prenositeľnosťou Makefile súborov medzi platformami, keďže tieto špecifikujú akcie ako postupnosť príkazov v shelli. Konfigurácia je vykonávaná v XML súboroch, ktoré definujú strom úloh (tasks) pre projekt. Každá z úloh je implementovaná ako trieda v jazyku Java, v ktorom je implementovaný aj celý Ant a teda je jednoduché pridať nové úlohy, ako aj preniesť existujúce úlohy na nové systémy. Tento systém využívajú najmä projekty písane v jazyku Java.

Bazel

Open-source verzia interného build systému spoločnosti Google, ktorý bol prvý krát zverejnený v roku 2014. Využíva podobné koncepty ako Apache Ant. Funkcionalita tohto systému bola popísaná v roku 2012 sériou článkov na Google Developers blogu [7] a slúži ako inšpirácia pri návrhu nášho systému. Podľa plánov vydania [1] sa ale Google

[3] `https://www.gnu.org/software/make/`
[4] `https://ant.apache.org/`

nechystá zverejniť ich implementáciu distribuovanej kompilácie, ktorá je popísaná vo vyššie spomínanej sérii článkov.

1.4 Návrh riešenia

V tejto časti popíšeme jednotlivé komponenty systému a ich funkcionalitu.

1.4.1 Ciele kompilácie

Kompilovaný projekt delíme na ciele kompilácie (anglicky targets), ktoré definujú jednotlivé základné stavebné prvky objektu. Každý cieľ má určený svoj typ, napríklad aplikácia písaná v jazyku C, alebo knižnica funkcií jazyka C++. Okrem typu môžeme cieľu určiť závislosť na splnení iných cieľov, alebo špecifikovať vstupné zdrojové súbory a súbory prostriedkov.

Výsledkom splnenia každého cieľa kompilácie je generovaný súbor. Tento súbor vzniká aplikáciou pravidiel (akcií) na zdrojové súbory cieľa alebo iných generovaných súborov. Jeden cieľ môže špecifikovať aj postupnosť niekoľkých pravidiel.

Z grafu závislostí medzi cieľmi kompilácie sa vytvorí bipartitný graf pravidiel a súborov, ktorý popisujeme nižšie.

Zdrojové súbory

Toto sú súbory, ktoré sa nachádzajú priamo v repozitári projektu, sú vytvorené ľuďmi (alebo nástrojmi) a nie generované týmto systémom. V grafe závislostí sú koncovými vrcholmi, keďže nemajú žiadne závislosti.

Generované súbory

Súbory vytvorené ako výstup z nejakého pravidla, častokrát slúžia zároveň aj ako vstupné súbory pre ďalšie akcie.

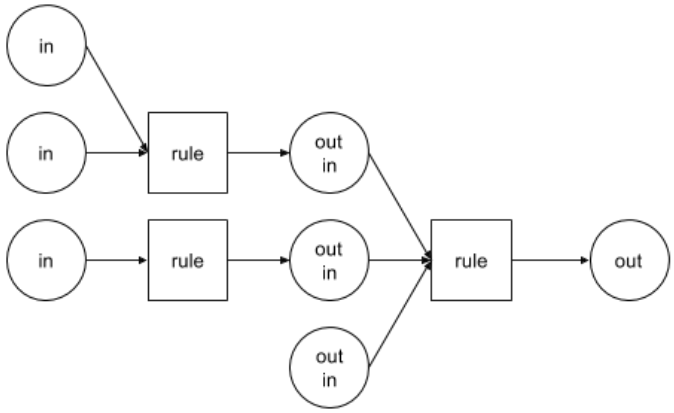
Pravidlá

Popisujú vzťah a postup vytvorenia výstupného súboru zo súborov vstupných. Každé pravidlo má práve jeden výstupný súbor a 0 alebo viac vstupných. Pravidlá zahŕňajú postupy ako kompilácia, linkovanie a podobne.

Graf závislostí

Pri vyžiadaní kompilácie cieľa vytvoríme orientovaný graf závislostí, ktorý vyžadujeme, aby bol acyklický. V prípade nájdenia cyklu systém ukončí beh s chybou o nesplniteľnosti požiadavky.

Vrcholy grafu tvoria súbory a pravidlá, kde hrana od súboru k pravidlu existuje práve vtedy, keď je súbor vstupom pravidla, a od pravidla k súboru, ak je súbor výstupom pravidla. Takto vytvorený graf bude bipartitný, kde disjunktnými množinami sú pravidlá a súbory. Na obrázku 1.1 sa nachádza príklad takéhoto grafu.



Obr. 1.1: Príklad grafu závislostí, okrúhle vrcholy predstavujú súbory, štvorcové vrcholy predstavujú pravidlá.

Takto zostrojený graf bude potom použitý slave komponentmi na vytvorenie výstupných súborov s použitím pravidiel a príkazov ktoré so sebou nesú.

1.4.2 Master

Aplikáciu bude predstavovať program, bežiaci na užívateľskom počítači. Tento program nazvime *master*.

Úlohou mastera bude v okamihu zavolania a špecifikovania cieľa kompilácie spracovať konfiguráciu všetkých cieľov daného projektu a vytvoriť strom závislostí. Následne master kontaktuje jednotlivé *slave* komponenty bežiace na rôznych strojoch (lokálny počítač, serverová farma a pod.) a rozdistribuuje ciele na tieto počítače čo možno najefektívnejšie.

Pokiaľ cieľ alebo podciele závisia na vonkajšom projekte a repozitár tohto projektu je dostupný, tento projekt je stiahnutý a použitý. V prípade nedostupnosti niektorej zo závislostí program ukončí beh s chybovou hláškou.

1.4.3 Slave

Slave je program starajúci sa o vykonávanie úloh zadaných masterom. Na fyzickom počítači, či už lokálnom, alebo na serveri bude bežať zvyčajne niekoľko slave programov, nakoľko jeden slave program bude navrhnutý na vykonávanie úloh pre práve jeden cieľ. Keď voľný slave obdrží úlohu, v prvom kroku sa pozrie do zdieľaného úložiska objektov,

ak je toto dostupné, či už žiadaný súbor neexistuje. V prípade jeho existencie slave pošle výsledný súbor priamo masterovi a nebude vykonávať kompiláciu. V prípade neexistencie výsledného súboru v žiadanej verzii bude cieľ preložený na postupnosť akcií a tieto budú vykonané.

1.4.4 Úložisko objektov

Úložisko objektov slúži ako cache na zabránenie opakovanej kompilácii rovnakých súborov. Výsledné súbory tu budú nahrané po dokončení kompilácie slave programom. Ukladanie bude prebiehať podľa názvu zdrojového súboru a jeho kontrolného súčtu, aby sa ľahko rozlíšili rôzne verzie súboru, napríklad lokálne modifikovaná od tzv. HEAD verzie[5].

V prípade nášho úložiska sme sa rozhodli použiť na kontrolné súčty hashovaciu funkciu SHA512, kvôli čo najväčšej minimalizácii kolízií.

1.4.5 Dostupnosť zdrojových súborov

Zdrojový kód je verzionovaný systémom Git[6] a zverejnený pod licenciou MIT v GitHub repozitári na adrese `https://github.com/imterra/forge`.

[5] najnovšia verzia zverejnená v hlavnom repozitári

[6] `https://git-scm.com`

Kapitola 2

Použité technológie

V tejto časti sa bližšie pozrieme na použité technológie, dôvody na ich využitie, ich výhody a nevýhody.

2.1 Go

Jazyk Go je kompilovaný, staticky typovaný programovaný jazyk pôvodne vyvinutý spoločnosťou Google. Jeho syntax vychádza zo syntaxe jazyka C. Jazyk poskytuje automatickú správu pamäte (garbage collection), limitované prvky objektovo orientovaného programovania a podporuje súbežnosť na báze komunikujúcich súbežných procesov.[3]

2.1.1 Objektovo orientované programovanie

Jazyk Go nepodporuje existenciu objektov v štýle OOP jazykov ako Java. Namiesto toho sa OOP v jazyku Go implementuje pomocou štruktúr a metód s tzv. typovým prijímačom (method receiver). Príklad uvádzame v listingu 1. Funkcia Area je metódou typu Square.

```
1  type Square struct {
2          side float64
3  }
4
5  func (sq Square) Area() float64 {
6          return sq.side * sq.side
7  }
```

Listing 1: Ukážka implementácie metódy v jazyku Go.

Zapuzdrenie sa v Go rieši pomocou exportovaných a neexportovaných prvkov štruktúry. Táto konvencia sa netýka iba štruktúr, ale aj funkcií exportovaných v rámci modulu. Názvy exportovaných metód a prvkov začínajú veľkým písmenom. Pri použití neexportovanej časti modulu, alebo štruktúry skončí kompilácia programu chybou.

```
import "math"

type Shape interface {
        Area() float64
}

type Square struct {
        side float64
}

func (sq Square) Area() float64 {
        return sq.side * sq.side
}

type Circle struct {
        radius float64
}

func (c Circle) Area() float64 {
        return math.Pi * math.Pow(c.radius, 2)
}
```

Listing 2: Použitie rozhrania na implementáciu dedičnosti v jazyku Go.

Dedičnosť sa v Go rieši dvomi spôsobmi. Prvým z nich je použitie skladania, kedy sa do podtriedy vloží anonymný prvok nadtriedy, z ktorej dedíme. Toto umožňuje kompozíciu a delegáciu objektov. Druhým spôsobom je definícia rozhrania (interface). Rozhranie je popísanie skupiny funkcií. Akýkoľvek typ, ktorý implementuje všetky funkcie rozhrania je považovaný za objekt tohto rozhrania. Kontrola, či objekt spĺňa požiadavky rozhrania nastáva v jazyku Go počas kompilácie, vývojári jazyka túto metódu nazývajú štruktúralne typovanie a vychádza z protokolov jazyka Smalltalk.[2] Príklad uvádzame v listingu 2. Rozhranie Shape vyžaduje implementáciu funkcie Area, ktorú implementuje aj typ Square, aj typ Circle a teda obe môžeme použiť ako objekt typu Shape.

2.1.2 Súbežnosť v jazyku Go - Goroutines, channels

Jednou z veľkých výhod jazyka Go oproti iným jazykom určeným na podobné použitie, ako je C++ je vstavaná podpora pre súbežnosť, ktorá sa dá dosiahnuť použitím dvoch vlastností jazyka, tzv. goroutines a údajového typu kanál (channel).

Goroutine je funkcia, ktorá sa volá pomocou kľúčového slova `go`. Táto funkcia je spustená v samostatnom ľahkom vlákne. Tieto vlákna ale nemôžeme považovať za ekvivalent vlákien - threads, v OS, kedže sú oveľa lacnejšie na vytvorenie a použitie. V implementácii sú takto bežiace goroutines multiplexované na reálne thready OS bežiacej aplikácie.

Na synchronizáciu medzi goroutines sa používa údajový typ kanál (channel). Jedná

sa o implementáciu dátovej štruktúry rad (queue) so správnou synchronizáciou prístupu, aby sa zabránilo kritickým súbehom. Súčasťou definície každého kanálu je aj údajový typ, ktorého objekty sa pomocou neho posielajú. Kanál má určenú kapacitu, prednastavená je hodnota 1. V prípade pokusu o zápis do plného kanálu táto operácia blokuje, tak isto ako pri pokuse o čítaní z prázdneho kanálu. Vzhľadom na to je jednoduché implementovať napríklad synchronizáciu pomocou semafórov.

2.1.3 RPC

Súčasťou štandardnej knižnice jazyka Go je aj modul `net/rpc`, ktorý implementuje volanie vzdialených procedúr (remote procedure call - RPC). Táto služba umožňuje, zavolať registrované funkcie na vzdialenom serveri, ku ktorému máme prístup iba cez internet.

Na funkcie ktoré je takto možné volať sú kladené nasledovné nároky:

- Funkcia musí byť metódou exportovaného typu
- Funkcia musí byť exportovaná
- Funkcia musí akceptovať 2 argumenty, oba exportovaného alebo vstavaného typu
- Druhý argument funkcie musí byť smerník
- Návratový typ funkcie musí byť typ `error`

Modul RPC podporuje niekoľko protokolov na prenos údajov, vrátane možnosti implementovať vlastný. Medzi najpoužívanejľie patrí HTTP, prípadne TCP s použitím gob kódovania údajov. Ďalší rozšírený formát na prenos údajov je JSON.

2.1.4 Serializačný formát gob

Gob je formát na prenos binárnych údajov cez sieť. Používa sa napríklad pri RPC službe na prenos objektov jazyka Go. Je samo-popisujúci, každý prvok obsahuje okrem údajov aj popis jeho typu. Tento formát vychádza dizajnovo z formátu Protocol Buffers od spoločnosti Google. Viac informácii o serializácii a význame serializovaných dát je uvedené v [8].

2.1.5 SimpleYAML

SimpleYAML[4] je knižnica pre jazyk Go, určená na prácu s textovým formátom YAML. Jedná sa o veľmi jednoduchú knižnicu, ktorá z textovej reprezentácie vo formáte YAML vytvorí ekvivalentné natívne objekty jazyka Go.

2.2 YAML

YAML je dátový serializačný jazyk, ktorý je ale zároveň aj čitateľný pre ľudí, čo z neho robí ideálny jazyk pre použitie nielen na prenos údajov, ale napríklad aj na použitie v konfiguračných súboroch.

Od verzie 1.2 je YAML oficiálne spätne kompatibilný s jazykom JSON[5], čo znamená, že parser jazyka YAML dokáže parsovať aj akýkoľvek JSON dokument.

Základnými údajovými typmi jazyka sú, podobne ako v JSON: polia, asociatívne polia, reťazce a z nich vytvorené zložitejšie vnorené štruktúry. Okrem toho jazyk YAML umožňuje používanie spätných referencií alebo definíciu používateľských typov (závisle od parsera). Všetky vlastnosti, vrátane syntaxe popisuje voľne dostupná špecifikácia jazyka [6].

Kapitola 3

Teória

V tejto kapitole sa budeme venovať teórii stojacej v pozadí programovanej aplikácie. Vysvetlíme si algoritmy na rozdelenie práce v distribuovanom prostredí, predstavíme si algoritmy vytvárania a prechádzania grafu závislostí, ktoré sú neoddeliteľnou súčasťou každého build systému.

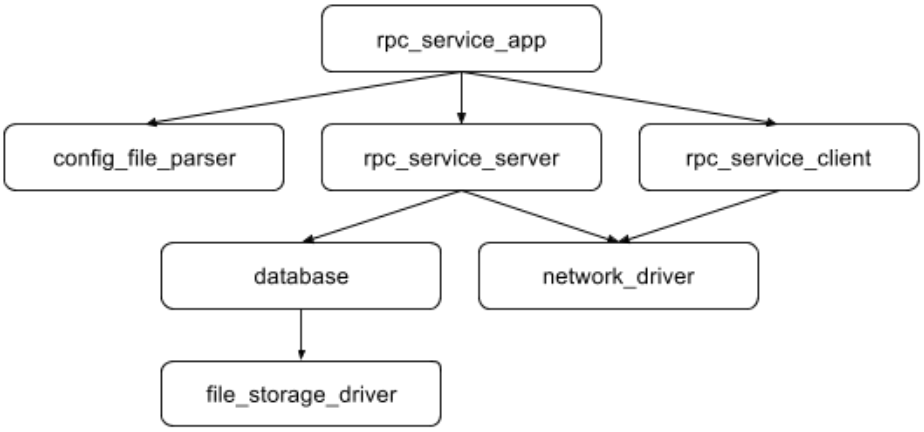
3.1 Graf cieľov

Jednou z neoddeliteľných súčastí akéhokoľvek väčšieho projektu je používanie knižníc na poskytovanie funkcionality, ktorú projekt požaduje. Typickými príkladmi je spracovávanie testových alebo binárnych formátov ako JSON alebo YAML, alebo komunikácia cez sieť.

Pre svoju úspešnú kompiláciu projekt vyžaduje, aby tieto knižnice, na ktorých závisí, boli dostupné, častokrát už v skompilovanej podobne, na kompilujúcom počítači. Takéto knižnice nazývame závislosťami projektu a v BUILD súboroch ich uvádzame, ako ciele kompilácie v zozname dependencies, z anglického termínu pre závislosť.

Naša aplikácia pri spustení prejde rekurzívne definície vyžiadaných cieľov a ich závislostí, pričom vytvára orientovaný graf, tzv. graf cieľov. Pre tento graf vyžadujeme, aby bol acyklický. V opačnom prípade nevieme splniť požiadavky cieľa, ktorý obsahuje danú cyklickú závislosť.

Po vytvorení tohto grafu je tento prejdený od listov nahor. Listy grafu cieľov sme definovali ako ciele, ktoré nemajú žiadne závislosti a sú teda koncovými uzlami grafu. Každý z cieľov grafu je následne splnený. V prípade úspešného splnenia cieľa vyžiadaného používateľom je celá kompilácia ukončená úspechom. Príklad grafu uvázame na obrázku 3.1.



Obr. 3.1: Príklad grafu cieľov pre aplikáciu zloženú z RPC servera a klienta

3.2 Algoritmus distribúcie akcií

Dôležitou časťou akéhokoľvek distribuovaného systému je rozdelenie úloch pre jednotlivé komponenty. V našej aplikácii sme na tento účel vytvorili Master komponent, ktorý je spustený používateľom. Master komponent na rozdelenie úloh používa prioritnú radu (priority queue). Pre každý zo Slave komponentov si udržujeme zoznam súborov, ktoré daný Slave komponent má k dispozícii, následne spočítame, koľko zo súborov, ktoré sú potrebné na kompiláciu sú pre každý zo serverov dostupné. K tomuto skóre pripočítame aktuálne voľný počet pracovných vlákien (v prípade, že počet voľných vlákien je 0, špecifikujeme výsledné skóre na 0). Následne akciu pošleme na server s najvyšším skóre.

Kapitola 4

Implementácia

Táto kapitola sa venuje implementácii aplikácie. Konkrétne popíšeme úlohy jednotlivých komponentov, detailne vysvetlíme fungovanie a správanie celého systému, pozrieme sa na zaujímavé časti kódu a rozoberieme rôzne problémy, na ktoré sme počas implementácie narazili.

4.1 Master

Master komponent je hlavná, používateľská, časť aplikácie. Implementovali sme ju v jazyku Go. Pri spustení aplikácia kontaktuje Slave komponenty, ktoré sú určené nastaveniami, od každého si vyžiada informácie potrebné pre distribúciu úloh a takto inicializovaná začne so spracovávaním cieľov zadaných používateľom.

V rámci inicializácie taktiež spúšťame lokálny Slave komponent, ktorého úlohou je spracovávanie úloh na lokálnom počítači.

V nasledujúcich častiach postupne popíšeme jednotlivé časti a funkcionalitu Master komponentu.

4.1.1 Spracovávanie BUILD súborov

BUILD súbory obsahujú popis projektu z pohľadu nášho systému. Súbory štandardne pomenúvame `build.yaml` a ich obsah je vo formáte YAML. Na spracovanie formátu YAML v jazyku Go sme sa rozhodli použiť knižnicu `simpleyaml`[1]. Táto knižnica prečíta text vo formáte YAML a umožní s údajmi pracovať pomocou štandardných údajových štruktúr jazyka Go, ako je napríklad integer, string, pole alebo mapa.

Listing 3 obsahuje zdrojový kód funkcie ParseFile, ktorá s použitím knižnice simpleyaml spracúva BUILD súbory a vytvára z nich objekty reprezentujúce ciele. Formát BUILD súborov, aj s príkladmi uvádzame v časti 5.3.

[1] `https://github.com/smallfish/simpleyaml`

```
65
66  func ParseFile(path, targetname, packageroot string) Target {
67          source, err := ioutil.ReadFile(path)
68          if err != nil {
69                  log.Error(err.Error(), util.Exiter)
70          }
71
72          yaml, err := simpleyaml.NewYaml(source)
73          if err != nil {
74                  log.Error(fmt.Sprintf("cannot parse file %s (not a valid YAML)", path), util.Exiter)
75          }
76
77          filedir := filepath.Dir(path)
78
79          rel_targetname := filepath.Base(targetname)
80
81          targetdata := yaml.Get(rel_targetname)
82
83          targettype, err := targetdata.Get("type").String()
84          if err != nil {
85                  log.Error(fmt.Sprintf("target %s does not exist", targetname), util.Exiter)
86          }
87
88          var target Target
89          switch targettype {
90          case "lib_c":
91                  target = MakeLibCTarget(targetname, targetdata, packageroot, filedir)
92          case "app_c":
93                  target = MakeAppCTarget(targetname, targetdata, packageroot, filedir)
94          }
95
96          return target
```

Listing 3: Funkcia ParseFile, ktorá spracúva BUILD súbory

4.1.2 Reprezentácia cieľov, rozhranie Target

```
8   type Target interface {
9           GetName() string
10          GetSources() []string
11          GetResources() []string
12          GetDependencies() []Target
13          GetOutputFile() *actions.File
14  }
```

Listing 4: Rozhranie popisujúce objekt reprezentujúci cieľ kompilácie

Každý typ cieľa reprezentujeme ako triedu implementujúcu rozhranie typu Target. Jeho definícia je uvedená v listingu 4. Postupne popíšeme jednotlivé metódy.

GetName

Metóda vracajúca názov daného cieľa ako je uvedený v BUILD súbore.

GetSources

Táto metóda vráti zoznam zdrojových súborov daného cieľa. V prípade neexistenie takýchto súborov, metóda vráti prázdne pole. Zdrojové súbory sú v BUILD súbore špecifikované ako zoznam `sources`.

Medzi zdrojové súbory patria hlavne zdrojové súbory programu v určitom programovacom jazyku, ktoré je potrebné skompilovať. Sú to najčastejšie textové súbory so špecifickou príponou.

GetResources

Metóda vracajúca zoznam súborov prostriedkov. Funkcionalita je rovnaká ako v prípade metódy GetSources. Súbory prostriedkov sú špecifikované ako zoznam `resources`.

Medzi súbory prostriedkov patria súbory, ktoré je potrebne spracovať iným spôsobom ako zdrojový kód programu. Ako príklad uvádzame ikony, obrázky alebo videá v prípade počítačových hier.

GetDependencies

Táto metóda vráti zoznam cieľov, na ktorých aktuálny cieľ závisí. Každý člen tohto zoznamu implementuje rozhranie Target. Pomocou tejto metódy pristupujeme k vzniknutému grafu závislostí.

GetOutputFile

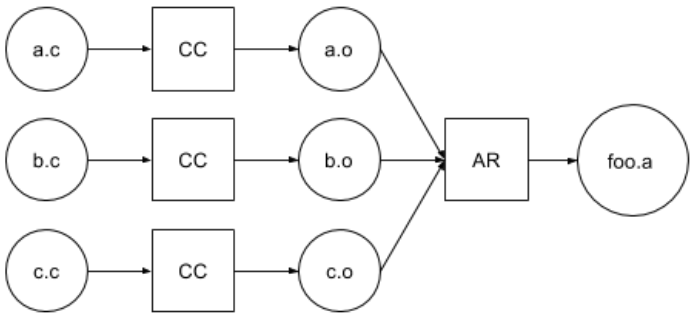
Táto metóda vráti referenciu na vygenerovaný súbor, objekt typu File. Pomocou tohto objektu pristupujeme ku bipartitnému grafu súborov a akcií.

4.1.3 Trieda LibCTarget

Trieda LibCTarget implementuje cieľ typu `lib_c`, ktorý sa používa na kompiláciu zdrojových kódov jazyku C, ktoré potom môžu využívať rôzne aplikácie.

Trieda LibCTarget vygeneruje staticky linkovanú knižnicu obsahujúcu skompilované zdrojové súbory. Takto vygenerovanú knižnicu môžeme použiť vo fáze linkovania akejkoľvek aplikácie.

Ukážka vytvoreného grafu akcií pre OS Linux je znázornená na obrázku 4.1, skladá sa z dvoch akcií: V prvej fáze je každý zdrojový súbor skompilovaný do objektového



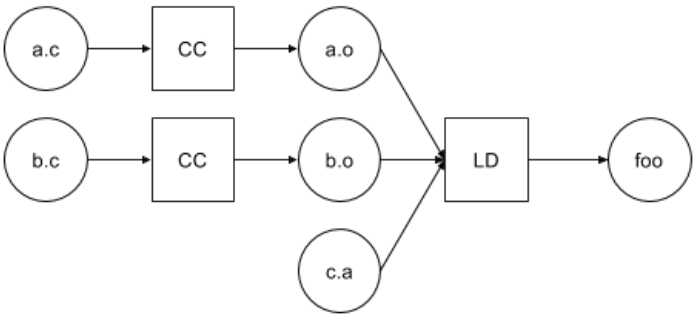
Obr. 4.1: Ukážka grafu akcií vytvoreného pre typ LibCTarget

súboru jazyka C[2] a následne sú všetky takto skompilované súbory vložené do archívu vytvoreného programom ar(1)[3].

4.1.4 Trieda AppCTarget

Táto trieda reprezentuje cieľ typu `app_c`, ktorý sa používa na špecifikáciu spustiteľného programu / aplikácie napísanej v jazyku C.

Trieda AppCTarget vygeneruje staticky linkovaný spustiteľný súbor obsahujúci skompilované zdrojové kódy, zlinkované s knižnicami uvedenými ako závislosťami.



Obr. 4.2: Ukážka grafu akcií pre typ AppCTarget, súbor c.a je výstupom cieľa `c` typu LibCTarget

Na obrázku 4.2 sa nachádza ukážka grafu akcií pre AppCTarget v prostredí OS Linux. Tento graf je tvorený akciami kompilácie (pre zdrojové súbory) a linkovania[4] výsledných objektov s knižnicami.

4.1.5 Prevod grafu cieľov do grafu akcií

Ako uvádzame v časti 4.1.2, metóda GetOutputFile rozhrania Target je zodpovedná za vytvorenie správneho grafu akcií pre daný cieľ. Graf akcií je bipartitný graf zložený

[2]kompilátor jazyka C sa štandardne označuje pomocou písmen CC, z anglického C Compiler
[3]`http://linux.die.net/man/1/ar`
[4]štandardný linker v OS UNIX a Linux má názov ld, z ktorého vzniklo označenie LD

z objektov rozhrania Action (akcie, úlohy) a objektov rozhrania File (zdrojové alebo vygenerované súbory).

Nasleduje popis funkcionality metódy pre jednotlivé triedy:

LibCTarget

```
23          }
24          return infiles, nil
25  }
26
27  func MakeCObjects(name string, sources []string, headers []string, file_list map[string]*File) []*File {
28          inputs := make([]*File, len(sources))
29
30          header_files := make([]*File, len(headers))
31          for i := range headers {
32                  f, ok := file_list[headers[i]]
33                  if ok {
34                          header_files[i] = f
35                          continue
36                  }
37                  file := &File{Filename: headers[i], Action: nil, Sem: make(chan int, 1)}
38                  file_list[headers[i]] = file
39                  header_files[i] = file
40          }
41
42          for i := range sources {
43                  filename := strings.TrimPrefix(sources[i], "//")
44                  outfilename := strings.TrimSuffix(filename, ".c") + ".o"
45
46                  infiles := make([]*File, len(header_files)+1)
47
48                  f, ok := file_list[outfilename]
49                  if ok {
50                          inputs[i] = f
51                          continue
52                  }
53
54                  f, ok = file_list[filename]
55                  var file *File
56                  if ok {
57                          file = f
58                  } else {
59                          file = &File{Filename: filename, Action: nil, Sem: make(chan int, 1)}
60                          file_list[filename] = file
61                  }
```

Listing 5: Implementácia metódy generujúcej akcie kompilácie súborov v jazyku C

V pomocnej metóde MakeCObjects, ktorú sme uviedli v listingu 5, prechádzame zoznam mien zdrojových súborov a z každého vytvoríme dvojicu objektov typu File, jeden reprezentujúci zdrojový súbor a druhý reprezentujúci vygenerovaný objektový súbor. Následne pre každú takúto dvojicu zdrojového a z neho vygenerovaného súboru

vytvoríme akciu typu CompileCAction (rozhrania Action), ktorej vstupným súborom bude zdrojový súbor a výstupným vygenerovaný objektový.

Následne vytvoríme súbor reprezentujúci vytváranú knižnicu ako celok (typu File), ktorý bude výstupným súborom prislúchajúcej akcie typu ArLinkAction (rozhrania Action). V predchádzajúcom kroku vygenerované objektové súbory použijeme ako vstupné súbory pre túto akciu. Implementačné detaily uvedieme v listingu 6.

```
48 func (t *LibCTarget) GetOutputFile() *actions.File {
49         if file_list == nil {
50                 file_list = make(map[string]*actions.File)
51         }
52
53         outfile_name := strings.TrimPrefix(t.Name, "//") + ".a"
54
55         f, ok := file_list[outfile_name]
56         if ok {
57                 return f
58         }
59
60         inputs := actions.MakeCObjects(t.Name, t.Sources, t.Resources, file_list)
61         ar_action := actions.Action{
62                 Name:    strings.TrimPrefix(t.Name, "//"),
63                 Infiles: inputs,
64                 Method:  "Task.ArLink",
65         }
66         outfile := actions.File{
67                 Filename: outfile_name,
68                 Action:   &ar_action,
69                 Sem:      make(chan int, 1),
70         }
71
72         file_list[outfile_name] = &outfile
73         return &outfile
74 }
```

Listing 6: Metóda GetOutputFile triedy LibCTarget

AppCTarget

Metóda sa na zdrojových súboroch správa rovnako ako v triede LibCTarget. Po vytvorení takejto knižnice prejdeme rekurzívne závislosti a ich výstupné súbory, spolu s práve vytvorenou knižnicou predáme ako vstupné súbory akcii LinkAction (rozhrania Action), ktorej výstupným súborom (typu File) je požadovaná spustiteľná aplikácia. Implementačné detaily sa nachádzajú v listingu 7.

```
92  func (t *AppCTarget) GetOutputFile() *actions.File {
93          if file_list == nil {
94                  file_list = make(map[string]*actions.File)
95          }
96
97          outfile_name := strings.TrimPrefix(t.Name, "//")
98
99          f, ok := file_list[outfile_name]
100         if ok {
101                 return f
102         }
103
104         c_inputs := actions.MakeCObjects(t.Name, t.Sources, t.Resources, file_list)
105         inputs := make([]*actions.File, len(c_inputs)+len(t.Dependencies))
106
107         in_count := len(c_inputs)
108
109         for i := range c_inputs {
110                 inputs[i] = c_inputs[i]
111         }
112
113         for i := range t.Dependencies {
114                 inputs[i+in_count] = t.Dependencies[i].GetOutputFile()
115         }
116
117         link_action := actions.Action{
118                 Name:    outfile_name,
119                 Infiles: inputs,
120                 Method:  "Task.LdLink",
121         }
122
123         outfile := actions.File{
124                 Filename: outfile_name,
125                 Action:   &link_action,
126                 Sem:      make(chan int, 1),
127         }
128
129         file_list[outfile_name] = &outfile
130         return &outfile
131 }
```

Listing 7: Metóda GetOutputFile triedy AppCTarget

4.1.6 Distribúcia akcií

Na účely vykonania všetkých akcií a získania výstupneho súboru pre každý cieľ sme implementovali funkciu MakeFile. Táto funkcia sa spúšťa asynchrónne s informáciami o tom aký súbor sa má získať, aké nastavenia sa pritom majú použiť. Súčasťou je aj určenie kanálu (dátový typ chan), ktorý sa použije na informovanie volajúceho o úspešnom získaní súboru. V prípade akejkoľvek chyby funkcia signalizuje chybu, oznámi ju používateľovi a spôsobí ukončenie behu aplikácie.

Získanie vstupných súborov

Funkcia MakeFile funguje rekurzívne, v prípade, že súbor sa dá získať spustením akcie, ju najprv asynchrónne zavoláme na vstupné súbory tejto akcie a počkáme, kým všetky úspešne skončia.

Zabránenie opakovanej kompilácii

Kedže chceme zabrániť opakovanej kompilácii, pokiaľ sa vstupné súbory nezmenili, v ďalšom kroku skontrolujeme novosť súborov. Za týmto účelom máme v koreňovom adresári skrytú zložku `.metadata`, ktorá obsahuje súborovú štruktúru ekvivalentnú s projektami. Každý súbor, nazvaný rovnako ako jeho ekvivalent s ktorým pracujeme obsahuje iba textovú informáciu, SHA512 kontrolný súčet obsahu súboru počas jeho predchádzajúceho použitia. Tieto informácie zapisujeme po ukončení vykonávania všetkých akcií, tesne pred ukončením behu aplikácie. Takto získaný kontrolný súčet porovnáme s kontrolným súčtom existujúceho súboru a v prípade, že sa tieto líšia, vykonáme akciu. V prípade neexistencie súboru je akcia vykonaná tiež.

Vykonanie akcie

Master komponent pred samotným vykonaním akcie musí zabezpečiť, aby volaný Slave mal prístup ku všetkým súborom, ktoré akcia potrebuje na úspešné vykonanie. To zabezpečujeme pomocou metódy SendFile RPC služby File, ktorú popisujeme v časti 4.2.2. V prípade používania lokálne bežiaceho Slave komponentu, je táto časť vynechaná.

Po úspešnom odoslaní všetkých súborov je akcia vykonaná zavolaním ekvivalentnej metódy RPC služby Task na komponente. Túto službu, vrátane metód pre jednotlivé akcie popisujeme v časti 4.2.1.

V prípade, že volaný komponent nebeží lokálne, následne vyžiadame obsah vytvoreného súboru pomocou metódy RecvFile služby File. Následne tento obsah zapíšeme do nášho pracovného priečinka, informujeme volajúceho o úspešnom skončení a ukončíme beh funkcie.

Problémy a riešenia

Najväčším problémom s ktorým sme sa stretli bolo zabezpečenie, aby akcie neboli volané viac krát, v prípade, že niekoľko rôznych súborov záleží na rovnakom súbore. Kedže systém je implementovaný ako vysoko paralelný, vznikol tu potenciálny kritický súbeh (race condition), kedy by sa počas vykonávania akcie na Slave komponente spustilo vykonávanie rovnake akcie v inom vlákne.

Na zabránenie tomuto problému sme zjednotili objekty súborov pre celú aplikáciu a prístup k nim synchronizovali pomocou zamykacej techniky semafórov.

Na zabezpečenie efektívnejšieho spracovania sme tiež v rámci kritickej sekcie, po spustení vykonávania akcie zabezpečili, aby ďalšie volania funkcie MakeFile považovali vytvorený súbor za zdrojový a teda sa nepokúšali vykonávať akcie spojené s jeho získaním. V údajovom type File sme toto dosiahli nastavením smerníku na akciu na `nil`, čo je v jazyku Go reprezentácia prázdneho smerníku.

4.2 Slave

Slave komponent je určený na vykonávanie úloh. Implementujeme ho v jazyku Go ako RPC[5] server. V rámci tohto serveru poskytujeme niekoľko implementovaných RPC služieb, z ktorých každá implementuje pomocou svojich procedúr určitú časť funkcionality.

V nasledujúcich častiach si popíšeme jednotlivé služby a procedúry ktoré poskytujú.

4.2.1 RPC služba Task

Služba Task poskytuje procedúry na vykonávanie úloh priamo súvisiacich so splnením cieľa. Tieto úlohy su ekvivalentné akciám na strane Master komponentu.

```
8  type Args struct {
9          Name        string
10         Inputs      []FileInfo
11         SendContent bool
12 }
```

Listing 8: Štruktúra špecifikujúca argumenty procedúr RPC služby Task

```
14 type Response struct {
15         ActionOutput string
16         FileContents []byte
17 }
```

Listing 9: Štruktúra špecifikujúca návratovú hodnotu procedúr RPC služby Task

Každá z týchto procedúr dostáva ako argumenty meno akcie, zoznam vstupných súborov s cestou relatívnou ku koreňovému priečinku systému, ich SHA512 kontrolným súčtom a informáciu, či volajúci vyžaduje poslanie výsledného súboru. Implementáciu uvádzame v listingu 8.

[5]Remote Procedure Call - vzdialené volanie procedúr

Výsledkom volania procedúry je výstup, ktorý vznikol počas jej vykonávania a, pokiaľ volajúci o to požiadal, obsah vzniknutého súboru. Implementáciu uvádzame v listingu 9.

Pri spúšťaní Slave komponentu umožňujeme používateľovi zvoliť, koľko úloh súčasne sa môže vykonávať pomocou argumentu `jobs`. Ako prednastavenú hodnotu sme zvolili počet CPU ktoré sa na danom počítači nachádzajú. Tento počet ovplyvňuje počet súčasne aktívnych procedúr služby Task, ktoré na zabezpečenie používajú semafór implementovaný pomocou atribútu objektu služby. Funkcionalitu semafóru sme dosiahli pomocou dátového typu channel, ktorému určujeme veľkosť na základe hodnoty argumentu `jobs`. Zároveň u každej procedúry tejto služby vyžadujeme, aby pri spustení zapísala do semafóru údaje. Pokiaľ semafór nemá kapacitu na prijatie ďalšej hodnoty, akcia zápisu v jazyku Go blokuje a teda procedúra bude čakať na uvoľnenie kapacity. Po úspešnom ako aj neúspešnom ukončení musi procedúra hodnotu zo semafóru prečítať aby tým uvoľnila miesto pre vykonávanie ďalších úloh.

```
9  func processInputs(inputs []proto.FileInfo, dir string, prepend []string) []string {
10         ret := make([]string, len(inputs)+len(prepend))
11         P := len(prepend)
12
13         for i := range prepend {
14                 ret[i] = prepend[i]
15         }
16
17         for i := range inputs {
18                 ret[i+P] = filepath.Join(dir, inputs[i].Filename)
19         }
20
21         return ret
22 }
```

Listing 10: Funkcia na prípravu argumentov pre volanie príkazu

```
24 func prepareResponse(outfilepath string, output []byte,
25         send bool, resp *proto.Response) error {
26
27         data, err := ioutil.ReadFile(outfilepath)
28         if err != nil {
29                 return err
30         }
31
32         resp.ActionOutput = string(output)
33         if send {
34                 resp.FileContents = data
35         } else {
36                 resp.FileContents = make([]byte, 0)
37         }
38
39         return nil
40 }
```

Listing 11: Funkcia na prípravu odpovede služby Task

V nasledujúcich častiach popisujeme jednotlivé procedúry služby Task. V týchto procedúrach sú činnosti ako príprava odpovede na odoslanie (listing 11) alebo príprava argumentov pre volanie externého príkazu (listing 10) často opakované, preto sme sa rozhodli ich implementácie uviesť samostatne.

CompileC

Táto procedúra vykonáva kompiláciu C súboru do objektového súboru. Vzhľadom na to, že táto akcia vyžaduje iba 1 zdrojový súbor, rozhodli sme sa signalizovať chybu kedykoľvek na vstupe dostaneme viac súborov.

```
20
21 func (t *Task) CompileC(args *proto.Args, resp *proto.Response) error {
22         t.Semaphore <- 1
23         defer func() { <-t.Semaphore }()
24
25         var outfilename string
26         if len(args.Inputs) < 1 {
27                 return errors.New("compile requires at least one file to be compiled")
28         }
29
30         if !strings.HasSuffix(args.Inputs[0].Filename, ".c") {
31                 outfilename = args.Inputs[0].Filename + ".o"
32         } else {
33                 outfilename = strings.TrimSuffix(args.Inputs[0].Filename, ".c") + ".o"
34         }
35
36         outfile_path := filepath.Join(t.Config.Root, outfilename)
37         infile_path := filepath.Join(t.Config.Root, args.Inputs[0].Filename)
38
39         output, err := exec.Command("gcc", "-std=c99", "-c", "-o",
40                 outfile_path, infile_path).CombinedOutput()
41
42         if err != nil {
43                 return errors.New(fmt.Sprintf("%s\n%s", err.Error(), output))
44         }
45
46         err = prepareResponse(outfile_path, output, args.SendContent, resp)
47         return err
```

Listing 12: Implementácia úlohy kompilácie programovacieho jazyka C pre OS Linux

V listingu 12 sme ukázali implementáciu procedúry pre OS Linux a kompilátor GNU GCC. Táto procedúra volá externý príkaz s predpripravenými parametrami, ktorého výstup posunie v návratovej hodnote volajúcemu.

ArLink

Táto procedúra vytvorí staticky linkovateľnú knižnicu. V OS Linux sa jedná o archív obsahujúci jednotlivé objektové súbory, ktorý je vytvorený programom ar, podľa ktorého sme pomenovali aj procedúru. Jej implementáciu pre OS Linux uvádzame v listingu 13. Rovnako ako v prípade CompileC procedúry sa jedná o volanie externého príkazu.

```
49
50  func (t *Task) ArLink(args *proto.Args, resp *proto.Response) error {
51          t.Semaphore <- 1
52          defer func() { <-t.Semaphore }()
53
54          var outfilename string
55          if len(args.Inputs) < 1 {
56                  return errors.New("library linking requires at least one file")
57          }
58
59          outfilename = args.Name + ".a"
60          outdir := t.Config.Root
61
62          outfile_path := filepath.Join(outdir, outfilename)
63
64          ar_args := processInputs(args.Inputs, outdir, []string{"rcs", outfile_path})
65
66          output, err := exec.Command("ar", ar_args...).CombinedOutput()
67          if err != nil {
68                  return errors.New(fmt.Sprintf("%s\n%s", err.Error(), output))
69          }
70
71          err = prepareResponse(outfile_path, output, args.SendContent, resp)
72          return err
```

Listing 13: Implementácia staticky linkovanej knižnice pre OS Linux

LdLink

Procedúra LdLink, nazvaná podľa programu ld(1) určeného na linkovanie programov, vytvára výslednú aplikáciu zlinkovaním statických knižníc uvedených ako závislosti cieľa a objektových súborov, ktoré vzniknú kompiláciou zo zdrojových kódov programu.

```
74
75  func (t *Task) LdLink(args *proto.Args, resp *proto.Response) error {
76          t.Semaphore <- 1
77          defer func() { <-t.Semaphore }()
78
79          var outfilename string
80          if len(args.Inputs) < 1 {
81                  return errors.New("application linking requires at least one file")
82          }
83
84          outfilename = args.Name
85          outdir := t.Config.Root
86
87          outfile_path := filepath.Join(outdir, outfilename)
88
89          ld_args := processInputs(args.Inputs, outdir, []string{"-o", outfile_path})
90
91          output, err := exec.Command("gcc", ld_args...).CombinedOutput()
92          if err != nil {
93                  return errors.New(fmt.Sprintf("%s\n%s", err.Error(), output))
94          }
95
96          err = prepareResponse(outfile_path, output, args.SendContent, resp)
97          return err
```

Listing 14: Implementácia linkovania aplikácie v OS Linux

V OS Linux implementácia volá GNU GCC, ako uvádzame v listingu 14, ktorý poskytuje zjednodušené linkovanie, vďaka množstvu predpripravených nastavení pre program ld.

4.2.2 RPC služba File

Služba File poskytuje možnosti prenosu súborov medzi Slave a Master komponentmi. V rámci tejto služby implementujeme nasledovné procedúry:

SendFile

Jedná sa o požiadavku volajúceho na obsah súboru nachádzajúceho sa na serveri. Súčasťou požiadavky je meno súboru a jeho kontrolný súčet algoritmom SHA512. Odpoveď servera obsahuje meno súboru a jeho obsah, vrátane metadát, ktoré, okrem iného, popisujú prístupové práva k súboru. Tieto informácie vyžadujeme za účelom zachovania napr. práv na spustenie aplikácie.

Formát požiadavky aj odpovede je opísany v listingu 15. Požiadavku reprezentuje štruktúra FileRequest a odpoveď štruktúra File.

V prípade, že by súbor neexistoval, alebo nebol prístupný, procedúra signalizuje chybu.

```
7  type File struct {
8          Filename string
9          Content  []byte
10         Mode     os.FileMode
11 }
12
13 type FileRequest struct {
14         Filename string
15 }
16
17 type FileResponse struct {
18         Filename    string
19         Checksum    [64]byte
20         Overwritten bool
21 }
```

Listing 15: Štruktúry požiadaviek a odpovedí služby File

RecvFile

Procedúra na odoslanie súboru serveru. Požiadavkou je popis súboru, štruktúra File v listingu 15. V prípade neexistencie adresára v ktorom sa má súbor umiestniť, sa ho procedúra pokúsi vytvoriť, následne zapíše súbor na disk. V prípade akýchkoľvek problémov signalizuje chybu. Výsledok procedúry popisuje štruktúra FileResponse v listingu 15, ktorá okrem mena súboru a kontrolného súčtu údajov obsahuje tiež informáciu, či táto požiadavka prepísala existujúci súbor.

4.2.3 RPC služba Util

Služba Util je určená na poskytovanie rôznych pomocných procedúr, najmä procedúr uľahčujúcich distribúciu úloh. V rámci tejto služby poskytujeme nasledovné procedúry:

NumTasks

Jednoduchá procedúra, pomocou ktorej sprístupňujeme informáciu o maximálnom povolenom množstve simultánne bežiacich úloh na komponente. Túto informáciu využíva Master na zefektívnenie distribúcie úloh, ako sme opísali v 4.1.6.

Vzhľadom na limitácie RPC protokolu, procedúra vyžaduje aspoň jeden argument. Rozhodli sme sa preto použiť najzákladnejší typ integer, ktorého hodnota je ignorovaná. Návratovou hodnotou procedúry je taktiež integer, maximálny počet úloh, ako je definovaný pri spustení Slave komponentu.

Kapitola 5

Dokumentácia systému

V tejto kapitole popíšeme inštaláciu, konfiguráciu a používanie systému. Taktiež popíšeme formát BUILD súborov a prácu s nimi. Nakoniec vysvetlíme postup vývoja nových typov cieľov a akcií.

5.1 Inštalácia

5.1.1 Prerekvizity

Aplikácia nepotrebuje pre svoj beh žiadnu externú aplikáciu, kedže jazyk Go linkuje spustiteľné súbory staticky.

Pre inštaláciu zo zdrojových súborov je potrebovaný kompilátor jazyka Go vo verzii aspoň 1.4.2. Návod na inštaláciu je na stránke `https://golang.org/dl/`. Taktiež program potrebuje knižnicu simpleyaml. Je dostupná na stiahnutie na stránke `https://github.com/smallfish/simpleyaml/` alebo pomocou príkazu:
`go get github.com/smallfish/simpleyaml`.

Zdrojový kód aplikácie sa nachádza na stránke `https://github.com/imterra/forge`, získať sa dá napríklad pomocou príkazu
`git clone https://github.com/imterra/forge.git`.

5.1.2 Master

Zdrojové kódy Master komponenta sa nachádzajú v priečinku `client/`. Pre jeho kompiláciu potrebujeme v tomto priečinku spustiť príkaz `go build`. Tento príkaz vytvorí binárny spustiteľný súbor `client`, ktorý nainštalujeme nakopírovaním do niektorého z priečinkov v premennej `PATH` a premenovaním na `forge`.

5.1.3 Slave

Inštalácia Slave komponentu je ekvivalentná k inštalácii Master komponentu, ktorú sme popísali v časti 5.1.2.

Slave komponent sa nachádza v priečinku `worker/` a jeho kompiláciu dosiahneme spustením príkazu `go build`. Tento príkaz vytvorí spustiteľný binárny súbor `worker`, ktorý prekopírujeme do priečinku z premennej `PATH` a premenujeme na `forge-server`.

5.2 Konfigurácia

V tejto časti popisujeme spôsoby nastavenia oboch komponentov. Ako prvý uvádzame Slave komponent, kedže Master vyžaduje aspoň jeden bežiaci Slave komponent na svoju činnosť.

5.2.1 Slave

Slave komponent nemá vo verzii 1.0 implementované nastavenia cez konfiguračný súbor. Všetky nastavenia sa určujú pri spustení, cez štandardné argumenty príkazového riadka.

Každý z argumentov sa pri volaní definuje vo forme `-nazov=hodnota`, kde názov je meno argumentu a hodnota je hodnota ktorú mu chceme špecifikovať.

jobs

Určuje maximálny počet súčasne bežiacich kompilačných procesov. Prednastavená hodnota je rovnaká ako počet CPU jadier na počítači.

port

Určuje TCP port, na ktorom Slave prijíma volania RPC služby pre vykonanie úloh. Táto hodnota je potrebná pri definovaní adresy Slave komponentu na strane Mastera. Prednastavená hodnota je 1103.

root

Určuje koreňovú zložku projektov na serveri. V rámci tejto zložky sa ukladajú všetky súbory ktoré Slave prijme, alebo vytvorí. Pokiaľ by hodnota argumentu nebola nastavená, použije sa hodnota premennej prostredia `FORGE_ROOT`. V prípade, že ani premenná prostredia nie je nastavená, prednastavenou hodnotou je priečinok `forge/` v domovskom adresári používateľa, ktorý Slave komponent spustil. Priečinok takto určený musí existovať.

5.2.2 Master

Master komponent používa konfiguračný súbor vo formáte YAML, ktorý sa, pokiaľ nie je povedané inak, nazýva `.forge.yaml` a nachádza sa v domovskom adresári používateľa spúšťajúceho Master komponent. V prípade jeho neexistencie sa pokúsime získať nastavenia z globálneho súboru `/etc/forge.yaml`. Konfiguračný súbor nie je vyžadovaný na úspešné spustenie aplikácie.

Tento súbor môže špecifikovať rovnaké hodnoty ako argumenty na príkazovom riadku, ktoré sú mu však nadradené.

jobs

Toto nastavenie špecifikuje počet súčasne bežiacich úloh pre lokálne spustený Slave komponent. Pokiaľ je toto číslo nastavené na 0, lokálny komponent sa nespúšťa. Prednastavenou hodnotou je počet jadier CPU na lokálnom počítači.

root

Určuje koreňovú zložku projektov na serveri. Všetky súbory projektov vyžadované na kompiláciu sa musia nachádzať v tejto zložke a ich cesty musia byt v BUILD súboroch špecifikované relatívne k nej. V prípade nešpecifikovania tohto argumentu na príkazovom riadku sa použije premenná prostredia `FORGE_ROOT`, v prípade jej neexistencie sa použije nastavenie z konfiguračných súborov. V prípade, neexistencie konfiguračných súborov sa ako koreňová zložka použije priečinok `.forge/` z domovského adresára používateľa.

worker

Určuje adresy Slave komponentov (okrem lokálneho ak sa tento spúšťa automaticky), ktoré sa majú použiť pri kompilácii. Adresy sa uvádzajú v štandardnom tvare `adresa:port`, kde adresa je buď IP alebo DNS adresa servera.

Na príkazovom riadku môže byť viac Slave komponentov uvedených použitím niekoľkých argumentov worker, alebo ako čiarkami oddelený zoznam pre jeden argument.

```
forge --worker=slave1.workers.com:1103,slave2.workers.com:1103
forge --worker=slave1.workers.com:1103 --worker=slave2.workers.com:1103
```

V prípade konfiguračného súboru sa zoznam Slave komponentov určuje ako zoznam jazyka YAML pod hlavičkou `worker`.

5.3 BUILD súbory

BUILD súbory popisujú vzťahy medzi súbormi v projekte ako sériu cieľov, ktorých splnením sa splní požadovaná úloha, najčastejšie kompilácia. Ako formát týchto súborov sme použili formát YAML, ktorý sme popísali v časti 2.2.

```
auvm:
  type: app_c
  dependencies: [ main , stack , util , parse , init , object , intable , ins , auvmlib , lib/io ]

stack:
  type: lib_c
  sources: [ stack.c ]
  resources: [ auvm.h , stack.h , object.h ]

util:
  type: lib_c
  sources: [ util.c ]
  resources: [ auvm.h , stack.h , object.h ]
```

Listing 16: Ukážka BUILD súboru projektu auvm

Na listingu 16 sme uviedli príklad BUILD súboru projektu auvm[1].

5.3.1 Definícia cieľa

Definíciou cieľa je asociatívne pole, ktorého kľúče a s nimi asociované hodnoty, spoločne s názvom cieľa, presne popisujú každý cieľ. Teraz popíšeme všetky povolené kľúče.

type

Jediná informácia vyžadovaná v akomkoľvek cieli, popisuje výsledky aj podmienky pre splnenie daného cieľa. V súčasnej verzii podporujeme ciele typu `lib_c` pre knižnice jazyka C a `app_c` pre spustiteľné aplikácie v jazyku C.

Výsledkom splnenia cieľa typu `lib_c` je súbor s názvom `nazovcieľa.a` obsahujúci staticky linkované objektové súbory, ktoré vzniknú kompiláciou zdrojových súborov.

Po splnení cieľa s typom `app_c` je spustiteľný súbor s rovnakým názvom ako je názov cieľa. Tento súbor vznikne zlinkovaním knižníc špecifikovaných ako závislosti a skompilovaných zdrojových súborov.

sources, resources

Hodnoty `sources` a `resources` sú polia jazyka YAML obsahujúce za sebou vymenované mená súborov. Povolené a podporované je určenie súborov v adresári BUILD súboru alebo jeho podadresároch pomocou určenia cesty relatívnej k polohe BUILD súboru.

[1] `https://github.com/pepol/auvm`

V prípade typu cieľa v jazyku C sa každý súbor z poľa `sources` samostatne skompiluje do objektového súboru, následne sú všetky tieto súbory zlinkované do statickej knižnice a, v prípade typu aplikácie, následne zlinkované s výstupmi cieľov, na ktorých tento cieľ závisí, vytvárajúc spustiteľnú aplikáciu.

Pre ciele v jazyku C sa súbory z poľa `resources` používajú na určenie nekompilovaných súborov, ktoré sú ale ku kompilácii potrebné. Jedná sa napríklad o hlavičkové súbory jazyka C. Súbory v tomto poli sú totižto iba prenesené na kompilujúci počítač, prípadne pri opätovnej kompilácii je aj ich obsah kontrolovaný voči zmenám.

dependencies

Pole `dependencies` obsahujé názvy cieľov, ktorých splnenie je predpokladom k úspešnému splneniu aktuálneho cieľa. Meno cieľa môže byť uvedené buď ako meno cieľa z rovnakého BUILD súboru, alebo ako meno cieľa definovaného v niektorom z podpriečinkov.

V prípade, že projekt závisí na externej knižnici a táto tiež používa systém Forge, môžeme ju špecifikovať pomocou plne kvalifikovaného mena cieľa (FQTN[2]). Tvar FQTN popisujeme v časti 5.3.2.

5.3.2 Plne Kvalifikované Meno Cieľa - FQTN

Každý cieľ má meno, ktoré musí byť unikátne v rámci BUILD súboru v ktorom je definovaný. Avšak nemôžeme vyžadovať, aby mená boli unikátne medzi všetkými projektmi, ktoré na sebe vzájomne závisia. Preto systém Forge obsahuje tzv. FQTN, plne kvalifikované meno cieľa. V princípe je FQTN ekvivalentné absolútnej ceste v priečinkovej štruktúre v OS UNIX. Na rozlíšenie od absolútnej cesty sme rozhodli, že FQTN začína dvomi znakmi lomítka (`//`), za ktorými nasleduje postupnosť mien priečinkov v tomto virtuálnom priečinkovom strome, oddelených znakmi lomítka. Príkladmi FQTN sú napríklad: `//auvm/auvm`, `//auvm/lib/io`, alebo
`//github/imterra/forge/client/client`.

FQTN znižujú, ale neeliminujú úplne kolíziu mien cieľov, preto sme v časti 5.3.3 uviedli rady určené najmä pre používateľov verejných úložísk zdrojových kódov.

V prípade použitia systému Forge v rámci spoločnosti je odporúčané použiť doménové meno firmy ako prefix pre hierarchiu projektov. Príkladom je naša open-source organizácia Imterra, ktorej všetky FQTN začínajú s `//imterra/`.

[2]Fully-Qualified Target Name

FORGE_ROOT, BUILD súbory a FQTN

V časti 5.4.2 popisujeme premennú prostredia `FORGE_ROOT`, ktorá špecifikuje koreňový priečinok systému Forge na danom počítači. FQTN špecifikuje výskyt cieľa vzhľadom ku tomuto priečinku.

Meno cieľa je časť FQTN za posledným lomítkom. Pre nájdenie definície cieľa sa cieľ s takýmto názvom hľadá v BUILD súbore, ktorý sa nachádza v podpriečinku `FORGE_ROOT` určeným cestou končiacou posledným lomítkom FQTN. Teda pre FQTN `//imterra/forge/test/foo/bar` sa cieľ `bar` bude hľadať v súbore `$FORGE_ROOT/imterra/forge/test/foo/build.yaml`.

5.3.3 Odporúčania pre písanie BUILD súborov

Pri písaní BUILD súborov odporúčame používať zjednodušený formát YAML a pre dobrú čitateľnosť nevyužívať spätnú kompatibilitu s jazykom JSON. Pokiaľ by ale súbor bol generovaný automatickým prevodom z iného formátu, je JSON formát akceptovateľný.

Pri zverejňovaní projektov používajúcich Forge na verejných úložiskách zdrojových kódov typu GitHub oporúčame používať FQTN začínajúce reťazcom `//github/USERNAME/REPOSITORY`[3], kde `USERNAME` je používateľské meno (alebo názov organizácie), ktorá repozitár zverejnila a `REPOSITORY` je názov repozitára. Ekvivalentné pomenovanie odporúčame aj pre ďalšie hostingové služby. Toto umožňuje používateľom systému závislého na takto publikovanom systéme jednoducho stiahnuť zdrojové kódy bez potreby hľadania na internete.

Pri formátovaní YAML súborov odporúčame využívať inline verziu zápisu poľa pre polia s malým počtom hodnôt, ale pre dlhé hodnoty, alebo veľký počet hodnôt odporúčame, pre zlepšenie čitateľnosti, definíciu poľa cez odrážky.

5.4 Použitie systému na kompiláciu

V tejto časti popisujeme spúšťanie a používanie systému Forge. Ako prvý uvádzame Slave komponent, kedže bežiaci Slave komponent je potrebný pre správnu funkcionalitu.

5.4.1 Slave

Slave komponent spúšťame, obvykle na serveri, pomocou príkazu `forge-server`. Nastavenia koreňového priečinku, portu na ktorom je dostupný a počtu kompilačných vlákien vykonávame podľa na to určených argumentov, ako sme uviedli v časti 5.2.1.

[3] Adresa projektu na stránke bude `https://github.com/USERNAME/REPOSITORY`

Slave komponent sa na rozdiel od tradičných démonov nespúšťa na pozadí, preto je odporúčané spustiť ho ako proces na pozadí pomocou operátora shellu `&`, ako sme ukázali v listingu 17, prípadne pomocou zastavenia bežiaceho procesu a použitia príkazu `bg`.

```
~> forge-server --jobs=4 --port=4200 --root=/tmp/forgebuild &
```

Listing 17: Ukážka spustenia Slave komponentu na pozadí s neštandardnými nastaveniami

5.4.2 Master

Pri spustení Master komponentu mu, okrem prípadných nastavení pomocou argumentov, ako uvádzame v časti 5.2.1, zadávame aj názov cieľa alebo niekoľko cieľov (oddelených medzerami), ktoré vyžadujeme splniť. Ciele uvádzame pomocou ich FQTN, avšak, ak aktuálny adresár je podpriečinkom koreňového adresára systému, môžeme uviesť aj relatívnu cestu k cieľu, od tohto aktuálneho adresára. Obidva spôsoby adresácie sme ukázali v listingu 18.

V prípade spustenia Master komponentu s nastaveným parametrom `jobs` na hodnotu inú ako 0, tento spustí ako podproces lokálny Slave komponent počúvajúci na porte 1103, ktorý je po ukončení vykonávania Master komponentu taktiež ukončený.

```
/home/user/src/foo> forge --root=/home/user/src --worker 10.0.1.7:1103 //auvm/auvm foo
```

Listing 18: Kompilácia cieľa auvm pomocou FQTN a lokálneho cieľa foo, s použitím vzdialeného Slave komponentu

5.5 Rozširovanie funkcionality systému

Dôležitou schopnosťou systému Forge musí byť rozšíriteľnosť. Preto v tejto časti popisujeme spôsoby na implementáciu 2 najčastejších rozšírení: akcií a typov cieľov.

5.5.1 Pridávanie nových akcií

Implementácia novej akcie je prevedená najmä na strane Slave komponentu. Na jej sprístupnenie používateľom ju treba implementovať ako metódu typu `Task` z modulu `tasks`. Táto metóda musí spĺňať všetky kritériá pre metódu RPC služby. V prípade, že by typ `Task` nespĺňal všetky požiadavky pre akciu, napríklad neobsahuje isté informácie,

je možné metódu implementovať na inom, novom, type, ktorý je ale potom potrebné zaregistrovať ako RPC službu v hlavnom module Slave komponentu.

Na strane Master komponentu nie je akciu potrebné implementovať, je potrebné len uviesť správnu RPC metódu pri jej vytvárani z cieľa. Ako vytvoriť nový typ cieľa uvádzame v ďalšej časti.

5.5.2 Pridávanie nových typov cieľov

Pridávanie nových typov cieľov prebieha výlučne na strane Master komponentu. Na úspešnú implementáciu je potrebné pridať jeho podporu do spracovávnia BUILD súborov a následne definovať vytvorený objekt, tak aby tento spĺňal požiadavky rozhrania `Target`.

Rozhranie `Target` vyžaduje implementáciu 5 metód, kde najdôležitejšia pre splnenie cieľa je metóda `GetOutputFile`, ktorá prekladá cieľ do grafu súborov a akcií na nich prevedených. Odporúčame nový cieľ implementovať pomocou typu definovaného v module `target`, kedže tento modul interne sprístupňuje hešovaciu tabuľku všetkých doteraz vytvorených súborov. Táto sa používa na zabránenie vytvorenia viacerých objektov typu `File`, ktoré reprezentujú rovnaký fyzický súbor a teda následné problémy s opakovanou kompiláciou. Táto tabuľka je dostupná ako modulu prístupná premenná `file_list`. Jej kľúčmi sú cesty k súborom relatívne ku koreňovému adresáru systému Forge.

Na sprístupnenie takto implementovaného typu cieľa tvorcom BUILD súboru, je potrebné o ňom informovať funkciu `ParseFile`, ktorá spracúva tieto súbory. Na konci funkcie sa nachádza `switch-case` výraz, ktorý na základe textovej reprezentácie typu cieľa v BUILD súbore vráti objekt potrebného typu. Na spracovanie YAML údajov, ktoré sú dostupné sme pre typy `lib_c` a `app_c` implementovali pomocné funkcie `MakeLibCTarget` a `MakeAppCTarget`. Odporúčame inšpirovať sa ich implementáciou pri tvorbe nových pomocných funkcií.

Pre zjednodušenie vytvárania závislostí v podobe ďalších cieľov sme vytvorili funkciu `MakeDependencies`, ktorá z YAML údajov spracuje pole `dependencies` a následne opakovaným volaním funkcie `ParseFile`, vytvorí tieto ciele. Použitie tejto funkcie je silne odporúčané.

Kapitola 6

Testovanie systému

V tejto kapitole otestujeme a porovnáme výkon systému na zdrojových kódoch existujúcich open-source projektov. Porovnáme čas behu kompilácie pomocou štandardného kompilačného programu GNU Make, pomocou Forge kompilujúceho iba lokálne a pomocou Forge kompilujúceho pomocou 2 Slave komponentov, jedného bežiaceho na lokálnom počítači a druhého bežiaceho na vzdialenom serveri.

6.1 Výber projektov na testovanie

Kedže Forge podporuje zatiaľ iba kompiláciu zdrojových súborov jazyka C, rozhodli sme sa použiť projekt písaný iba v jazyku C. Druhým kritériom výberu bola jednoduchosť organizácie projektu a automatizovateľnosť tvorby BUILD súboru. Zároveň sme ale vyžadovali, aby bol projekt rozsiahly, vzhľadom na povahu testovania. Pri malom, napríklad jednosúborovom projekte by nebola distribúcia kompilácie použiteľná, prípadne by väčšiu rolu vo výsledkoch hralo aktuálne zaťaženie počítača. Toto by mohlo mať za následok skreslenie skutočných výsledkov.

Pre testovanie sme vybrali projekt redis[1], ktorý spĺňal všetky naše požiadavky. Tento projekt na kompiláciu používa iba štandardný nástroj GNU Make, bez použitia GNU Autotools, čo značne zjednodušilo tvorbu BUILD súboru.

6.2 Postup testovania

Pre každú z testovaných konfigurácií sme spustili kompiláciu programu `redis-server`. Kompiláciu sme opakovali pre každý program 5 krát, aby sme zabránili skresleniu výsledkov kvôli vyťaženiu počítača. Všetky opakované kompilácie boli spúšťané na čistých zdrojových kódoch, aby sme zabezpečili reálne spustenie kompilácie.

[1] `https://redis.io`

6.2.1 Parametre testovania

Testovanie prebiehalo na laptope s operačnou pamäťou s kapacitou 16GB a dvojjadrovým procesorom Intel Core i7-5500U s frekvenciou 2.4GHz a podporou technológie HyperThreading, ktorá spôsobuje, že procesor je schopný pracovať ako štvorjadrový. Pri použití programu GNU Make, ako aj lokálneho Slave komponentu systému Forge sme preto nastavili počet pracujúcich vlákien na 4.

Vzdialený Slave komponent bežal na serveri so štvorjadrovým procesorom Intel Atom C2750 s frekvenciou 2.4GHz a operačnou pamäťou s kapacitou 4GB.

6.2.2 Vytvorenie BUILD súboru

BUILD súbor pre projekt redis sme vytvorili na základe originálneho súboru Makefile, ktorý sa štandardne používa pri kompilácii pomocou nástroja GNU Make. Tento súbor obsahuje pravidlo, ktoré zabezpečí generáciu tzv. súboru závislostí, popisujúceho všetky súbory, ktoré zdrojový súbor jazyka C používa pomocou direktívy `#include`.

Takto vytvorený súbor závislostí sme potom skriptom `dep2buildfile.sh`[2] skonvertovali na takmer hotový BUILD file. Tento skript ale vygeneruje len ciele pre jednotlivé knižnice programu. Následne sme pridali ďalší cieľ reprezentujúci výslednú aplikáciu `redis-server`, ktorá závisela na knižniciach, podľa špecifikácie cieľa `redis-server` v súbore Makefile.

6.2.3 Meranie času

Meranie času sme zabezpečili pomocou štandardnej utility OS Linux time[3]. Táto utilita na príkazovom riadku očakáva príkaz, ktorého čas vykonávania meria. Okrem celkového času vykonávania zbiera aj hodnoty o vyťažení CPU (toto vyťaženie môže dosiahnuť maximálne hodnotu $N \times 100\%$, kde N je počet logických CPU na počítači.

6.3 Výsledky testovania

V tejto časti uvádzame prehľad nameraných výsledkov, spolu s vysvetleniami a poznámkami. V tabuľkách sme uviedli prehľad vyťaženia procesorov a dĺžku trvania behu programu pre používateľa, to znamená od spustenia programu po jeho úspešné ukončenie. Posledný riadok tabuľky obsahuje priemerné hodnoty.

[2] Tento skript je zverejnený spolu so zdrojovými kódmi programu Forge

[3] `http://linux.die.net/man/1/time`

GNU Make

Číslo pokusu	Využitie CPU	Čas
1.	332%	21.74s
2.	318%	21.79s
3.	324%	21.72s
4.	272%	21.78s
5.	289%	21.69s
Priemer	307%	21.74s

Tabuľka 6.1: Výsledky pre GNU Make

Nedistribuovaný Forge

Číslo pokusu	Využitie CPU	Čas
1.	270%	6.85s
2.	323%	6.90s
3.	317%	6.71s
4.	335%	6.96s
5.	332%	6.87s
Priemer	315%	6.85s

Tabuľka 6.2: Výsledky pre nedistribuovaný Forge

Z výsledkov vidíme, že systém Forge je asi 3-krát rýchlejší ako GNU Make. Toto je pravdepodobne kvôli tomu, že Make spúšťa príkazy kompilácie nepriamo, spustením shellu, ktorý potom spúšťa samotný kompilátor, na rozdiel od systému Forge, ktorý spúšťa kompilátor priamo ako dcérsky proces. Ďalším dôvodom môže byť samotné parsovanie build programov, kde Makefile je oveľa zložitejšie spracovávať, najmä kvôli rôznym definíciam, premenným a volaniam externých príkazov. Formát YAML, ktorý používa pre svoje BUILD súbory Forge je jednoducho a rýchlo spracovateľný.

Distribuovaný Forge

Pri testovaní použitia serveru na kompiláciu sme sa rozhodli vyskúšať 2 varianty nastavení. Obe nastavenia využívajú plnú kapacitu serveru, pričom prvé zároveň využíva aj plnú kapacitu testovacieho laptopu, zatiaľčo druhé využíva laptop iba z polovice.

Číslo pokusu	Využitie CPU	Čas
1.	263%	6.22s
2.	302%	6.04s
3.	278%	5.68s
4.	300%	6.20s
5.	302%	6.25s
Priemer	289%	6.08s

Tabuľka 6.3: Výsledky pre distribuovaný Forge so 4 lokálnymi vláknami

V porovnaní s predchádzajúcim, čisto lokálnym behom systému vidíme mierny pokles trvania kompilácie, ako aj pokles vyťaženia procesora, presne ako bolo očakávané. Zrýchlenie spôsobené zdvojnásobením počtu kompilujúcich vlákien by sa prejavilo viditeľnejšie pri väčšom projekte.

Číslo pokusu	Využitie CPU	Čas
1.	25%	2.45s
2.	18%	2.69s
3.	28%	2.46s
4.	26%	2.41s
5.	26%	2.42s
Priemer	24%	2.48s

Tabuľka 6.4: Výsledky pre distribuovaný Forge s 2 lokálnymi vláknami

Výsledok behu s 2 lokálnymi vláknami môže vyzerať veľmi prekvapivo. Hlavným dôvodom tohto zrýchlenia je pravdepodobne uvoľnenie jedného z lokálnych vlakien na spracovávanie stálej sieťovej komunikácie so serverom a teda využívanie reálnych 6 jadier. Pri použití 4 lokálnych vlákien je totiž vlákno zabezpečujúce komunikáciu blokované vláknom so spusteným kompilačným procesom, čo môže spôsobovať spomalenie systému.

Pre výsledky lepšie vypovedajúce o efektívnosti implementácie by bol potrebný test kompilácie projektu s rádovo stovkami súborov na desiatkach serverov. Problémom takéhoto testu je ale náročnosť na prostriedky. Druhým problémom s ktorým sme sa stretli bolo nájdenie vhodného, dostatočne veľkého projektu. Z týchto dôvodov sme tento test neuskutočnili.

Záver

Úspešne sa nám podarilo implementovať jednoduchý distribuovaný build systém napísaný v jazyku Go. Pri implementácii systému sme sa držali konvencií jazyka Go a snažili sme sa všetky zdroje používateľského počítača využívať čo možno najefektívnejšie. Systém Forge sme zverejnili ako open-source pod MIT Licenciou na stránke `https://github.com/imterra/forge`.

Z výsledkov testov v kapitole 6 vidíme, že systém je naprogramovaný efektívne a dokázal poskytnúť takmer 9-násobné zrýchlenie iba s použitím jedného servera oproti lokálne bežiacemu systému GNU Make.

Najväčším problémom s ktorým sme sa stretli bol návrh algoritmu na distribúciu kompilačných úloh na servery. Pri našej implementácii sme použili jednoduchý naivný algoritmus na základe vyťaženosti stroja a dostupnosti súborov. Pre efektívnejšiu distribúciu by bolo potrebné analyzovať, aké ďalšie postupy by bolo vhodné použiť.

V budúcnosti by sme radi rozšírili systém o podporu ďalších programovacích jazykov, ako aj zlepšili podporu aktuálne podporovaných. Taktiež by sme radi pridali podporu na zabezpečenú komunikáciu medzi komponentmi, zdieľané úložisko súborov a podporu pre systémy správy verzií zdrojových kódov, aby bola aplikácia reálne použiteľná v komerčnej sfére.

Literatúra

[1] Bazel feature roadmap. Dostupné z http://bazel.io/roadmap.html.

[2] Effective go - interfaces. Dostupné z https://golang.org/doc/effective_go.html#interfaces.

[3] The go programming language specification. Dostupné z https://golang.org/ref/spec#Introduction.

[4] simpleyaml - godoc. Dostupné z https://godoc.org/github.com/smallfish/simpleyaml.

[5] Yaml 1.2 specification - relation to json. Dostupné z http://www.yaml.org/spec/1.2/spec.html#id2759572.

[6] Yaml ain't markup language (yaml) version 1.2. Dostupné z http://www.yaml.org/spec/1.2/spec.html.

[7] Christian Kemper. Build in the cloud: How the build system works, 2011. Dostupné z http://google-engtools.blogspot.sk/2011/08/build-in-cloud-how-build-system-works.html.

[8] Rob Pike. The go blog - gobs of data, 2011. Dostupné z https://blog.golang.org/gobs-of-data.

Príloha A

K práci je priložené CD so zdrojovým kódom aplikácie.