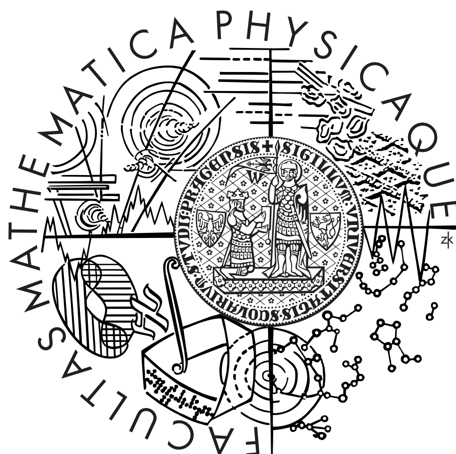
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta



DIPLOMOVÁ PRÁCA

Miroslav Lenčéš

Jazyk pro programování asynchronních serverů

Katedra softwarového inženýrství

Vedúci bakalárskej práce: RNDr. Leo Galamboš Ph.D.
Študijný program: Informatika
Študijný smer: Softwarové systémy

Praha 2009

Na tomto mieste by som sa rád poďakoval vedúcemu Diplomovej práce, RNDr. Leo Galambošovi Phd., rodine a priateľom za podporu a pripomienky, ktoré mi pomohli vypracovat túto prácu.

Prehlasujem, že som svoju diplomovú prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičiavaním práce.

V Prahe dňa 1. februára 2009. Miroslav Lenčéš

Obsah

1 Úvod **1**
1.1 Motivácia 1

2 Analýza a návrh riešenia **3**
2.1 Modelová aplikácia 3
2.2 Programovací jazyk 4
2.3 Štruktúra prostredia 4
2.4 Prenos užívateľských dát 5
2.5 Sieťová komunikácia 6
2.6 Paralelizmus 8
2.6.1 Synchronizácia dát 9
2.7 Ciele práce 10
2.8 Zadanie 12

3 Implementácia **13**
3.1 Moduly aplikácie 13
3.2 Pomocné dátové štruktúry 14
3.2.1 DatagramProperty 14
3.2.2 DatagramProperties 14
3.2.3 Datagram 15
3.2.4 NodeInfo 18
3.2.5 HostInfo 19
3.2.6 QueueItem 20
3.2.7 RelocationItem 20

4 Bloky výpočtu **22**
4.1 Nastavenia bloku 23
4.2 Generátor 24
4.3 Voľné a fixné bloky 25
4.4 Hrany 25
4.4.1 Synchronizačné hrany 26

5 Modul Router 28
5.1 Vytvorenie smerovacieho grafu . . . 28
5.1.1 Načítanie definovaných blokov . . . 28
5.1.2 Vytvorenie grafu . . . 29
5.1.3 Kontrola grafu . . . 30
5.2 Ukončenie blokov . . . 31
5.3 Verejné rozhranie . . . 32

6 Modul Network 35
6.1 Inicializácia . . . 35
6.2 Spojenie . . . 36
6.2.1 Server . . . 37
6.2.2 Klient . . . 37
6.3 Prijímanie správ . . . 38
6.4 Ukončenie sieťového uzlu . . . 38
6.5 Výpadok v sieti . . . 39
6.6 Verejné rozhranie . . . 39

7 Modul ThreadPool 42
7.1 Vytvorenie inštancie . . . 42
7.1.1 Rozloženie záťaže . . . 43
7.2 Ukončenie inštancií . . . 44
7.3 Verejné rozhranie . . . 44

8 Modul Connector 45
8.1 Inicializácia . . . 45
8.2 Preposielanie dát . . . 45
8.2.1 Vloženie datagramu do fronty . . . 46
8.2.2 Spracovanie datagramu . . . 46
8.3 Synchronizácia . . . 47
8.3.1 Preskočenie synchronizácie . . . 48
8.3.2 Nejednoznačnosť synchronizácie . . . 49
8.4 Ukončenie bloku . . . 50
8.5 Presúvanie bloku . . . 50
8.5.1 Inicializácia presunu . . . 50
8.5.2 Hlasovanie . . . 51
8.5.3 Kolízia presunu . . . 52
8.5.4 Presun bloku . . . 53
8.6 Výpadok spojenia . . . 54
8.7 Verejné rozhranie . . . 54

9 Záver 57
9.1 Výber použitých metód . . . 58

A Obsah média **60**
A.1 Zdrojový kód 60
A.1.1 Požiadavky 60
A.1.2 Kompilovanie 60
A.2 Príklad použitia 61
A.2.1 Popis aplikácie 61
A.2.2 Bloky aplikácie 61
A.2.3 Výpočet aplikácie 62

Název práce: Jazyk pro programování asynchronních serverů
Autor: Miroslav Lenčéš
Katedra: Katedra softwarového inženýrství
Vedúci diplomovej práce: RNDr. Leo Galamboš Ph.D.
e-mail vedúceho: Leo.Galambos@mff.cuni.cz
Abstrakt: Prostředí PLAS slouží k jednoduchému vytváření aplikací, které jsou spustitelné na různých platformách, v podobě komponentového systému v programovacím jazyce JAVA. Výsledná aplikace může být distribuována na vícero uživatelem definovaných vzájemně přepojených počítačích. Uživatel plně ovládá výpočetní část aplikace a tok dat, tedy jednotlivé komponenty a hrany mezi nimi. Prostředí zabezpečuje přenos a synchronizaci dat potomkům na základě definovaných pravidel, zapisovanými pomocí anotací a je pro uživatele plně transparentní.
Zpracovávaná data mohou být rozděleny na vícero kopií a tedy jednotlivé části komponentového systému mohou zpracovávat data simultánně. V definovaných komponentach jsou různé kopie dat opět sjednoceni.
Klíčová slova: distribuovaná aplikace, paralelní spracování, asynchrónní server, multiplatformové prostředí, komponentový systém

Title: Programming Language for Asynchronous Servers
Author: Miroslav Lenčéš
Department: Department of Software Engineering
Supervisor: RNDr. Leo Galamboš Ph.D.
Supervisor's e-mail address: Leo.Galambos@mff.cuni.cz
Abstract: PLAS framework is designed for easy development of multiplatform applications in form of component system in JAVA programming language. Target application can be distributed to multiple connected, user defined computer. User has full control over computation part of component and data flow ie. vertices of component system and edges between them. Framework acts as a transparent communication layer, delivers data to child states using user defined routing informations, which are defined in annotation of an object.
Data can be split into multiple copies on demand (when state has multiple edges to child nodes defined) so parts of aplicationi can process their own copy simultaneously. Split data can be again merged in user defined parts of component system.
Keywords: distributed application, parallel computing, multiplatform framework, component system

Kapitola 1

Úvod

1.1 Motivácia

V dnešnej dobe je trendom vývoj aplikácií, ktoré si užívateľ môže prispôsobiť či už po stránke grafickej alebo v malej miere aj funkčnosti. Azda najvýraznejšie sa tento trend prejavuje vo vývoji webových aplikácií a grafických rozhraní, teda v prezentačnej vrstve. Avšak prispôsobiteľnosť je v niektorých prípadoch vyžadovaná resp. výhodná aj vo výpočetnej časti programu. Na príklad pri spracovaní dát, ktoré síce majú rovnaký charakter, ale zásadne sa líši ich kódovanie, spôsob zápisu.

Na takýto problém som narazil pri vytváraní programu, ktorý slúžil na spracovávanie meteorologických a klimatologických dát. Tieto dáta pochádzali z rôznych zdrojov, čo samozrejme prinášalo rozdielne spôsoby ich zápisu, definície napr. rôzne jednotky fyzikálnych veličín, kódovanie hodnôt na zlepšenie prenositeľnosti (prevedenie typu float na celočíselný pri vhodnej mierke a posune). Každý nový zdroj dát teda vyžadoval väčšiu či menšiu úpravu zdrojového kódu, čo vo väčšine prípadov znamenalo zásah programátora.

Tak pre spracovanie odlišnej množiny dát vznikli samostatné verzie programov, ktoré dáta priamo spracovávali alebo len pripravili prevedením do známeho formátu. Takýto postup je však časovo náročný ako pre programátora (z hľadiska analýzy rozširovateľnej verzie programu, alebo vytvorením a úpravou N skoro identických kópií programu) tak aj pre užívateľa samotného, ktorý musí čakať na úpravu programu na spracovanie konkrétneho typu dát.

Preto som sa rozhodol k danému problému pristupovať odlišne a snažil som sa nájsť nie príliš komplexný nástroj v ktorom by mal užívateľ po vytvorení kostry programu dostatočnú voľnosť v úprave toku dát respektíve

výpočetných častí. Pretože som však žiadny vhodný nástroj takéhoto typu nenašiel, rozhodol som sa v tejto práci predostrieť implementáciu prostredia, ktoré by takúto funknčnosť umožňovalo.

Kapitola 2

Analýza a návrh riešenia

2.1 Modelová aplikácia

Modelová aplikácia podľa, ktorej bolo prostredia PLAS formované slúži na spracovanie klimatologických dát, ktoré sú uložené prevažne vo formáte netCDF. Tento formát umožňuje ukladanie n-rozmerných dát, ale tvar ukladaných dát bližšie neupresňuje. Jedná sa teda o aplikáciu na spracovanie dát s variabilným formátom vstupných dát. Táto aplikácia bude nasadená na bežnej užívateľskej stanici, teda by mala čo najmenej obmedzovať inú prácu užívateľa.

Vstupné dáta pochádzajú z rôznych pracovíšť, čo znamená že dáta nie sú zapísane jednotným spôsobom. Môžu sa líšiť fyzikálnou jednotkou (napr. pre teplotu v stupňoch Celzia, Kelvina alebo Fahrenheita), ktorá bola pri ich meraní použitá aj formou zápisu. Kvôli úspore miesta sú často reálne čísla prevádzané na menšie celočíselné typy, vhodnými operáciami, ktoré zachovávajú žiadanú presnosť. Dalej môžu byť dáta definované ako číselná rada hodnôt pre konkrétný bod na Zemi, alebo sa môže jednať o mriežku, ktorá popisuje počasie na určitom území.

Operácie, ktoré sú nad jednotlivými dátami vykonávané sú menej náročné, typicky O(n). Väčšinou sa jedná o štatistické spracovanie vstupu (nájdenie rastúcich podpostupností, určenie periódy oscilácie, . . .), avšak aj tu musí byť aplikácia dosť flexibilná, pretože dáta rozdielnych veličín budú spracovávané odlišným spôsobom. Užívateľ teda musí mať možnosť na základe charakteru dát zvoliť odlišnú vetvu výpočtu.

Výsledná aplikácia teda musí užívateľovi umožnovať vytvorenie jednotlivých výpočetných častí a podľa tvaru/charakteru dát umožnovať zmenu priebehu výpočtu čiže použitie resp. preskočenie jednotlivých výpočetných

častí. Musí tiež umožnovať tvorbu ľubovoľného počtu užívateľských premenných rôznych typov a ich prenos medzi jednotlivými výpočetnými časťami aplikácie.

2.2 Programovací jazyk

S ohľadom na prenositeľnosť prostredia PLAS a výsledných aplikácií som si za programovací jazyk vybral Java (konkrétna verzia 1.6).

Prostredie JRE *(Java Runtime Environment)* je momentálne možné nasadiť na väčšinu dostupných platforiem, na ktorých dokáže plne využiť výhody multi-procesorových a multi-jadrových počítačov, čím umožňuje užívateľovi rýchle spracovanie požiadavkov (aj pri zvýšenej miere paralelizmu, kedy je časť aplikácie spustená súčasne viackrát v rôznych vláknach).

Taktiež poskytuje všetky potrebné nástroje pre tvorbu aplikácií v prostredí PLAS už v štandardnej inštalácii bez nutnosti pridávania softwaru tretích strán.

2.3 Štruktúra prostredia

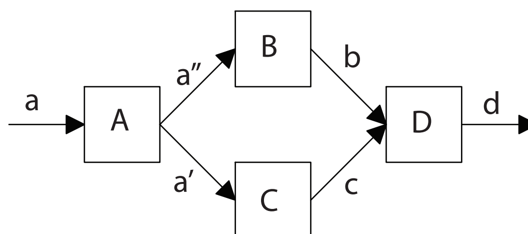
Hlavným problémom pri návrhu štruktúry programu je jednoduché a najmä rozširovateľné definovanie výpočetnej časti aplikácie. Teda programátor aplikácie musí byť schopný bez rozsiahlejších úprav zdrojového kódu pridať resp. odobrať časti výpočtu. Nejjednoduchším a zrejme aj najpraktickejším riešením je použitie samostatných objektov, ktoré sú dynamicky načítané pri štarte aplikácie tj. zásuvných modulov (*pluginov*). Jednoduchým pridaním resp. zmazaním pluginu môže užívateľ pridať do výslednej aplikácie novú funkcionalitu, resp. odobrať prebytočné časti.

Ďaľšou výhodou zásuvných modulov, okrem dynamického rozširovania aplikácie, je vytvorenie spoločného základu, ktorý bude obsahovať logiku potrebnú k inicializácii časti výpočtu a predávanie vstupných/výstupných dát. Vzhľadom k výberu objektového programovacieho jazyka, je spoločná časť implemetnovaná v abstraktnej triede, ktorú užívateľské pluginy rozširujú. Užívateľ sa tak môže pri tvorbe aplikácie sústrediť výhradne na výpočetnú časť. Spoločná časť môže taktiež obsahovať systém front, v ktorých sú uložené jednotlivé požiadavky. Uzol teda nemusí po spracovaní dát odosielať požiadavok na nové dáta svojím predchodcom, ale vyberá dáta z fronty čakajúcich požiadavkov. Výpočetná časť uzlu tak môže byť spustená vo viacerých samostatných inštanciách, bez navyšovania záťaže siete spôsobenej požiadavkami na nové dáta.

Oproti monolytickým programom má systém pluginov zrejmú nevýhodu. Užívateľ musí vytvoriť konfiguráciu aplikácie, ktorá určuje v akom poradí budú operácie obsiahnuté v jednotlivých pluginoch na vstupné dáta aplikované. Pri inicializácii musia byť načítané a overené užívateľom vytvorené pravidlá a v prostredí musí existovať kontrolná časť/modul, ktorý toto poradie vykonávania operácií zachová tj. zaručí, že dáta budú doručené správnemu následníkovi.

Toto spomalenie spôsobené kontrolným modulom je však vyvážená pozitívami, ktoré jeho existencia prináša. Medzi tieto výhody patri napr.:

- **ovládanie priebehu výpočtu** - pravidlá, ktorými sú pluginy definované môžeme rozšíriť aby obsahovali nielen následníka, ale aj stav za akého majú byť dáta následníkovi odoslané. Na adresáciu nasledujúceho pluginu teda môžeme použiť dvojicu `[stav, následník]`. Užívateľ tak môže definovať viacero pravidiel, pričom použité pravidlo záleží od nastaveného stavu.
  Na nasledujúcom obrázku je príklad takejto zmeny toku. Pri výstupe z časti `A` sú za stavu a" dáta odoslané do pluginu `B` a do `C` v stave a'.



Obr. 2.1: Zmena výpočtu, zmenou stavu

- **rozšírenie na distribuovanú aplikáciu** - po rozšírení modulu, ktorý medzi jednotlivými pluginmi dáta preposiela o posielanie/prijímanie správ skrz sieťové rozhranie je možné previesť aplikáciu na distribuovanú. Samozrejme rozloženie aplikácie na viacero sieťových uzlov prináša ďalšie problémy ako napr. výpadok sieťového uzlu, detekovanie ukončenia aplikácie, fragmentovanie siete ...

2.4 Prenos užívateľských dát

Aby bolo zaručené čo najmenšie obmedzovanie užívateľa pri tvorbe aplikácií, je potrebné umožnovať prenos nielen základných typov premenných (Integer, String, ...), ale aj užívateľom definovaných typov. S ohľadom na možnosti,

ktoré poskytuje programovací jazyk JAVA som si vybral vstavanú metódu serializácie/deserializácie objektov. Aj keď sa nejedná o najefektívnejší spôsob, umožňuje pohodlné prenášanie vlastných typov cez sieťové rozhranie.

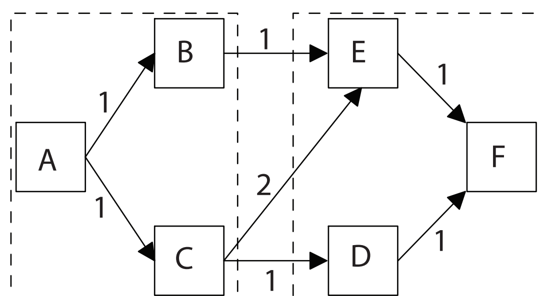
Na ukladanie jednotlivých premenných môžeme použiť hash tabuľku, ktorá je v jazyku vstavaná a poskytuje možnosť vytvorenia variabilného počtu premenných rôznych typov, ku ktorým môže užívateľ pristupovať pomocou vlastného klúča/mena premennej.

Takýto druh prenosu viac menej zodpovedá požiadavkám modelovej aplikácie, kde nie sú medzi komponentami prenášané veľké objemy dát pri zaručenej minimálnej rýchlosti. Ide skôr o menšie variabilné dáta, čo sa týka typov, preto som sa na tento aspekt pri návrhu prenosu dát zameral viac.

2.5 Sieťová komunikácia

K implementácii sieťovej časti prostredia, resp. aplikácie som sa rozdhodol použiť štandardné sieťové rozhranie programovacieho jazyka JAVA. Toto rozhranie už v základnej podobe obsahuje všetky nástroje, ktoré su pre funkčnosť výsledných aplikácií potrebné.

Jednotlivé uzly v prostredí spoločne komunikujú metódou unicast. Pre tvorbu komponentového systému sa síce viac hodí metóda multicast, najmä na odosielanie synchronizačných dát avšak jeho spoľahlivá verzia nie je v základnej inštalácii prostredia JRE dostupná. Teda nie je zaručené, že všetkým klientom bude správa doručená, čo zvyšuje mieru nutnej synchronizácie. Existuje samozrejme možnosť použiť verziu spoľahlivého multicastu JRM [7], je však otázne či by aplikácia plne využila poskytovaných možností a nebol to zbytočný "overhead" pre potreby užívateľa.



Obr. 2.2: Distribuovaný komponentový systém

Výsledné aplikácie vytvorené v prostredí PLAS majú decentralizovaný

charakter. Použitie centrálneho serveru síce uľahčuje synchronizáciu dát v rámci aplikácie, avšak spoľahlivosť programu je viazaná na spoľahlivosť centrálneho serveru. V prípade pádu serveru teda nie je možné vo výpočte dalej pokračovať.

Aby sa čo najviac znížila nadmerná synchronizácia klientov, pokúsil som sa o minimalizovanie sieťovej komunikácie v ktorej sa napr. koniec výpočtu neurčuje spracovaním všetkých dát, ale postupným kaskádovým ukončovaním aplikačných uzlov.

Nedostatkom tejto metódy je obmedzenie sa na acyklické grafy. V prípade cyklických sietí sa vzhľadom k tomu, že všetky vzťahy medzi uzlami aplikácie definuje užívateľ, nedá použiť pevné ukončovanie uzlov. Uzol môže byť ukončený až po zastavení všetkých jeho rodičov. V prípade uzlov ležiacich na kružnici však žiadny nemá informácie o množstve dát v ostatných uzloch a smere ktorým budú dáta odoslané. Riešenie tohoto problému by vyžadovalo pravidelnú synchronizáciu informácií o stave dát uzlov v grafe na základe, ktorého by každý uzol dokázal detekovať svoju potrebnosť. Pre potreby modelovej aplikácie sú však acyklické grafy sietí postačujúce, dáta sú spracovávané len v jednom prechode bez nutnosti ich na výstupe znovu "prehnať" aplikáciou. V prípade, že by užívateľ predsa len cyklické spracovanie požadoval je toto možné emulovať napr. pomocou dočasných súborov, prípadne dočasných tabuliek v databázi.

Výpadok vrcholu

V prípade nedostupnosti sieťového uzlu na ktorom by mal výpočet dalej pokračovať je nutné aplikáciu alebo násilne ukončiť, alebo pokračovať vo výpočte s tým, že úloha vypadnutého uzlu bude nahradená iným počítačom, ktorý dostupný je. Prostredie dovoľuje obe alternatívy, teda užívateľ môže pre každú komponentu určiť či je nutné aby bežala iba na danom uzle (je závislá napr. na súboroch, ktoré tento počítač obsahuje), alebo je možné ju presunúť.

Pri presunutí však tento vrchol "stráca" pamäť tj. dáta, ktoré v dobe pádu spracovával sú stratené. Možnosť bufferovania odosielaných správ som sa rozhodol neimplementovať, pretože pre modelovú aplikáciu jedna z požiadaviek bola beh na bežných užívateľských staniciach. Vzhľadom k fronte použitej k ukladaniu dát čakajúcich na spracovanie a ich odosielaniu do fronty potomka po spracovaní, by musel každý vrchol udržiavať nielen dáta ktoré ešte nespracoval, ale aj dát ktoré už boli odoslané potomkom. Každý uzol by tieto dáta musel ukladať, až do ich úplného spracovania aplikáciou aby v prípade výpadku mohli byť nahradené.

Príklad:
Majme aplikáciu zloženú zo 4 vrcholov A,B,C,D s orientovanými hranami AB, BC, CD. Pre prípad výpadku komponenty C musí vrchol B udržiavať dáta, ktoré ešte neboli ňou spracované. Avšak v prípade výpadku komponent B,C musí nespracované dáta nahradiť vrchol A. Tj. musí mať uložené dáta, ktoré neboli spracované vrcholom B aj tie ktoré neboli spracované vrcholom C.

Takéto bufferovanie však nadmerne zaťažuje pamäťový systém počítačov na ktorých je aplikácia spustená. V prípade užívateľských staníc to môže znamenať prílišné zaťaženie na úkor užívateľa, ktorý ju využíva.

Na prvý pohľad to vyzerá, že takýto spôsob zotavenia sa z výpadku vrcholu je reálne nepoužiteľný, pretože je potrebné aplikáciu znovu spustiť nad rovnakými dátami.. Avšak v prípade veľkých dát v ktorých sú jednotlivé časti spracovávané nezávisle (napr. postupné spracovanie rozdielnych setov dát) to môže vesť k značnej úspore času, pretože je nutné znovu spracovať len časť dát.

Izolované vrcholy

Vo výsledných aplikáciách je nutné kontrolovať existenciu izolovaných uzlov resp. parcelizáciu siete. Teda uzly resp. časti siete, ktoré spolu nemôžu komunikovať, teda nemôžu byť ani ukončené pretože neprijmú ukončovaciu správu. Aby sa takejto situácii predišlo každý uzol pravidelne kontroluje svoje okolie a v prípade, že ostatné uzly (najmä zdrojový uzol, ktorý generuje dáta pre celú aplikáciu) nie sú dostupné svoj beh ukončí.

2.6 Paralelizmus

Nie od každého užívateľa sa dá očakávať vhodná miera optimalizácie či už algoritmu, alebo výber správnej dátovej štruktúry na minimalizovanie času jednotlivých operáci. Preto sa ponúka možnosť využitím vhodnej miery a formy paralelizácie pre zrýchlenie výslednej aplikácie. Ako však vybrať vhodnú mieru paralelizácie?

Ak predpokladáme rozloženie výpočtu do zásuvných modulov, dá sa očakávať istá miera logického rozdelenia výpočtu do viacerých nezávislých skupín. Úlohy v rámci jednej skupiny na sebe závisia a je nutné ich vykonať v určitom definovanom poradí, ale úlohy medzi skupinami sú nezávislé. Teda úlohy z rôznych skupín je možné spustiť simultánne za predpokladu, že úlohy v rámci

jednej skupiny budú vykonávané v definovanom poradí. Zároveň je nutné vybrať úroveň paralelizácie tak aby bola čo najmenej pre cieľového užívateľa zaťažujúca.

Existuje viacero rôznych taxonomií paralelných architektúr. Nás však zaujíma najmä rozdelenie podľa toho ako jednotlivé časti paraleného algoritmu vnímajú užívateľom definované dáta. Z tohto pohľadu ich môžeme rozdeliť na dve hlavné skupiny paralelizácia výpočtu na úrovni dát a úlohy.

Paralelizácia výpočtu na úrovni dát

Paralelizmus je v tomto prístupe dosahovaný delením vstupných dát na menšie časti a aplikovaním rovnakých transformácií vo všetkých výpočetných jednodkách. Na užívateľa však tento prístup paralelizácie kladie nutnosť znalosti algoritmov vyššej úrovne, ktoré umožňujú efektívne spracovanie rozdelených dát a ich následné spojenie, pri zachovaní pôvodnej myšlienky algoritmu resp. spracovania dát.

Paralelizácia výpočtu na úrovni úlohy

Tento prístup paralelizácie sa zaoberá možnosťou spustiť nad vstupnými dátami viac simultánne bežiacich nezávislých úloh. Teda miesto delenia dát na viacero častí, rozdelíme samotnú aplikáciu na viac nezávislých behov. Každý beh má vlastnú kópiu vstupných dát na ktoré aplikuje požadované operácie.

Z hľadiska bežného užívateľa vyzerá prijateľnejšia druhá možnosť pretože rozdelenie problému na čiastočné úlohy je jednoduchšie a intuitívnejšie než hľadanie spôsobu implementácie algoritmu metódou *divide-et-impera*. Najmä pri už existujúcom rozdelení na nezávislé celky/pluginy je možné aplikáciu jednoduchšie previesť do distribuovanej podoby presunutím skupiny úloh na vzdialený počítač a preposielaním potrebných dát.

2.6.1 Synchronizácia dát

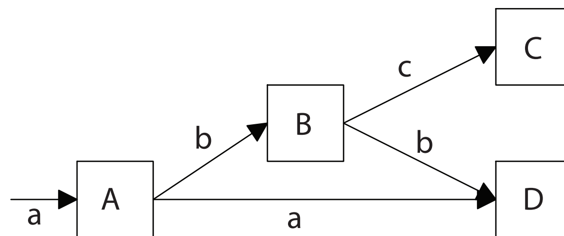
Na to aby mohli časti aplikácie bežať paralelne je však nutné vytvoriť mechanizmus, ktorý dáta rozdelí tak aby každá vetva mohla spracovávať svoju kópiu a znovu ich spojiť v definovanom bode/uzle, tak aby boli zachované premenné definované v jednotlivých vetviach.

Rozdelenie

Rozdelenie dát je relatívne triviálnou záležitosťou. Stačí vytvoriť novú inštanciu prenosového objektu, ktorý bude niesť rovnaké dáta ako pôvodný objekt.

Synchronizácia

Prvým problémom synchronizácie/spojovania rozdelených dát je určenie súvisejúcich dát, resp. určiť ktoré dáta pochádzajú zo spoločného prapredka. Riešenie, ktoré je použité v tejto konkrétnej implementácii je, že si každý objekt udržuje jednoznačný identifikátor svojho prapredka. Tzn. ak majú dva prenosové objekty rovnaký identifkátor prapredka, museli vzniknúť rozdelením jedného objektu a teda je možné ich spojiť.



Obr. 2.3: Delenie a spájanie datagramov

Ďaľším problémom je zistenie počtu existujúcich kópií dát. Ako príklad si predstavme aplikáciu so vzťahmi definovanými podľa obrázku 2.3. V tomto prípade sa musí blok D dozvedieť o možnom rozdelení dát v bloku B. S ohľadom na decentralizovanosť siete a unicastovú komunikáciu medzi jednotlivými uzlami siete nie je príliš vhodné použiť preposielanie synchronizačných informácií. Jednak to môže prílišne zaťažovať sieť, ale takisto pamäť uzlov, ktoré si musia pamätať všetky synchronizačné informácie. Vhodnejší je prístup, v ktorom sú v jednotlivých prenosových objektoch synchronizačné dáta priložené a teda uzol, ktorý s nimi pracuje môže zistiť, že bol objekt rozdelený. Pri jednotlivých objektoch sa ukladá história štiepenia (v príklade popísanom na obrázku sa blok D, z týchto informácií dozvie o existencii verzie objektu c po prijatí objektu b). Ale tiež je nutné ukladať informácie o spojovaní jednotlivých objektov, kvôli zisteniu ešte aktívnych inštancií objektu.

Spojovanie jednotlivých objektov do jednej výslednej verzie, znamená zlúčenie hash tabuliek, ktoré udržujú zoznam užívateľom definovaných premenných. Premenné s rovnakým názvom budú nahradené novšou verziou, preto je dôležité aby užívateľ premenným, ktoré nemajú byť prepísané zvolil unikátny názov.

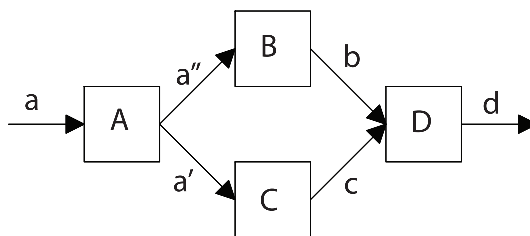
2.7 Ciele práce

Cieľom práce je vytvorenie prostredia (PLAS), ktoré umožní užívateľovi tvorbu distribuovaných aplikácií, ktoré nie sú obmedzené typom úlohy či

spracovávanými dátami. Užívateľ musí mať plnú kontrolu nad jednotlivými výpočetnými časťami programu a tokom dát medzi nimi.

Výsledná aplikácia, vytvorená pomocou prostredia PLAS musí umožnovať:

- dostatočnú voľnosť pri definovaní potrebných premenných.
  Užívateľ nesmie byť limitovaný žiadnou šablónou preddefinovaných premenných.
- paralelné spracovávanie vstupných dát v jednotlivých častiach aplikácie, zreplikovaním časti programu vo vláknach.
- rozdeliť výpočet do paralelných behov (viď Obr. 2.4).



Obr. 2.4: Zmena výpočtu, zmenou stavu

  - rozdelenie dát v časti A
  - uzly B,C paralelne spracovávajú kópie pôvodnych dát (a', a")
  - v užívateľom definovanom bode (v tomto prípade pred vstupom do uzlu D), sa zlúčia a výsledná verzia je spracovaná uzlom D.

Zo sieťového hľadiska musí aplikácia:

- byť ľahko rozšíriteľná na viacero hostiteľských počítačov. Užívateľ musí byť schopný jednoducho rozdeliť zaťaženie počítača presunutím časti aplikácie na iný uzol v sieti.
- pri výpadku sieťového uzlu byť schopná rozhodnúť či je možné pokračovať daľej resp. či sa dá vypadnutý uzol nahradiť alebo je nutné výpočet ukončiť.
- detekovať izolované uzly. Každý uzol musí byť schopný detekovať výpadok svojho pripojenia tzn. stratí spojenie so zvyšnou časťou siete a ukončiť svoj beh korektne.

2.8 Zadanie

Z predchádzajúceho rozboru a cieľov práce je zrejmé, že výsledné prostredie sa odchyľuje od zadania diplomovej práce. Zadanie práce požaduje vytvorenie programovacieho jazyka, kdežto forma prostredia navrhnutého v tejto kapitole je kompontentový systém.

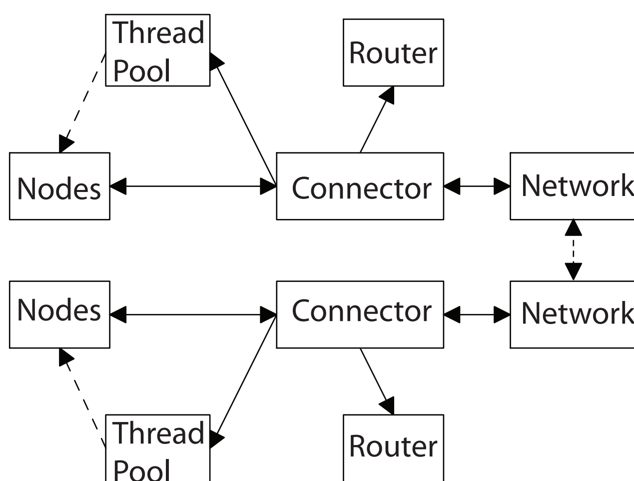
Táto odchýlka resp. zmena zadania vznikla upresnením zámeru vedúcim práce, doktorom Galambošom a zapracovaním návrhov, pripomienok ktoré vyvstali počas konzultácií v priebehu písania práce. Od pôvodného zámeru programovacieho jazyka, som teda upustil v prospech komponentového systému, ktorý lepšie zodpovedá požiadavkám.

Formálnu zmenu zadania sa mi nepodarilo docieliť, pretože sa mi po spracovaní práce nepodarilo doktora Galamboša zastihnúť.

Kapitola 3

Implementácia

3.1 Moduly aplikácie



Obr. 3.1: Moduly aplikácie

Prostredie PLAS pozostáva z jednotlivých prepojených modulov:

- **Connector** - hlavný prepojovací modul. Zbiera spracované datagramy od spustených blokov (či už lokálnych alebo vzdialených skrz modul Network) a preposiela ich ďalej.
- **Router** - načítava a udržuje smerovaciu tabuľku, podľa ktorej sú presmerovávané datagramy medzi užívateľom definovanými blokmi.
- **Network** - sieťová časť aplikácie prostredníctvom ktorej sú odosielané resp. prijímané požiadavky vzdialených uzlov.

- **Thread Pool** - stará sa o spúšťanie a ukončovanie užívateľských komponent.
- **Nodes** - užívateľom definované výpočetné bloky komponentového systému.

3.2 Pomocné dátové štruktúry

3.2.1 DatagramProperty

Táto dátová štruktúra sa používa k ukladaniu užívateľských premenných a časových razítiek ich poslednej modifikácie. Toto razítko je používané pri spojovaní datagramov, aby sa pri zachovali novšie verzie hodnôt. Razítko je aktualizované pri každej editácii premennej.

3.2.2 DatagramProperties

Tento objekt udržuje zoznam premenných (resp zoznam DatagramProperty) a obsahuje funkcie na ich manipuláciu. Premenné sú uložené v asociatívnej HashMape tzn. užívateľ k nim vždy pristupuje pomocou mena danej premennej.

Štruktúra sa predáva každému užívateľom definovanému bloku k spracovaniu. Tým že užívateľ nemá k dispozícii celý objekt Datagram, ale iba objekt obsahujúci premenné je zaručené, že užívateľ nemôže omylom poškodiť objekt Datagram napr. nedopatrením vymazať synchronizačné dáta čím by bol ďalej vo výpočte nepoužiteľný.

Zoznam funkcií na manipuláciu užívateľom definovaných premenných:

- **insertProperty** - vloženie novej premennej
- **replaceProperty** - nahradenie hodnoty existujúcej premennej novou
- **existsProperty** - dotaz na existenciu premennej
- **getProperty** - načítanie aktuálnej hodnoty
- **removeProperty** - odstránenie premennej

Ukážka prístupu k premenným v komponentách, pomocou vyššie zmienených funkcií:

```
/**
 * Metóda workerMethod, v ktorej je definovaná výkonná
 * časť každej komponenty.
 */
protected int workerMethod(final DatagramProperties props)
{
   Integer value = 0, new_value = 0;

   // overenie existencie premennej
   if (props.existsProperty("value"))
   {
      // nacitanie hodnoty premennej
      value = (Integer)props.getProperty("value");
   }
   ...
   // zápis novej premennej
   props.insertProperty("new_value", new_value);

   // nahradenie pôvodnej hodnoty
   props.replaceProperty("value", value);
}
```

3.2.3 Datagram

Dátová štruktúra datagram, definovaná v `plas.datatypes.Datagram`, zabaľuje premenné definované v jednotlivých blokoch (štruktúru `DatagramProperties`) a dáta, ktoré sú potrebné pre ich smerovanie a synchronizáciu vo výslednej aplikácii.

Zoznam premenných

- `datagramCounter` - statické počítadlo datagramov, zaručuje jednoznačnosť premennej `Datagram.hash` v rámci jedného sieťového uzlu.
- `status` - užívateľom definovaný stav, podľa ktorého sa vyberá sada hrán po ktorými budú dáta odoslané.
- `hash` - jednoznačne identifikuje objekt Datagram v priebehu celého výpočtu.
- `parentHash` - používa sa k identifikácii rodiny datagramov. Datagramy, ktoré majú spoločného prapredka patria do jednej skupiny, rodiny.

- `splitHistory` - zoznam hashov datagramov, ktoré boli vytvorené z jedného pôvodného datagramu.
- `mergeHistory` - zoznam hashov datagramov z jednej rodiny, ktoré už boli s týmto datagramom spojené.
- `properties` - zoznam premenných definovaných v jednotlivých blokoch automatu (viď. 3.2.1 resp 3.2.2).

Vytvorenie datagramu

Pri vytváraní nového datagramu je nutné zaručiť jednoznačnosť premennej `Datagram.hash` v priebehu celého výpočtu. V rámci jedného sieťového uzlu túto vlasnosť zaručuje statické počítadlo `Datagram.datagramCounter`. Aby bola jednoznačnosť zaručená v celej sieti je k výslednej hodnote pripojený hash adresy sieťového uzlu na ktorom bol datagram vytvorený.

Ďaľšia možnosť je oznamovanie vytvorenie nového datagramu ostatným klientom v sieti, toto by však pri unicastovej komunikácii zvyšovalo zaťaženie siete. Takisto by takéto riešenie zaťažovalo pamät jednotlivých sieťových uzlov, ktoré by si museli ukladať už použité hashe a pri každom vytvorení novej inštancie kontrolovať kolíziu a zaručiť, že dva uzly nevytvoria datagram s kolidujúcim hashom v jeden moment (napr. viacfázové vytváranie).

Rozdelenie datagramu

Delením vznikne identická kópia pôvodného datagramu, ktorá môže byť samostatne spracovávaná v blokoch aplikácie. Nový datagram obsahuje rovnaké užívateľom definované premenné ako orignál, má rovnaký stav a má toho istého prarodiča ako originál.

Záznam o deleniach datagramov sa ukladá v zozname `splitHistory`. Rozdelovanému datagramu sa do tejto premennej pridá hash novo vytvorenej kópie a novému datagramu je naopak skopírovaný celý pôvodný zoznam. Vďaka zoznamu `splitHistory` nie je nutné rozosielať informácie o rozdelení datagramu, uzol ktorý datagram spracováva môže pomocou tohoto zoznamu zistiť počet štiepení pri príjme datagramu teda v momente kedy ho bude spracovávať. Týmto sa obemdzuje zbytočná komunikácia typu 1:N medzi blokmi.

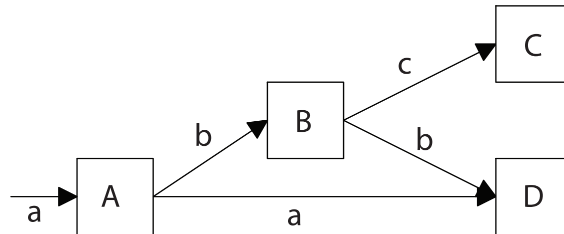
Spojenie datagramu

Spojovanie datagramov z užívateľovho pohladu vlastne znamená zlúčenie premenných oboch datagramov do jedného výsledného zoznamu. Aby sa neprepisovali nové (zmenené) hodnoty premenných starými, pri zjednocovaní sa

vždy zachováva verzia s vyšším časovým razítkom (viď. 3.2.1).

Avšak aby sme si udržali aktuálne synchronizačné údaje je nutné zlúčiť aj zoznamy uložené v premenných `splitHistory` a `mergeHistory`. Pretože oba datagramy prechádzali automatom odlišnou cestou sú ich synchronizačné dáta takisto rozdielne.

Príklad:
Majme datagram b, ktorý vznikol delením datagramu datagramu a. Na výstupe



Obr. 3.2: Delenie a spájanie datagramov

bloku B sa b ďalej rozdelí a vznikne nový datagram c. Pôvodné kópie a, b sú zase zlúčené pred vstupom do bloku D.

Datagram	`splitHistory`	`mergeHistory`
a	SHA1(a), SHA1(b)	SHA1(a)
b	SHA1(a), SHA1(b)	SHA1(b)

Po rozdelení v bloku B:

a	SHA1(a), SHA1(b)	SHA1(a)
b	SHA1(a), SHA1(b), SHA1(c)	SHA1(b)
c	SHA1(a), SHA1(b), SHA1(c)	SHA1(c)

Po spojení pred blokom D:

a	SHA1(a), SHA1(b), SHA1(c)	SHA1(a), SHA1(b)
c	SHA1(a), SHA1(b), SHA1(c)	SHA1(c)

Pretože datagrami a, b prechádzali stavovým automatom rôznou cestou, každý má iné synchronizačné informácie. Preto je nutné pri ich zlievaní zlúčiť aj štruktúry ukladajúce delenie resp. spájanie datagramov.

V opačnom prípade by datagram na výstupe D nemal informáciu o existencii datagramu c a ďalšie zlučovanie by už nemohlo nastať. Výsledný datagram by "preskakoval" synchronizačné body, pretože by podľa informácií v `splitHistory` a `mergeHistory` už boli zjednotené všetky existujúce kópie pre danú rodinu datagramov.

3.2.4 NodeInfo

Každá inštancia tejto dátovej štruktúry zodpovedá jednému užívateľom definovanému bloku výpočtu.

Obsahuje informácie o požadovanom chovaní každej časti. Okrem udržovania zoznamu potomkov aj rodičov, ukladá aj informácie o umiestnení bloku v rámci siete a maximálnom počte simultáne spustených inštancií.

Zoznam premenných

- `name` - užívateľom definovaný názov bloku resp. objektu aplikácie.
- `hostInfo` - objekt typu HostInfo, ktorý udržuje umiestnenie bloku v rámci siete.
- `maxInstances` - maximálny počet simultáne spustených vlákien v ktorých môže inštancia bloku bežať.
- `scheduledStop` - príznak ukončenia bloku. Ak je nastavený na `true`, blok už nemá žiadnych ďaľších predchodcov. Ak teda spracuje všetky naplánované datagramy môže byť ukončený.
- `fixed` - príznak presunuteľnosti bloku pri výpadku sieťového uzlu na ktorom má byť blok spúšťaný. Ak je nastavený na `true`, výpočet nemôže pri výpadku pokračovať a musí byť ukončený predčasne.
- `generator` - ak je tento príznak nastavený na `TRUE` jedná sa o blok, ktorý generuje dáta pre automat.
- `datagramQueue` - počet datagramov, ktoré sú pre tento blok naplánované.
- `parent_nodes` - zoznam rodičovských uzlov bez rozlíšenia stavu hrán, ktorými sú prepojené. Pri zastavení rodiča je tento zo zoznamu odobraný. Teda ak je zoznam prázdny blok nemá rodičov a môže byť ukončený.
- `sync_parent_edges` - zoznam synchronizačných hrán, ktoré smerujú z rodičov do tohto uzlu, zoskupených podľa stavu.

- `child_nodes` - zoznam hrán smerujúcich do potomkov, zoskupených podľa stavu.

Fixné bloky

Užívateľ môže pri návrhu aplikácie, rozhodnúť o presúvateľnosti jednotlivých blokov.

Ak je konkrétny blok z ľubovoľného dôvodu viazaný na konkrétny počítač výpočet sa pri jeho výpadku násilne ukončí na všetkých počítačoch na ktorých je aplikácia spustená. V prípade voľných blokov (vlastnosť `fixed` je nastavená na `false`), je pri výpadku blok presunutý na dostupný sieťový uzol.

Ukončenie bloku

Kvôli snahe o minimalizáciu sieťovej komunikácie typu 1:N sa nerozosielajú správy o vytvorení datagramu resp. kópie. Miesto toho sa na detekovanie aktivity uzlu používa jednoduchý čítač `datagram_queue`. Tento je zvýšený vždy keď je mu od rodičovského bloku predaný datagram, alebo v prípade generátora (blok, ktorý vytvára datagramy aplikáciou spracovávané) v každej iterácii. A znížený po každom spracovaní resp. odoslaní datagramu potomkom.

Výpočetný blok je možné ukončiť až po tom čo spracoval všetky naplánované datagramy (teda čítač `datagram_queue` je rovný nule) a blok už nemá žiadneho aktívneho rodiča, ktorý by mi bol preposlať ďalšie dáta na spracovanie.

Informácia o ukončení bloku je ostatným sieťovým uzlom rozoslaná až po reálnom ukončení. Teda až po tom čo spracuje všetky čakajúce dáta a ostatné sieťové uzly ho môžu odstrániť zo zoznamu aktívnych blokov.

3.2.5 HostInfo

Ukladá informácie o sieťovom uzle na ktorom má byť daný blok výpočtu lokalizovaný. Pre všetky bloky, ktoré sú na rovnakom počítači je tento objekt spoločný, pretože obsahuje tiež inštanciu objektu `Socket` skrz ktorý prebieha komunikácia so vzdialeným klientom.

Zoznam premenných

- `hostAddress` - IP adresa vzdialeného uzlu.
- `socket` - objekt typu `Socket`, ktorým je samotná komunikácia realizovaná.

- `required` - tento príznak určuje postrádateľnoť sieťového uzlu v priebahu výpočtu. Ak je nastavený na `true` uzlu prislúcha fixný (nepresunuteľný) blok výpočtu.
- `reconnected` - ak je tento príznak nastavený na `true`, uzol sa v priebehu výpočtu odpojil a znovu pripojil.

3.2.6 QueueItem

V tejto dátovej štruktúre sú uložené datagrami, ktoré ešte neboli spracované cieľovými blokmi. Zabaľuje informácia nutné k preposlaniu dát správnym potomkom a tiež dáta potrebné v synchronizátore k zjednoteniu správnych verzií.

Zoznam premenných

- `sourceNode` - názov uzlu, ktorý datagram do fronty čakajúcich vložil.
- `data` - objekt datagram.
- `targetNode` - cieľový uzol datagramu.
- `firstEntry` - ak je tento príznak nastavený na `TRUE` je to prvý datagram z jeho rodiny, ktorý je potrebné synchronizovať pred spracovaním v cieľovom uzle.
- `parents` - zoznam rodičov, ktoré sú s daným targetNode spojené synchronizačnou hranou.

Pre jednoduché smerovanie datagramu, kedy žiadna použitá hrana nie je synchronizačná sa používajú len prvé dve premenné. Pretože premenná `data` obsahuje objekt Datagram s definovaným stavom a premenná `sourceNode` typu NodeInfo pre daný stav zoznam potomkov sú tieto postačujúce k odosolaniu dát správnej množine potomkov.

Ak však datagram musí byť pred vstupom do cieľového uzlu synchronizovaný je potrebné mať viac informácií o stave požiadavku. Ako napríklad zoznam rodičov od ktorých by mali byť prijaté daľšie časti výslednej (spojenej) verzie dát.

3.2.7 RelocationItem

Definuje prvok realokačnej fronty. Predpokladajme, že nastal výpadok uzlu v sieti a jedná sa o voľný blok (môže byť presunutý na iný dosiahnuteľný

počítač v sieti). Výpočet musí byť pozdržaný resp. všetky datagrami smerujúce tomuto bloku musia byť pozdržané kým sa nepodarí blok presunuť.

Presunutie vzhľadom k decentralizácii siete prebieha formou voľby tzn. uzol ktorý výpadol detekoval vyberie náhradného vlastníka a rozošle výzvu ostatným klientom. Ak nastane prípad dvoch uzlov, ktoré zistia výpadok je nutné vyriešiť kolíziu výziev. V tejto konkrétnej implementácii závisí váha výzvy od jej časového razítka tzn. staršia (skoršia) výzva je použitá a novšia sa zamietne.

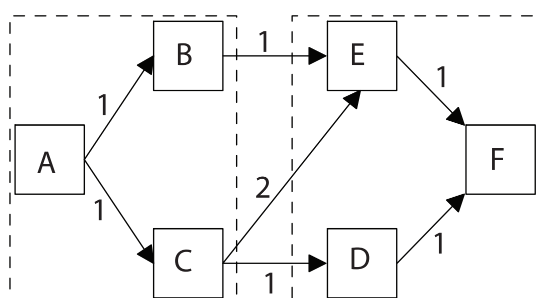
Všetky datagrami odosielané presúvanému bloku výpočtu a stav výzvy sú uložené v fronte realokácií, do momentu kým nie je oznámený výsledok výzvy.

Zoznam premenných

- `datagramList` - zoznam datagramov, ktoré čakajú na spracovanie v presúvanom bloku.
- `confirmation_list` - uzly od ktorých zatiaľ neprišlo potvrdenie presunutia.
- `createTimeStamp` - časové razítko založenia výzvy o presunutie. V prípade kolízie má väčšiu váhu staršia výzva.
- `targetHost` - uzol na ktorý by mal byť blok presunutý.

Kapitola 4

Bloky výpočtu



Obr. 4.1: Distribuovaný komponentový systém

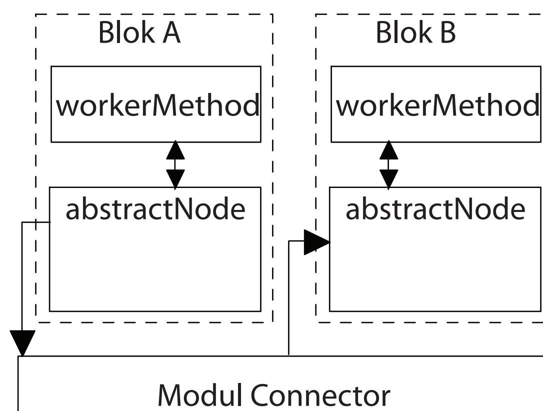
Bloky výpočtu predstavujú výkonnú časť celého komponentového systému. Práve v týchto blokoch dochádza k samotnému spracovaniu vstupných dát. Prostredie PLAS slúži ako prepojovacia vrstva, ktorá na základe definovaných pravidiel vytvára smerovaciu tabuľku a prenáša dáta medzi blokmi či už lokálnymi alebo vzdialenými.

Pre vytvorenie funknčnej aplikácie je nutné aby užívateľ vytvoril jednotlivé bloky/vrcholy s definovaným chovaním a prenosové hrany, ktoré z blokov vytvárajú aplikáciu. Vzhľadom k rozloženiu aplikácie do jednotlivých užívateľom definovaných komponent a ich nezávislosti, je rozprestrenie výpočtu skrz sieť intuitívne jednoducho stačí definovať pre daný blok vzdialený sieťový uzol.

V príklade na obrázku 4.1 sú bloky rozložené na dva sieťové uzly (naznačené prerušovanou čiarou). Pre pridanie ďalšieho sieťového uzlu stačí naň presunúť aspoň jeden z existujúcich blokov, prípadne zadefinovaním nového bloku. Prostredie PLAS teda umožňuje vytvorenie distribuovanej aplikácie aj za predpokladu, že užívateľ nie je príliš zbehlý v programovaní.

Každý blok automatu musí byť objekt v balíčku `plas.nodes.*` (z iného umiestnenia nebude načítaný) a rozšírením abstraktnej triedy `AbstractNode`, prípadne `AbstractGeneratorNode` kde preťažujú metódu `workerMethod`. Tieto triedy implementujú beh každého bloku v samostatnom vlákne, frontu požiadovkov resp. dát čakajúcich na spracovanie.

Taktiež obsahujú rozhranie na preposielanie dát, či už pridanie nových do fronty, alebo preposielanie spracovaných modulu Connector.Teda užívateľ potrebuje implementovať iba funkciu `workerMethod`.



Obr. 4.2: Predávanie požiadavkov užívateľským blokom

4.1 Nastavenia bloku

Chovanie a vlastnosti blokov sú nastaviteľné pomocou anotácií, definovanými v balíčku `plas.nodes.annotations.*`. Vďaka anotáciam je možné eliminovať konfiguračný súbor, čím sa zvyšuje prenositeľnosť aplikácie. Všetky potrebné informácie sú už obsiahnuté priamo v skompilovanej aplikácii a zároveň môže užívateľ tieto vlasnosti prehľadne editovať.

Zoznam možných nastavení pre blok je v nasledujúcom zozname, v zátvorkách sú uvedené východzie hodnoty:

- **generator** (false)
  Príznak, ktorý určuje či daný blok dáta spracováva alebo naopak generuje. Vo výslednej aplikácii môže byť len jeden blok typu generátor.
- **fixed** (true)
  Príznak presúvateľnosti bloku. Ak je nastavený na `true` v prípade výpadku spojenia s týmto blokom musí byť aplikácia ukončená, v opačnom prípade môže byť presunutá na iný uzol.

- **max_instances** (1)
  Maximálny počet vlákien v ktorých je blok automatu spustený. Vybrané bloky môžu spracovávať dáta paralelne vo viacerých vláknach, čím umožňuje užívateľovi zlepšiť paralelizmus výslednej aplikácie.

- **host** (localhost)
  IP adresa, alebo meno počítača (*hostname*) na ktorom má byť blok umiestnený.

- **targets** (prázdny zoznam)
  Zoznam anotácií typu HostAnnotation, ktoré popisujú hrany medzi blokmi.

Príklad nastavenia komponenty pomocou anotácií:

```
@NodeAnnotation
(
max_instances = 1,
targets =
{
  // synchronizačná hrana do komponenty B
  @TargetAnnotation(status = 0, name = "B",
     synchronize=true),

  // hrana do komponenty B
  @TargetAnnotation(status = 1, name = "B")
})
public class A extends AbstractNode
{
   ...
}
```

4.2 Generátor

Každá aplikácia vytvorená pomocou prostredia PLAS musí mať jeden generujúci blok. Tento blok v každom cykle vytvára nový objekt Datagram, ktorý je predaný najprv metóde `workerMethod` a následne skrz modul Connector správnym potomkom, ktorý ďalej tieto dáta spracovávajú.

Tento blok sa líši od ostatných rozširovaním triedy `AbstractGeneratorNode`, ktorej vlákno v každom cykle vytvorí nový datagram a predá ho metóde na spracovanie, miesto spracovávania požiadavkov z fronty.

Z implentačného hľadiska užívateľa je skoro rovnaký ako ostatné bloky automatu. Jediný rozdiel (okrem rozširovania inej základnej triedy a nastavenia príznaku v anotácii) je, že generátor musí byť explicitne zastavený užívateľom volaním metódy `stopNode`. Ostatné bloky sú zastavované automaticky kaskádovým ukončovaním naprieč automatom.

4.3 Voľné a fixné bloky

V prípade distribuovanej aplikácie je potrebné počítať s výpadkami spojenia so sieťovými uzlami a rozhodnúť či má byť aplikácia ukončená, alebo daný uzol nie je pre aplikáciu nutný. PLAS umožnuje užívateľovi kontrolovať toto rozhodovanie pomocou voľných a fixných blokov.

Voľné bloky

Voľný blok môže bežať na ľubovolnom sieťovom uzle. V prípade výpadku spojenia teda môže iný uzol spustiť vlastnú verziu tohoto bloku a všetky požiadavky aplikácie budú presmerované na tento uzol. V tomto prípade blok "stráca pamäť" respektíve jedná sa o novú inštanciu, ktorá nezdieľa žiadne dáta s predchádzjúcou verziou.

Fixné bloky

Naopak fixný blok je časť stavového automatu, ktorá je z neakého dôvodu viazaná na konkrétny sieťový uzol (pripojenie k databázi, ...) a žiadny iný uzol v sieti nemôže tento uzol nahradiť resp. blok spustiť u seba. Prípadne je toto chovanie potrebné ak je v uzle potrebná "pamäť", teda ak si bok ukladá informácie o už spracovaných dátach a v prípade presunu bloku na iný uzol je táto pamät stratená. Aplikácia v prípade výpadku fixného bloku musí byť ukončená predčasne.

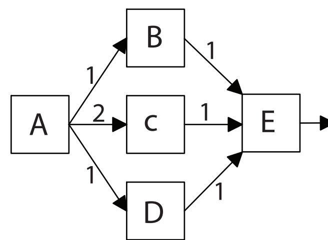
4.4 Hrany

Samozrejmou, veľmi podstatnou časťou komponentového systému sú hrany, ktoré spojujú izolované bloky do funkčnej aplikácie. Rovnako ako pri implementácii
spracovávania dát má užívateľ v prostredí PLAS úplnú kontrolu nad hranami spojujúcimi jednotlivé bloky.

Prostredie PLAS umožňuje vytvárať hrany medzi blokmi s definovaným číselným stavom. Podľa stavu spracovaných dát respektíve návratovej hod-

noty funkcie `workerMethod` je vybraná správna množina hrán po ktorých sú dáta odoslané na ďalšie spracovanie potomkom bloku. V prípade viacerých hrán s rovnakým stavom sú dáta pred odoslaním rozdelené a každý potomok dostane svoju kópiu dát.

Príklad:
V aplikácii definovanom podľa obrázku 4.3 ak dáta odosielané z bloku A majú nastavený stav 1, bloky B a D dostanú svoju kópiu a teda môžu spracovávať dáta zároveň.



Obr. 4.3: Delenie a synchronizácia datagramov

Hrany sú definované pomocou anotácií objektov/blokov v direktíve `target`. Anotácia typu HostAnnotation obsahuje nasledovné možnosti nastavenia:

- **status**
  Stav hrany. Spracované (odchádzajúce) dáta sú odoslané hranami/ou s rovnakým stavom aký majú nastavený.
- **name**
  Názov cieľového bloku. Jedná sa o názov triedy, ktorá implementuje chovanie cieľového bloku.
- **synchronize** (false)
  Príznak synchronizačnej hrany. Ak je nastavený na `true`, dáta sú pred vstupom do cieľového bloku zosynchronizvané.

4.4.1 Synchronizačné hrany

Synchronizačné hrany umožňujú definovanie bodov v ktorých sa zlievajú kópie dát, ktoré boli v niektorom predchádzajúcom bloku rozdelené.

V prostredí PLAS je možné synchronizovať/zjednocovať len datagrami pochádzajúce zo spoločného zdroja, teda musia patriť do rovnakej rodiny (s rovnakou hodnotou v premennej Datagram.parentHash).

Príklad:
Na príklad majme automat definovaný podľa obrázku 4.3 a synchronizačné hrany BE a DE. Dáta, ktoré boli paralelne spracovávané uzlami B a D, sú pred spracovaním v bloku E zjednotené do jednej verzie. Teda blok E bude mať k dispozícii premenné vytvorené v oboch blokoch.

Kapitola 5

Modul Router

Tento modul načítava jednotlivé užívateľom definované bloky umiestnené v balíčku `plas.nodes.*` a z anotácií, ktoré obsahujú vytvára smerovaciu tabuľku. Pomocou tejto tabuľky sú presmerovávané dáta medzi jednotlivými blokmi.

Ďaľsou funkciou modulu je kontrola aktívnych blokov, teda tých ktoré ešte neboli ukončené a vedie do nich cesta z generujúceho bloku. Ak uzol neobsahuje žiadne aktívne bloky môže byť z aplikácie odobraný bez vplyvu na jej funkčnosť.

5.1 Vytvorenie smerovacieho grafu

Pri inicializácii modulu Router sa načítajú prvky smerovacej tabuľky (užívateľom definované bloky) a vytvorenie vzťahov medzi nimi. Prostredie PLAS funguje na princípe decentralizovanej siete v ktorej sú si všetky uzly rovné čo sa týka informácií o stave siete a predávaní požiadavkov, preto je nutné inicializovať tabuľku v každom sieťovom uzle.

Pretože prostredie nepoužíva konfiguračné súbory, je smerovací graf vytváraný načítaním všetkých objektov uložených v balíčku `plas.nodes.*` a anotácií, ktorými je ich chovanie popísané.

5.1.1 Načítanie definovaných blokov

V prípade aplikácie spúšťanej z adresáru, ktorý obsahuje preložené zdrojové kódy udržuje JAVA peknú adresárovú štruktúru kde jednotlivé balíčky sú vlastne adresármi súborového systému a jednotlivé objekty v nich obsiahnuté zase súbormi. Teda v prípade načítavania balíčku `plas.nodes.*` stačí prejsť obsah adresáru plas/nodes/ a načítať názvy jednotlivých súborov

(obektov). Pomocou objektu typu ClassLoader a názvu súboru môžeme pre každý blok načítať JAVA triedu, ktorá mimo iné obsahuje aj užívateľom definované anotácie.

JAVA umožňuje spúštanie výsledných aplikácií nielen z adresáru, ktorý obsahuje všetky preložené zdrojové kódy ale aj zo súboru typu JAR. Tento súbor obsahuje preložené triedy a metadáta, ktoré aplikáciu popisujú v zkomprimovanej podobe. V tomto prípade je aplikácia prenositeľnejšia, ale nie je možné prechádzať adresárovú štruktúru aplikácie a tým načítať obsah balíčku `plas.nodes.*`.

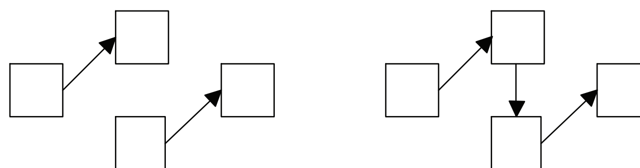
Existujú dve možnosti ako v tomto prípade získať potrebný zoznam blokov. Prechádzanie súboru JAR *on-the-fly* teda za behu aplikácie, alebo vytvorenie modifikovaného Manifest súboru. Pre prostredie PLAS som zvolil metódu upraveného Manifest súboru, ktorý obsahuje direktívu `Node-Classes` v tvare:

```
Node-Classes: AbstractGeneratorNode AbstractNode Blok1 Blok2 ...
```

Pri vytváraní JAR súboru je teda nutné buď ručne editovať súbor Manifest, alebo použiť priložený konfiguračný súbor pre kompilačný systém ant (build.xml), ktorý túto direktívu vytvorí automaticky pri preklade.

5.1.2 Vytvorenie grafu

K vytvoreniu smerovacieho grafu resp. tabuľky je nutné pre každý definovaný blok načítať anotácie, ktoré popisujú jeho chovanie resp. prepojenie s ostatnými blokmi výpočtu. Programovací jazyk JAVA umožňuje či už použitím východzieho alebo vlastného objektu typu ClassLoader načítať triedu prináležiacu jej názvu.



Obr. 5.1: Spojovanie podgrafov

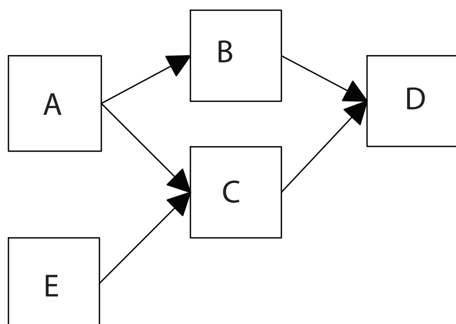
Pretože nie je možné zo zoznamu užívateľských blokov, bez jeho prechádzania zistiť počiatočný (generujúci) blok, je graf vytváraný postupne z podgrafov jednotlivých blokov a ich potomkov (viď. Obr. 5.1). V tomto postupe je pre každý blok vytvorená inštancia objektu NodeInfo a prechádzaním jeho potomkov sú tieto pripojené k rodičovi. Postupným prejdením všetkých blokov je vytvorený výsledný graf.

Poznámka:
Graf musí byť na všetkých sieťových uzlov vytvorený rovnakým spôsobom, resp. všetky zoznamy rodičov, potomkov musia byť rovnaké. Inak by mohol nastať prípad kedy pre jeden graf zapísaný rôznymi spôsobmi bude vygenerovaný rozdielny SHA-1 hash. Tento hash slúži k porovnaniu verzií smerovacieho grafu pri inicializácii sieťového spojenia medzi uzlami, teda aj keď s rovnakým grafom by sa klienti nedokázali spojiť.

5.1.3 Kontrola grafu

Graf je po vytvorení kontrolovaný na prázdnosť a existenciu generujúceho bloku (v každom smerovacom grafe aplikácie vytvorenej pomocou prostredia PLAS musí byť práve jeden generátor) tieto informácie sú zaznamenávané už pri vytváraní samotného grafu.

Ďalšou kontrolou je acyklickosť grafu. Na overenie je použitý prechod grafom do hĺbky z generujúceho bloku a ofarbovanie vrcholov. V rámci tejto kontroly je zisťovaná dostupnosť jednotlivých blokov z generátora. Nedostupné bloky a ich hrany sú z výsledného grafu odstránené, ich prítomnosť v priebehu aplikácie by mohla zablokovať výpočet programu.



Obr. 5.2: Smerovací graf s nedostupným vrcholom

Napr. majme graf definovaný podľa obrázku 5.2 v ktorom je z generujúceho bloku A nedostupný vrchol E. Prepdokladajme ďalej sychronizačné hrany AC, EC. Synchronizátor bloku C bude vždy čakať aj na dáta z vrcholov A,E. Blok E však žiadne dáta nespracováva a teda dáta čakajú v bloku C zbytočne.

Nasleduje kontrola kľúčového slova "localhost". V prípade aplikácie rozloženej na viacerých uzloch siete, je toto označenie hostiteľa mätúce. Teda v prípade distribuovanej aplikácie nie je možné používať spätnú slučku (*loopback*), ale pre všetky sieťové uzly musí byť použitá platná IP adresa resp. platný názov

počítaču. V prípade, že všetky užívateľské bloky majú definovanú anotáciu `host` ako "localhost", resp. používajú východziu aplikácia je nedistribuovaná a môže byť spustená na ľubovoľnom počítači.

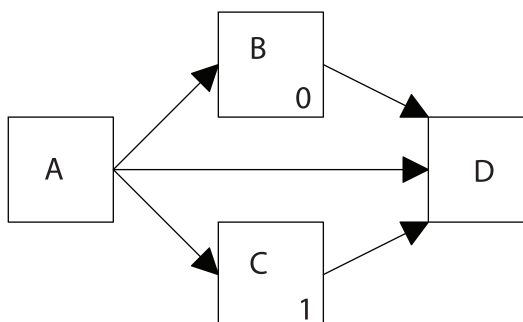
Poslednou fázou kontroly je overenie sieťové uzlu na ktorom je aplikácia spustená. Teda či je vobec počítač na ktorom je aplikácia spúšťaná definovaný ako hostiteľ niektorého užívateľského bloku.

5.2 Ukončenie blokov

Poznámka:
Táto sekcia sa netýka generujúcich blokov. Generátor musí byť vždy zastavený explicitne volaním metódy `stopNode`.

Bloky výpočtu definované užívateľom sú ukončované automaticky. Pri ukončení bloku je tento aj so svojimi hranami odobraný zo smerovacieho grafu. Všetkým potomkom sa teda zníži počet vstupných hrán o jedna. V prípade, že tento počet klesne na nulu resp. bol to posledný rodič daného potomka je tento tiež ukončený resp. naplánovaný na ukončenie v závislosti na dátach čakajúcich na spracovanie. Ak potomok spracoval všetky dáta čakjúce na vstupe môže byť ukončený, pretože všetci jeho rodičia už ukončení boli a teda nemôže prijať žiadne ďaľšie dáta. V opačnom prípade je zastavenie bloku odložené na dobu spracovania všetkých požiadavkov. Po ukončení bloku je táto informácia rozoslaná aj ostatným uzlom v sieti, kvôli zachovaniu synchronizovanej verzie smerovacie grafu.

Táto metóda ukončovania blokov je funkčná iba na acyklických grafov. V prípade cyklického grafu by kvôli decentralizovanosti aplikácie, museli jednotlivé bloky oznamovať vytváranie resp. spracovanie každého Datagramu, pretože neexistuje arbiter, ktorý by mohol rozhodnúť či už boli spracované všetky dáta. Čo by však v prípade unicastovej komunikácie a väčšej množiny dát nadmerne zaťažovalo prenosovú kapacitu siete.



Obr. 5.3: Ukončenie užívateľského bloku

Príklad:
Majme smerovací graf aplikácie definovaný podľa obrázku 5.3. Číselné hodnoty v blokoch určujú počet dát čakajúcich na spracovanie v danom bloku. Po ukončení bloku A sú odobrané všetky hrany z neho vedúce. Teda bloky B a C, ktoré už nemajú ďaľsích rodičov môžu byť ukončené. Blok B môže byť priamo ukončený, zatiaľ čo blok C musí pred zastavením spracovať dáta vo svojej vstupnej fronte, teda je len označený príznakom ukončenia. Bloku D stále zostáva jeden aktívny rodič jeho stav sa preto nezmení.

Zo smerovacieho grafu bude teda odobraný vrchol A, kaskádovo aj blok B. Bloky C a D zostanú naďalej aktívne, pričom blok C sa ukončí v momente vyprázdnenia svojej vstupnej fronty.

5.3 Verejné rozhranie

Verejné rozhranie definované v IRouter obsahuje nasledujúce metódy:

- `String` **getHash()**
  Vracia SHA-1 hash smerovacej tabuľky.
  Tento hash je používaný pri rozložení aplikácie na viacero sieťových uzlov k overeniu, že všetky uzle používajú rovnakú verziu smerovacej tabuľky.

  Návratová hodnota: SHA-1 hash dĺžky 20 bytov reprezentovaný znakovým reťazcom.

- `NodeInfo` **getGeneratorNode()**
  Vracia informácie resp. objekt obsahujúci informácie o bloku, ktorý generuje dáta spracovávané v aplikácii.
  Jedná sa o blok, ktorý užívateľ v anotácii označil príznakom generator (`generator=TRUE`).

Návratová hodnota: objekt typu NodeInfo, ktorý obsahuje informácie o generovacom uzle.

- `List<NodeInfo>` **activeNodes()**
  Vracia zoznam aktívnych blokov, teda tie ktoré ešte neboli ukončené a vedie do nich cesta z generujúceho bloku.

  Návratová hodnota: zoznam aktívnych blokov.

- `List<NodeInfo>` **activeNodes(HostInfo host)**
  Vracia zoznam aktívnych blokov, ktoré sú umiestnené na špecifikovanom sieťovom uzle. Bloky v zozname ešte neboli ukončené a vedie do nich cesta z generujúceho bloku. *Parametre:*
  - host - sieťový uzol.

  Návratová hodnota: zoznam aktívnych blokov umiestnených na zadanom sieťovom uzle.

- `boolean` **hasActiveLocalNodes()**
  Slúži na kontrolu aktívnych blokov, ktoré sú spustené na lokálnom počítači.
  Ak boli už všetky bloky príslúchajúce tomuto uzlu zastavené aplikácia už lokálny počítač k výpočtu nepotrebuje.

  Návratová hodnota: `TRUE` ak na lokálnom uzle existujú aktívne bloky.

- `List<NodeInfo>` **removeNode(NodeInfo removedNode)**
  Odstráni blok zo zoznamu aktívnych.
  Ak bol daný blok ukončený je možné ho odobrať zo zonamu aktívnych a všetkým potomkom odobrať hranu vedúcu z tohto uzlu. Ak potomok už nemá ďalšiu vstupnú hranu je možné ho tiež ukončiť resp. naplánovať na ukončenie ak má ešte nespracované požiadavky. Potomok, ktorý môže byť ukončený je vložený do vracaného zoznamu.

  Parametre:
  - removedNode - odoberaný blok.

  Návratová hodnota: zoznam blokov, ktoré môžu byť po ukončení tohoto bloku tiež ukončené (teda nemajú žiadnych ďalších rodičov a spracovali všetky naplánované Datagrami).

- `NodeInfo` **getNode(String name)**
  Vracia objekt popisujúci užívateľom definovaný blok výpočtu, so špecifikovaným názvom.

  Parametre:
  - name - užívateľom definovaný názov bloku.

Návratová hodnota: objekt typu NodeInfo, ktorý obsahuje informácie o danom uzle.

Výnimky:
- RouterException - ak neexistuje uzol s daným názvom.

- `Map<String, HostInfo>` **remoteHosts()**
Vracia zoznam všetkých sieťových uzlov definovaných pre aplikáciu. Tento zoznam je vytvorený z anotácií, ktorými sú jednotlivé bloky výpočtu popísané.

  Návratová hodnota: zoznam objektov popisujúcich sieťové uzly aplikácie.

- `HostInfo` **getHost(String hostAddr)**
Vracia objekt popisujúci sieťový uzol, ktorý prináleží zadanej IP adrese.

  Parametre:
  - hostAddr - IP adresa sieťového uzlu.

  Návratová hodnota: objekt obsahujúci informácie o vzdialenom sieťovom uzle.

- `List<HostInfo>` **parentHost(NodeInfo srcNode, HostInfo ommit)**
Vracia zoznam sieťových uzlov na ktorých sú umiestnené rodičia zadaného bloku. Táto funkcionalita sa využíva pri presúvaní voľného bloku na nový počítač z dôvodu výpadku v sieti.
Parametre:
  - srcNode - blok výpočtu, ktorý je nedostupný resp. je umiestnený na nedostupnom sieťovom uzle.
  - ommit - sieťový uzol na ktorom srcNode je umiestnený. Teda ten na ktorom došlo k výpadku spojenia.

  Návratová hodnota: zoznam rodičovských sieťových uzlov.

Kapitola 6

Modul Network

Tento modul obsahuje sieťovú komunikáciu medzi jednotlivými časťami aplikácie rozloženými na viacerých prepojených počítačoch.

Obsahuje vlákno prijímajuce spojenia (NetworkServerAccepter) od vzdialených uzlov v sieti na špecifikovanom porte a vlákno, ktoré počúva na jednotlivých už existujúcich spojeniach pomocou StreamPoll objektu.

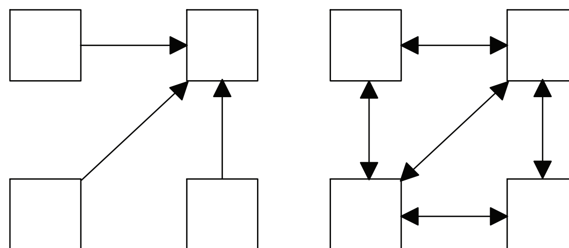
V prípade nedistribuovaej aplikácie tzn. aplikácie ktorá má definovaný len jeden počítač (a teda sieťová časť sa nepoužíva) nedôjde k zvytočnej inicializácii modulu. Distribuovanosť spúšťanej aplikácie je zisťovaná pomocou modulu Router. Pri načítavaní smerovacieho grafu sú načítané aj sieťové uzly na ktorých je aplikácia rozložená a v prípade, že má aplikácia definovaných dva a viac sieťových uzlov jedná sa o distribuovanú aplikáciu a sieťový modul musí byť inicializovaný.

6.1 Inicializácia

V inicializácii je načítaný smerovací graf zo smerovacieho modulu respektíve zoznam sieťových uzlov na ktorých je aplikácia rozložená. Podľa veľkosti tohoto zoznamu je ďaľšia inicializácia modulu preskočená, ak je aplikácia spúšťaná len na jednom (lokálnom) počítači.

V prípade splikácie distribuovanej medzi viacero sieťových uzlov štart modulu pokračuje vytvorením akceptovacieho a počúvacieho vlákna. Akceptovacie vlákno počúva na definovanom porte, definovanom v rozhraní INetwork a prijíma spojenia od vzdialených klientov.

6.2 Spojenie



Obr. 6.1: Tvar siete v aplikácii pri pripojení (vľavo), pri výpočte (vpravo)

Napriek decentralizovanosti prostredia PLAS musí aspoň na dobu pripojenia všetkých sieťových uzlov existovať spoločný server. Na tento server sa pripojujú všetky ostatné potrebné uzly a práve tento server inicializuje výpočet po pripojení všetkých klientov. V prípade úplne decentralizovaného prístupu by musel každý klient postupne kontrolovať dostupnosť ostatných uzlov v sieti a každému z nich odosielať autentifikačné dáta. Taktiež v prípade, že jednotlivé uzly nemajú úplne rovnaký smerovací graf, by sa ťažšie vyberala správna verzia grafu, ktoá má byť použitá. Pri centralizovanom prístupe sa vždy ako smerodatný používa smerovací graf serverového uzlu.

Po inicializácii sieťového modulu, je ďalší priebeh komunikácie resp. chovania uzlov decentralizovaný. Každý uzol sám kontroluje dostupnosť okolitých počítačov, odosielaním krátkych správ a rozhoduje sa pre vhodnú akciu pri výpadku hostiteľa časti aplikácie. To znamená, že z typu siete na obrázku 6.1 vľavo sa v priebehu výpočtu sieť zmení nákres v rovnakom obrázku vpravo, teda kedy komunikuje každý uzol s každým.

Synchronizácia medzi uzlami je vzhľadom ku komunikácii charakteru unicast čo najviac obmedzená, väčšina potrebných dát pre spracovanie je zhromaždená v objekte Datagram, ktorými sa užívateľské dáta medzi uzlami preposielajú. Tým sa znižuje používanie komunikácie 1:N v prípadoch ak môže byť nahradená iným prístupom a prostredie menej plýtva dostupnou sieťovou kapacitou.

Pri spustený si každá časť distribuovanej aplikácie za server vyberá sieťový uzol hosťujúci generujúci blok, ku ktorému sa snaží pripojiť. Vzhľadom existencii práve jediného bloku tohoto typu používa každá časť spoločný server.

6.2.1 Server

V aplikácii figuruje len pri pripojovaní klientov, či už je to pri prvotnom vytvorení siete alebo pri znovupripojení jednotlivých klientov v priebehu výpočtu aplikácie.

Pri štarte aplikácie blokuje jej beh do pripojenia všetkých klientov, vynucuje používanie jednotného smerovacieho grafu vo všetkých počítačoch na ktorých je aplikácia rozložená a filtruje pripojenia jednotlivých klientov na základe ich potrebnosti pri výpočte. Ak sa snaží na server pripojiť počítač, ktorý nie je definovaný v aktuálnom smerovacom grafe ako hostiteľ užívateľského bloku je spojenie s ním ukončené odoslaním chybovej správy `Messages.`
`E_CLIENT_UNKNOWN`. Klient po prijatí tejto správy musí svoj beh ukončiť, pretože používa smerovací graf, ktorý pre danú aplikáciu nie je platný.

V prípade potrebného klienta overuje server SHA-1 hash smerovacej tabuľky, ktorý klient pri prihlasovaná na server odosiela. V prípade nezhody je spojenie ukončené chybovou správou `Messages.E_CLIENT_HASH`, po prijatí ktorej sa musí klient ukončiť. V prípade zhody SHA-1 hashu smerovacieho grafu serveru a klienta je pripojenie akceptované.

Po pripojení posledného klienta server rozošle ostatným uzlom v sieti správu `Messages.NETWORK_STARTED`, ktorou oznamuje vytvorenie spojenia so všetkými potrebnými uzlami. Čím sa ukončí fáza inicializácie spojenia potrebných uzlov a výpočet aplikácie môže začať.

6.2.2 Klient

Prihlasuje sa na server (sieťový uzol obsahujúci generátor) správou `Messages.CLIENT_CONNECT` s parametrom SHA-1 hashu smerovacieho grafu, ktorý načítal jeho modul Router.

Po akceptovaní pripojenia tzn. ak server klienta neodmietol, blokuje beh svojej časti aplikácie do pripojenia ostatných klientov resp. do prijatia správy `Messages.NETWORK_STARTED`. Až po odoslaní tejto správy je na serverovom uzle spustený generujúci blok a teda existujú neaké dáta, ktoré môže lokálne bloky spracovávať. Ak by beh klientskej časti nebol na dobu inicializácie celej aplikácie pozdržaný môžu pri kontrole okolia vznikať nekonzistencie medzi grafmi jednotlivých uzlov (ešte nespustení klienti by mohli byť prehlásený za nedostupných).

6.3 Prijímanie správ

Všetky aktívne spojenia od vzdialených počítačov sú zjednotené v objekte `StreamPoll`, v ktorom sú pravidelne kontrolované neblokujúcimi metódami na prítomnosť čakajúcich správ. Použitím tejto metódy je umožnené prijímanie a spracovávánie správ od rôznych odosieľateľov v jednom vlákne sieťového modulu, miesto vytvárania samostatného vlákna pre každé spojenie.

Pseudokód:

```
timeout=0
zoznam=()
while (timeout<hornáMedz)
{
  for x in aktívne_spojenia
  {
    if x obsahuje dáta then
      pridaj x do zoznamu
  }
  if zoznam nie je prázdny then
    return zoznam

  čakaj(krok)
  timeout += krok
}
return NULL
```

6.4 Ukončenie sieťového uzlu

Ak sieťový uzol zistí, že už nie je v aplikácii potrebný tzn. všetky lokálne spustené bloky už boli ukončené (stav jednotlivých lokálnych blokov je kontrolovaný modulom Router) rozosiela všetkým pripojeným uzlom správu o ukončení svojho behu `Messages.CLIENT_DISCONNECT`. Týmto krokom však neukončí svoj beh resp. pripojenie k ostatným uzlom a ďalej prijíma správy z aplikácie.

Táto vlastnosť umožňuje užívateľovi na danom uzle ďalej monitorovať stav aplikácie a prijímať chyby, ktoré v nej nastali aj po ukončení lokálneho uzlu. Ukončenie aplikácie je blokované až do prijatia tejto správy od všetkých uzlov v sieti.

6.5 Výpadok v sieti

Každý uzol v sieti sám pravidelne kontroluje svoje okolie, stav pripojenia k jednotlivým uzlom pravidelným odosielaním správy typu `Messages.NOOP`. Táto správa slúži iba na detekovania pripojenia ku klientovi, tento na ňu žiadnym spôsobom neodpovedá. Pri detekovaní výpadku sieťového uzlu je táto informácia ďalej predaná modulu Connector, ktorý rozhoduje o potrebnosti daného uzlu pre beh výpočtu.

6.6 Verejné rozhranie

Verejné rozhranie definované v INetwork definuje nasledovné vlastnosti:

- `Integer` **PORT**
  Číslo portu na ktorom klinet čaká na nové spojenia.

- `void` **quit()**
  Ukončenie sieťového modulu.

- `void` **initialize(HostInfo server, String hash)**
  Inicializácia serverovej časti sieťového modulu.
  Spúšťa vlákna zodpovedjúce za akceptovanie prichádzajúcich spojení a počúvanie na vytvorených spojeniach.

  Parametre:
  - server - objekt popisujúci serverový uzol. Jedná sa o uzol, ktorý obsahuje generujúci blok.
  - hash - hash smerovacej tabuľky, ktorý je spočítavaný v module Router. Slúži na overenie smerovacej tabuľky sieťových uzlov.

  Výnimky:
  - NetworkInitException - ak sa nepodarí vytvoriť server socket object na danom porte.

- `void` **setRemoteHosts(Map<String, HostInfo> list)**
  Zoznam vzdialených sieťových uzlov na ktorých je aplikácia rozložená.

  Parametre:
  - list - zoznam sieťových uzlov.

- `void` **connect()**
  Pripojenie na serverový uzol, teda ten ktorý obsahuje generujúci blok.
  V prípade, že uzol obsahuje tento blok (je server), čaká na pripojenie všetkých ostatných klientov.
  Táto funkcie je blokujúca. Serverový uzol čaká na pripojenie všetkých klientov, následne po tom rozošle všetkým správu o začiatku výpočtu

`Messages.E_CLIENT_UNKNOWN`. Klienti blokujú beh programu až do prijatia tejto správy.

Výnimky:
  – NetworkInitException - ak sa nepodarí pripojiť na serverový uzol.

- `void` **checkHosts()**
  Kontrola dostupnosti všetkých vziadelných uzlov.

- `boolean` **activeHosts()**
  Vracia `FALSE` ak sú všetky sieťové uzly odpojené.

  Návratová hodnota: `FALSE` ak sú všetky sieťové uzly odpojené.

- `boolean` **ping(HostInfo target)**
  Odosiela `Messages.NOOP` správu danému vzdialenému uzlu, čím overí výpadok spojenia.

  Parametre:
  – target - cieľový uzol.

  Návratová hodnota: `FALSE` ak je cieľový uzol nedostupný.

- `boolean` **send(HostInfo targetHost, byte message)**
  Odoslanie správy sieťovému uzlu.

  Parametre:
  – targetHost - cieľový uzol.
  – message - odosielaná správa.

  Návratová hodnota: `FALSE` ak sa správu nepodarilo doručiť.

- `boolean` **send(HostInfo targetHost, byte message, Object data)**
  Odoslanie parametrizovanej správy sieťovému uzlu.
  Parameter správy data je odosielaný pomocou ObjectOutputStream.

  Parametre:
  – targetHost - cieľový uzol.
  – message - odosielaná správa.
  – data - paramter správy.

  Návratová hodnota: `FALSE` ak sa správu nepodarilo doručiť.

- `boolean` **send(HostInfo targetHost, byte message, Object[] dataArray)**
  Odoslanie parametrizovanej správy sieťovému uzlu.
  Parametre správy data sú odosielané pomocou ObjectOutputStream.

  Parametre:
  – targetHost - cieľový uzol.

  – message - odosielaná správa.
  – data - paramtre správy.

Návratová hodnota: `FALSE` ak sa správu nepodarilo doručiť.

- `boolean` **sendDatagram(NodeInfo targetNode, NodeInfo sourceNode, Datagram data)**
  Odoslanie datagramu k spracovaniu na vzdialený sieťový uzol.

  Parametre:
  – targetNode - cieľový blok správy.
  – sourceNode - zdrojový blok. Teda blok, ktorý správu spracoval.
  – data - datagram obsahujúci uživateľské dáta.

  Návratová hodnota:

- `void` **broadcast(byte message)**
  Odoslanie správy všetkým sieťovým uzlom.

  Parametre:
  – targetHost - cieľový uzol.
  – message - odosielaná správa.

- `void` **broadcast(byte message, Object data)**
  Odoslanie parametrizovanej správy všetkým sieťovým uzlom.
  Parameter správy data je odosielaný pomocou ObjectOutputStream.

  Parametre:
  – targetHost - cieľový uzol.
  – message - odosielaná správa.
  – data - Parameter správy.

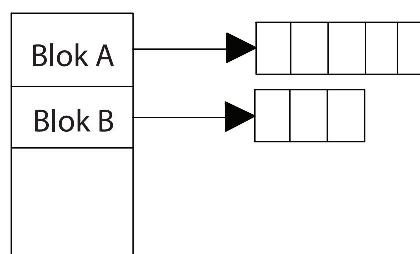
- `void` **broadcast(byte message, Object[] dataArray)**
  Odoslanie parametrizovanej správy všetkým sieťovým uzlom.
  Parametre správy data sú odosielané pomocou ObjectOutputStream.

  Parametre:
  – targetHost - cieľový uzol.
  – message - odosielaná správa.
  – data - paramtre správy.

Kapitola 7

Modul ThreadPool

Tento modul zaisťuje spúšťanie a ukončovanie užívateľom definovaných blokov. Aktívne inštancie blokov sú zoradané v zozname z ktorého je pri požiadavku modulu Connector vyberaná nasledovná tak aby sa využili rovnomerne.



Obr. 7.1: Reprezentácia inštancií blokov

7.1 Vytvorenie inštancie

Pri požiadavku modulu Connector na inštanciu daného bloku, je v tomto module skontrolovaný počet už vytvorených inštancií. V prípade, že je dosiahnuté maximum vlákien, ktoré môžu byť pre daný blok simultáne spustené je vrátená existujúca inštancia zo zoznamu.

V prípade, že ešte blok nebol v predchádzajúcom kroku automatu spustený, alebo ešte nie je vyčerpaná maximálna kvóta je pre daný blok vytvorená nová inštancia, ktorá beží v samostatnom vlákne.

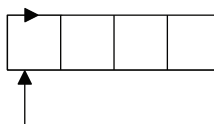
Ak sa prostrediu nepodarí vytvoriť novú inštanciu a žiadna ešte v zozname neexistuje je to chyba ktorú prostredie nemôže maskovať, pretože by aplikácia musela bežať bez užívateľom definovaného bloku. Preto musí modul túto chybu oznámiť volajúcej metóde výnimkou a aplikácia musí byť predčasne

ukončená. Pri zlyhaní pokusu o vytvorenie n-tého simultánneho vlákna sa jedná o zotaviteľnú chybu, pretože je možné použiť už existujúcu inštanciu a daný blok je používaný s obmedzným počtom vlákien.

7.1.1 Rozloženie záťaže

Medzi vláknami

Na rozhodnutie o vhodnom rozložení záťaže medzi jednotlivé vlákna užívateľom definovaného bloku nemá prostredie PLAS dostatok dát. Toto rozhodnutie musí vyplývať z analýzy spracovávania dát v jednotlivých blokoch aplikácie a konkrétnych dát čakajúcich na spracovanie.



Obr. 7.2: Prideľovanie požiadavkov inštanciám

Z tohoto dôvodu používa prostredia PLAS suboptimálne riešenie kde sú inštancie, ktorým je nasledujúca požiadavka odovzdaná cyklicky posúvané. Tým sa rovnomerne rozdelia vstupné dáta čo do počtu, bohužiaľ to neznamená že budú všetky vlákna spracovávať čakajúce požiadavky rovnako rýchlo. Rýchlosť spracovanie jednotlivých dát je závislá na algoritme v bloku obsiahnutom a konkrétnych vstupných dátach.

Medzi sieťovými uzlami

Rozloženie záťaže medzi sieťovými uzlami aplikácie je implicitné. Užívateľ definuje rozloženie jednotlivých komponent v sieti a počas behu aplikacie prostredie PLAS nepresúva komponenty medzi uzlami na základe ich vyťaženosti. Toto obmedzenie prostredia vychádza z dvoch predpokladov.

- Komponenta môže závisieť na prostriedkoch, ktoré su umiestnené prípadne dostupné iba z konkrétnej stanice. Napr. databáza, ktorá ma obmedzený prístup na jednotlivé počítače.

- Nerovnomerné zaťaženie uzlov v sieti je cielené. Napr. čo najmenšie obmedzovanie ostatných užívateľopv ak je aplikácie distribuovaná aj na užívateľské stanice.

7.2 Ukončenie inštancií

V prípade ukončenia užívateľského bloku je nutné ukončiť všetky bežiace vlákna, ktoré boli pre daný blok spustené. V prostredí PLAS nie je možné ukončiť jednotlivé vlákna bloku postupne, po spracovaní dát.

Inštancie blokov sú ukončované vždy až po zastavení bloku, teda v momente kedy už blok nemôže prijať nové dáta na spracovanie. V prípade ukončovanie jednotlivých vlákien, môže nastať prípad kedy vlákno v definovanom časovom úseku nemá naplánované žiadne dáta na spracovanie a svoj beh zastaví. Rodič však po zastavení vlákna prepošle danému bloku nové dáta na spracovanie a je nutné inicializovať vlákno nové miesto znovupoužitia pôvodne existujúceho.

7.3 Verejné rozhranie

Verejné rozhranie modulu definované v IThreadPool obsahuje nasledovné metódy:

- `void` **quit()**
  Zastavenie všetkých vlákien, ktoré boli v module spustené.

- `boolean` **quit(String name)**
  Zastavenie všetkých vlákien, ktoré boli v module spustené, pre konkrétny blok.

  Parametre:
  - name - názov bloku pre ktorý majú byť zrušené všetky inštancie.

  Návratová hodnota: `FALSE` ak žiadne inštancie pre daný blok neboli vytvorené.

- `void` **invokeGenerator()**
  Spustenie generujúceho bloku.

  Výnimky:
  - ThreadPoolException - ak sa nepodarí vytvoriť inštanciu generujúceho bloku.

- `AbstractNode` **get(NodeInfo node)**
  Vracia inštanciu požadovaného bloku, v prípade potreby je vytvorené nové vlákno v ktorom je daná inštancia spustená.

  Parametre:
  - node - informácie o spúšťanom bloku.

Kapitola 8

Modul Connector

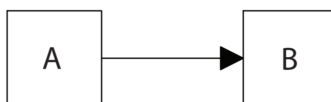
Hlavný modul prostredia PLAS. Prepojuje všetky moduly prostredia a predáva požiadavky medzi jednotlivými užívateľom definovanými blokmi.

8.1 Inicializácia

Pred inicializáciou modulu Connector, sú spustené všetky moduly na ktoré prepojuje. To znamená, že je už načítaný a overený smerovací graf modulom Router a v prípade potreby resp. distribuovanej aplikácie v sieťovom module vytvorené spojenie s potrebnými sieťovými uzlami. Teda aplikácia je overená a môže byť spustená.

Pri spustení sa pomocou modulu ThreadPool spustí generujúci blok na príslušnom hostiteľovi (server z pohľadu sieťového modulu) a vlákno, ktoré odoberá dáta z fronty čakajúcich požiadavkov a po príslušnom spracovaní odovzdáva náležitým potomkom zdrojového bloku.

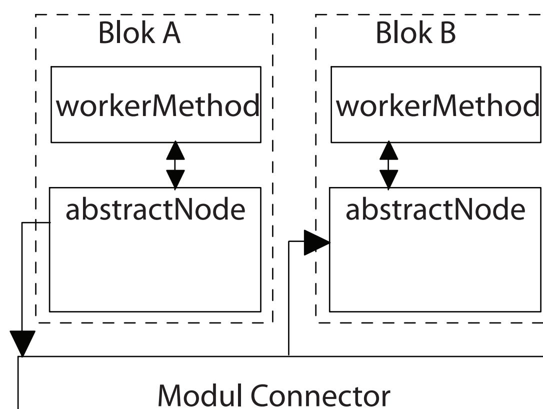
8.2 Preposielanie dát



Obr. 8.1: Preposielanie dát z pohľadu užívateľa

Preposielanie dát medzi komponentami prebieha z pohľadu užívatela priamo. Prostredie PLAS sa chová pre užívateľa transparentne, po priradení definovaného stavu sú dáta odoslané správnym potomkom. Ak priradený stav nezodpovedá žiadnej výstupnej hrane komponenty, dáta sú zahodené. Komunikáciu medzi užívateľským blokom a modulom Connector obstarávajú

metódy abstraktných tried `AbstractNode` resp. `AbstractGeneratorNode` pre generujúci blok (viď Obr. 8.2).



Obr. 8.2: Preposielanie dát z pohľadu prostredia PLAS

8.2.1 Vloženie datagramu do fronty

Po spracovaní dát jednotlivými blokmi sú metódami v abstraktných triedach, ktoré tvoria základ komponenty, dáta pridané do fronty čakajúcich požiadavkov v prepojovacom module volaním metódy `addDatagramToQueue`.

V prípade prijatia datagramu cez sieťový modul je navyše zvýšený čítač dát zdrojového bloku, čím sa zabraňuje jeho zastaveniu kým nie je prijatý požiadavok preposlaný správnemu potomkovi. Čítač je nutné inkrementovať v každom uzle siete, ktorý obsahuje potomka daného zdrojového bloku.

V prípade, že by sa hodnota tohoto počtu dát čakajúcich na spracovanie nemenila (zostala by nulová), uzol obsahujúci potomka by nemal informáciu o počte čakajúcich Datagramov a mohol by blok ukončiť predčasne, pri ukončovaní rodičovského bloku.

8.2.2 Spracovanie datagramu

V každom kroku vlákna prepojovacieho modulu je z fronty vybraný jeden čakajúci datagram. Zo smerovacieho grafu modulu Router je pre daný zdrojový blok a stav dát načítaná množina hrán respektíve potomkov.

Ak je táto množina pre stav dát prázdna tzn. zdrojový blok nemá pre tento stav definovaných následovníkov, jednalo sa o pre tento datagram o koncový stav. Takomto prípade bude zahodený, všetky prostriedky ktorými dis-

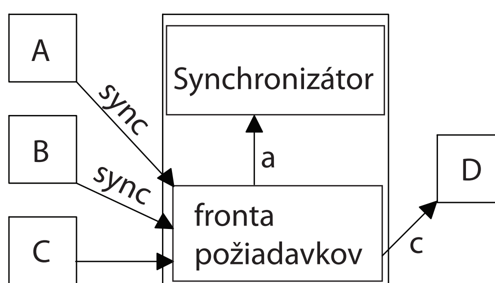
ponoval sú uvoľnené a modul Connector pokračuje spracovaním ďalších dát.

Ak je cieľový blok datagramu umiestnený na lokálnom počítači, je priamo vložený do fronty daného potomka a čítač dát čakajúcich na spracovanie potomka sa zvýši o jedna. Čím sa znemožňuje jeho ukončenie pred spracovaním čakajúcich dát.

Pre cieľové bloky umiestnené na vzdialenom sieťovom uzle sú dáta danému bloku resp. uzlu odoslaná metódou `sendDatagram` sieťového modulu. Ak je uzol obsahujúci potomka nedostupný musí spojovací modul rozhodnúť o vhodnej reakcii na výpadok uzlu. Ak je požadovaný blok fixný je aplikácia predčasne ukončená a správa o výpadku tohoto uzlu je doručená všetkým klientom na ktorých bežia časti aplikácie. V prípade voľných blokov zaháji spojovací modul hlasovanie o presune bloku na náhradný uzol.

Po spracovaní je čítač dát zdrojového bloku znížený o jedna. Ak čítač dosiahne nulu a zároveň blok už nemá žiadneho rodiča môže byť ukončený, pretože spracoval všetky dáta a ďalšie mu už nemôžu byť preposlané.

8.3 Synchronizácia



Obr. 8.3: Synchronizácia datagramov

Ak je hrana vedúca do cieľového bloku označená ako synchronizačná je nutné pred spracovaním zjednotiť všetky kópie dát. Zjednocujú sa všetky dáta, ktoré vstupujú do tohto bloku synchronizačnou hranou.

Zjednocovanie datagramov zabezpečuje objekt `Synchronizer`, ktorý udržuje čakateľov v zozname indexovanom názvom cieľového bloku a rodinou dát (hodnotou premennej `Datagram.parentHash`). Pre každý objekt čakajúci v synchronizátore, je uložený zoznam rodičovských blokov od ktorých ešte neprišiel datagram z danej rodiny, alebo zoznam rodičov na ktoré ešte bloku musí čakať. Po prijatí poslednej verzie dát resp. po prijatí dát od posledného

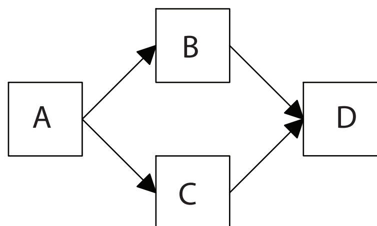
rodiča je výsledná verzia dát predaná do lokálnej fronty bloku.

V nákrese na obrázku 8.3 sú datagrami prijaté z vrcholu C priamo preposielané bloku D. Po prijatí datagramov z blokov A, B teda po synchronizačných hranách musia byť najprv jednotlivé kópie zjednotené. Datagram a (odosieľateľom je blok A) je preposlený do objektu `Synchronizer` kde je zablokovaný do momentu príchodu datagramu b s rovnakou hodnotou premennej `Datagram.parentHash` akú mala kópia a.

V prípade, že synchronizačná hrana vedie medzi blokmi hostovanými na dvoch rôznych sieťových uzloch, synchronizáciu prevádza hostiteľ cieľového bloku. Datagrami od všetkých rodičov sú mu odoslané a musia byť pozdržané v jeho synchronizačnom objekte.

8.3.1 Preskočenie synchronizácie

Za predpokladu, že sa datagrami v aplikácii "nestrácajú" tzn. že koncový stav pre datagram je až blok z ktorého už nevedú hrany a pri odosielaní sa využijú všetky odchádzajúce hrany, sú v synchronizátore vždy spojené zodpovedajúce datagrami v požadovanom počte. Ak sa datagram nemôže stratiť tak musí byť od rodičov preposlaný deťom.

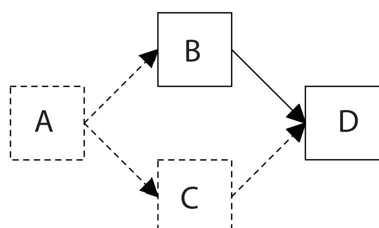


Obr. 8.4: Príklad preskočenia synchronizácie, zadanie

Prostredia PLAS umožňuje aj priebežné koncové stavy tzn. datagram sa môže stratiť v podstate ľubovoľnej komponente. V prípade že nastane táto situácia, teda rodič zahodí datagram, ktorý by mal byť odoslaný po synchronizačnej hrane. V takejto situácii musia dáta v Synchronizátore potomka čakať na ukončenie daného rodiča a až po jeho zastavení môže usúdiť, že dáta boli stratené a odobrať tohto rodiča zo zoznamu blokov od ktorých čaká na vstupné dáta. Tým vlastne "preskakuje" synchronizáciu s dátami.

Príklad:
Majme aplikáciu so smerovacím grafom podľa obrázku 8.4 so synchronizačnými hranami medzi vrcholmi BD a CD.

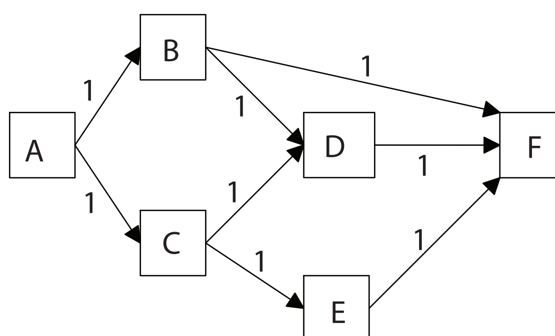
V tomto príklade blok A rozošle lokálnu kópiu svojim potomkom a vrchol C vstupné dáta zahodí. V bloku D sú tým pádom zablokované dáta pochádzajúce od bloku B. Až po ukončení bloku C, ktorý je ukončený kaskádou pri zastavení vrcholu A (stratí jediný rodičovský vrchol a nemá žiadne dáta k spracovaniu), može vrchol D spracovať čakajúce dáta.



Obr. 8.5: Príklad preskočenia synchronizácie, ukončenie blokov A,B

8.3.2 Nejednoznačnosť synchronizácie

V niektorých prípadoch môže zlé rozvrženie hrán medzi komponentami aplikácie viesť k nejednoznačnému výsledku výpočtu.



Obr. 8.6: Príklad grafu s nejednoznačnou synchronizáciou

Takáto situácia nastáva napr. v aplikácii definovanej podľa smerovacieho grafe na obrázku 8.6. V takejto aplikácii sú dáta rozosielané vždy všetkým potomkom a synchronizačné sú len hrany, ktoré vstupujú do bloku F. To však znamená, že blok D vždy spracováva resp. odosiela do vrcholu F dvoje dáta z rovnakej rodiny. Jedny pochádzajú od bloku B a druhé od vrcholu C. Niakým spôsobom sa nedá zaručiť, že do synchronizátoru vrcholu F vstúpi jedna verzia pred druhou s úplnou určitosťou. Poradie v akom dáta prejdú hranou DF zavisí od viacerých faktorov napr. rýchlosť siete za predpokladu, že sú oba posielané zo vzdialeného uzlu siete, časom spracovanie v jednotlivých vrcholoch, naplánovanie vlákien, ...

Riešením takejto situácie je buď prerozdeliť hrany v grafe a vytvoriť nový smerovací graf, ktorého výpočet bude ekvivalentný pôvodnému. Alebo zmena hrán vstupujúcich do bloku D na synchronizačné. Týmto sa dáta spoja už v bloku D a do synchronizátora F bude vstupovať po tejto hrane iba jedna verzia dát.

8.4 Ukončenie bloku

Po zastavení lokálneho bloku je tento odobraný zo smerovacieho grafu a spustená kaskádové ukončovanie po každej odchádzajúcej hrane. Kaskáda ukončuje potomkov, ktorý majú jediného rodiča tzn. jedinú vstupnú hranu po ktorej sme doň vošli a nulový čítač dát čakajúcich na spracovanie. V každom smere je táto kaskáda zastavená ak predchádzajúca podmienka nie je splnená teda blok ešte nemôže byť ukončený. Ak má blok, ktorý kaskádu zastavil iba jediného rodiča (ktorý bol ukončený), ale neprázdnu frontu je naplánovaný na ukončenie nastavením príslušného príznaku `NodeInfo.scheduledStop`.

Notifikácia o ukončení bloku je rozoslaná všetkým hostiteľom aplikácie, aby bola zachovaná jedna verzia smerovacieho grafu a aby mohli ukončovať výpočtové bloky, ktoré majú lokálne spustené.

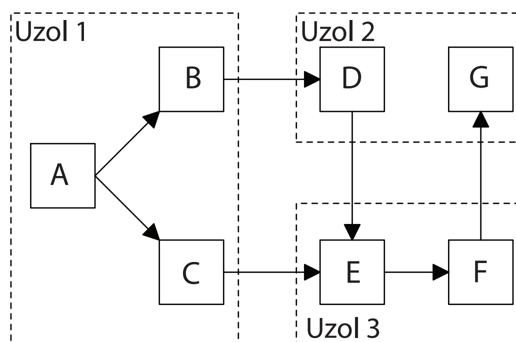
8.5 Presúvanie bloku

Ak sa pri odosielaní vzdialenému uzlu nepodarí modulu `Network` datagram doručiť, aplikácia musí svoj beh zastaviť, alebo chýbajúci blok nahradiť. Táto akcia záleží na voľnosti bloku.

V prípade fixných blokov je rozoslané chybové hlásenie všetkým uzlom v aplikácii a tá je predčasne ukončená. Ak sa jedná o voľný blok, spustí sa proces presunutia na nový uzol. Náhradný uzol je vyberaný zo zoznamu, ktorý tvoria rodičia daného uzlu (hostitelia rodičovských blokov).

8.5.1 Inicializácia presunu

Spojovací modul uzlu, ktorý detekoval nutnosť presunu bloku, vyberá náhradníka za neodpovedjúceho pôvodného hostiteľa. Zoznam náhradníkov obsahuje rodičovské uzly a klientov ktorý sa v priebehu výpočtu aplikácie znovu pripojili a teda majú voľnú kapacitu. Výber nového umiestnenia skončí ak existuje spojenie ku novému kandidátovi (podarí sa odoslať `Messages.NOOP` správu) alebo sa jedná o lokálny počítač. Vďaka druhej podmienke cyklus prechádzania kandidátov určite skončí pri prvom prechode, pretože ak lokálny



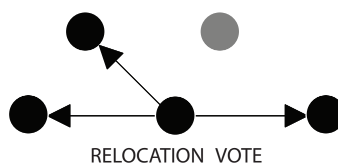
Obr. 8.7: Presun bloku, pôvodné rozloženie automatu

počítač iniciuje relokáciu bloku musí patriť medzi rodičovské uzly.

Po výbere vhodného kandidáta vytvorí záznam RelocationItem vo fronte presúvaných blokov. Tento objekt udržuje časové razítko začiatku presunu a zoznam sieťových uzlov od ktorých zatiaľ neprišlo potvrdenie náhradníka pre vypadnutý uzol. Všetky datagrami, ktoré majú byť odoslané presúvanému bloku sú pozdržané do úplného presunutia bloku.

Ako príklad uvažujme aplikáciu podľa obrázku 8.7 v ktorom v priebehu výpočtu dôjde k výpadku sieťového uzlu číslo 3.
Jeden z blokov C, D (prípadne oba ak výpadok detekujú simultánne) musí v prípade zistenia výpadku iniciovať presunutie vypadnutého bloku na náhradný sieťový uzol.

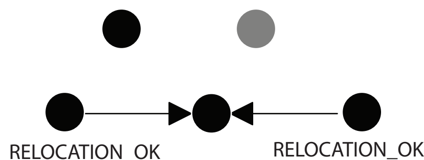
8.5.2 Hlasovanie



Obr. 8.8: Začiatok hlasovania

Vzhľadom na decentralizovanosť siete, nestačí oznámiť výpadok resp. presunu bloku centrálnemu serveru, ktorý by nové umiestnenie tohoto bloku distribuoval ďalej. Je nutné aby sa dohodli všetky uzly na ktorých je aplikácia, pred realizovaním samotného presunu. Inak by sa mohlo stať, že dva uzly vyberú rôznych náhradníkov. Taká situácia môže viesť k nedeterministickému priebehu výpočtu resp. zablokovaním celej aplikácie na synchronizátore presunutého bloku.

Aby podobná situácia nenastala, prostredie PLAS používa systém schvaľovania nového umiestnenia pre daný blok. Rozošle výzvu všetkým dostupným uzlom v sieti a čaká na potvrdenie od každého z nich. Až po schválení všetkými počítačmi v sieti je možné daný blok presunúť.



Obr. 8.9: Potvrdenie výzvy

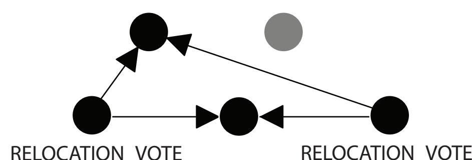
Výzva je rozoslaná formou správy `Messages.RELOCATION_VOTE` a parametrami názov presúvaného bloku, časové razítko začiatku presunu a nového kandidáta pre daný blok všetkým dostupným uzlom v sieti. Na túto výzvu odpovedajú jednotlivý klienti správou `Messages.RELOCATION_OK` ak súhlasia s presunom.

V predchádzajúcom príklade presúvanie bloku v aplikácii podľa obrázku 8.7 predpokladajme, že blok C detekoval výpadok uzlu číslo 3 a teda inicioval voľbu o presunutie bloku E. V tomto prípade čaká na potvrdenie iba od uzlu čislo 2, pretože uzol č.3 je nedostupný, teda nemôže hlasovať.

8.5.3 Kolízia presunu

Pod kolíziou presunu rozumieme situáciu v ktorej sa dva a viac uzlov snaží v približne rovnakom čase presunúť jeden blok výpočtu.

V taktomto prípade pri prijatí výzvy klient skontroluje či sám inicioval výzvu na presun tohoto bloku. Ak áno a zároveň je jeho časové razítko začiatku presunu staršie ako prijaté na výzvu neodpovedá, čím ju zamieta. Z toho vyplýva, že iniciátor výzvy ktorú uzol zamietol nemôže blok za žiadnych okolností presunúť, pretože potrebuje potvrdenie od každého pripojeného klienta.
Naopak ak výzvu pre daný blok neinicioval, prípadne jeho časové razítko je mladšie ako prijaté potvrdzuje výzvu odpoveďou `Messages.RELOCATION_OK`.
V prípade kolízie kedy uzly siete iniciovali výzvu v rovnakom okamžiku, teda časové razítka sú ekvivaletné sa porovná číslo vrcholu získané pomocou topologického triedenia. Je však nutné zaručiť, že vrcholy sú očíslované na všetkých počítačoch v aplikácii rovnako. Toto zaručuje modul Router, ktorý vytvára a následne prechádza graf vždy rovnakým spôsobom na všetkých

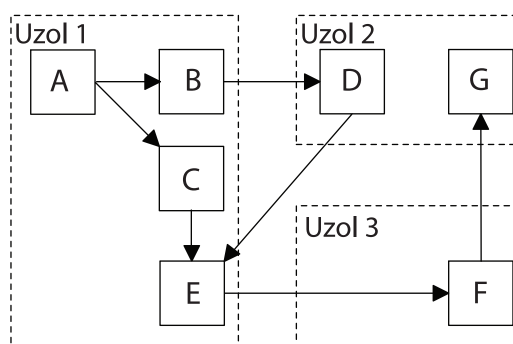


Obr. 8.10: Konflikt výziev

klientoch.

V príklade aplikácie z obrázku 8.7 táto situácia nastane ak bloky C a D detekujú výpadok sieťového uzlu č. 3 resp. nedostupnosť bloku E zároveň. V takom prípade oba uzly iniciujú relokáciu chýbajúceho bloku s vlastným náhradníkom.

8.5.4 Presun bloku



Obr. 8.11: Rozloženie aplikácie po presune bloku E

Po prijatí posledného potvrdenia výzvy presunutia konkrétneho bloku, rozošle pôvodný iniciátor voľby správu `Messages.RELOCATION_DONE` v ktorej ostatným sieťovým uzlom oznámi náhradníka za nedostupný počítač. Každý blok pri prijatí tejto správy upraví svoj smerovací graf a odošle všetky dáta, ktoré čakali na presun bloku novému hostiteľovi.

Presun uzlu resp. následná úprava smerovacieho grafu neovplyvní SHA-1 hash, ktorý modul Router spočítal pri inicializácii. Ak by bol tento hash zmenený pri jednotlivých presunoch uzlov, žiadny vypadnutý uzol by sa už nemohol znovu pripojiť do bežiacej aplikácie.

8.6 Výpadok spojenia

Detekovanie výpadku sieťového uzlu modulom Network je obsluhované spojovacím modulom, ktorý pravidelne kontroluje okolie lokálneho počítaču resp. dostupnosť okolitých počítačov.

Všetky bloky umiestnené na nedostupnom počítači sa v module Connector naplánujú na ukončenie. Tento príznak však neznamená, že sú naozaj ukončené (v priebehu výpočtu môže dôjsť k ich presunutiu na iný uzol) alebo odobrané zo smerovacieho grafu. K samotnému ukončeniu dôjde až v okamihu kedy sú odstránené všetky vstupné hrany tzn. zastavené všetky rodičovské bloky.

Príklad:
V aplikácii na obrázku 8.7 predpokladajme výpadok uzlu číslo 3. Bloky E, F sa v uzloch číslo 1 a 2 naplánujú na ukončenie. Ale za naozaj zastavené sa považujú až po ukončení blokov C, D. Ukončením týchto blokov dôjde k preznačeniu bloku E na vypnutý, pretože stratí všetky rodičovské bloky a kadkádovo aj bloku F.

8.7 Verejné rozhranie

Lokálne rozhranie

IConnectorNode definuje rozhranie medzi užívateľskými blokmi a modulom Connector. Toto rozhranie obsahuje nasledujúce metódy:

- `Datagram` **newDatagram()**
  Vytvorenie nového datagramu. Táto metóda je volaná v každom cykle generujúceho bloku. Pri vytvorení datagramu je upravený čítač dát v generátore.

  Návratová hodnota: nový datagram, ktorý je ďalej spracovávaný v aplikácii.

- `void` **addDatagramToQueue(String sourceNode, Datagram data)**
  Vloží spracovaný datagram do fronty požiadavkov. Modul Connector tieto datagrami po príslušnom spracovaní odovzdáva náležitým potomkom.

  Parametre:
  - sourceNode - názov zdrojového bloku, ktorý dáta odovzdáva.
  - data - objekt obsahujúci užívateľské dáta.

- `void` **nodeStoppedNotification(String nodeName)**
  Notifikácia o zastavení užívateľského bloku. Po prijatí tejto notifikácie je blok odobraný z modulu Router a kaskádovo sú ukončovaný jeho potomkovia.

  Parametre:
  - nodeName - názov ukončeného bloku.

Sieťové rozhranie

Verejné rozhranie definované v IConnectorNetwork, ktoré definuje komunikáciu medzi sieťovým modulom a modulom Connector obsahuje nasledujúce metódy:

- `void` **receivedDatagram(String sourceNode, Datagram data)**
  Vloží datagram prijatý modulom Network do fronty čakajúcich požiadavkov. Modul Connector tieto datagrami po príslušnom spracovaní odovzdáva náležitým potomkom.

  Parametre:
  - sourceNode - odosielací blok umiestený na vzdialenom sieťovom uzle.
  - data - prijatý datagram.

- `void` **stopNode(String nodeName)**
  Notifikácia o zastavení bloku aplikácie na vzdialenom počítači. Slúži ako synchronizačná správa na udržanie jednotného tvaru smerovacieho grafu v sieti.

  Parametre:
  - nodeName - názov ukončeného bloku.

- `void` **forceQuit(String errorMessage)**
  Vynúti násilné ukončenie aplikácie po prijatí chybovej správy cez sieťový modul.

  Parametre:
  - errorMessage - správa popisujúca dôvod ukončenia aplikácie.

- `void` **forceQuit(int code, String errorMsg)**
  Vynúti násilné ukončenie aplikácie a rozosiela chybovú správu ostatným uzlom v sieti, ktoré sa po jej prijatí musia tiež následne ukončiť.

  Parametre:
  - code - kód chybovej správy
  - errorMessage - text správy popisujúci dôvod násilného ukončenia aplikácie.

- `void` **deadHost(HostInfo host)**
  Notifikácia zo sieťového modulu pri detekovaní výpadku uzlu v sieti.

  Parametre:
  – host - neodpovedajúci uzol siete.

- `void` **relocateNodeChallenge(HostInfo caller, String nodeName, Long timestamp, String newHostAddr)**
  Notifikácia spojovacieho modulu modulom Network, po prijatí výzvy na presun voľnej komponenty.

  Parametre:
  – caller - objekt popisujúci uzol, ktorý výzvu zaslal.
  – nodeName - názov presúvaného bloku.
  – timestamp - časové razítko začiatku presunu.
  – newHostAddr - objekt popisujúci náhradné umiestnenie presúvaného bloku.

- `void` **relocateNodeConfirm(HostInfo caller, String nodeName)**
  Schválenie presunu vypadnutého uzlu na náhradný uzol siete. Volaním tejto metódy informuje sieťový modul o prijatí `Messages.RELOCATION_OK` správy.

  Parametre:
  – caller - odosieľatel schválania.
  – nodeName - názov bloku, ktorého presun odosieľatel schvaľoval.

- `void` **relocateNodeDone(String nodeName, String hostAddr)**
  Upozorňuje spojovací modul na príjem správy `Messages.RELOCATION_DONE`, ktorou sa potvrdzuje presun bloku a presúvaný blok má nové umiestnenie.

  Parametre:
  – nodeName - názov presunutého bloku.
  – hostAddr - náhradné umiestnenie pre vypadnutý blok.

- `void` **updateRouteTable(String nodeName, String hostAddr)**
  Informácia o úprava smerovacieho grafu. V prípade znovupripojenia sieťového uzlu server odosiela úpravy smerovacieho grafu (presun bloku), volaním tejto metódy sa zosynchronizuje lokálny smerovací graf.

  Parametre:
  – nodeName - blok výpočtu
  – hostAddr - nové umiestnenie bloku.

Kapitola 9

Záver

Na začiatku tejto práce vytýčený cieľ vytvoriť prostredie pre písanie distribuovaných aplikácií, sa podarilo naplniť. Pomocou prostredia PLAS je možné vytvoriť komponentový systém, ktorý môže byť rozložený medzi viacero sieťových uzlov prípadne bežať aj na jednom počítači.

Prostredie umožňuje užívateľovi definovať jednotlivé bloky výpočtu, ktoré sú vo výslednej aplikácii komponentami systému a jednotlivé hrany medzi nimi, ktorými sú dáta medzi komponentami preposielané. Prostredie sa voči týmto častiam chová ako transparetná prenosová a synchronizačná vrstva, ktorá po spracovaní dát v užívateľom definovanom bloku výpočtu odosiela upravené dáta ďalším častiam aplikácie na základe užívateľom definovaného stavu a pravidiel.

PLAS taktiež sprístupňuje užívateľovi výhody paralelného spracovávania vstupných dát, či už sa jedná o simultánne spúštanie jednotlivých blokov vo viacerých inštanciách. Alebo paralelnými behmi vo výslednej aplikácii pri ktorých jednotlivé komponenty spracovávajú vlastné kópie dát, opäť zjednocované v užívateľom definovaných bodoch.

Pretvorenie aplikácie na distribuovanú resp. rozloženie aplikácie na viacero sieťových uzlov je umožnené zadefinovaním umiestnenia bloku na sieťový uzol v anotácii objektu.

Na vytvorenie funkčnej aplikácie teda stačí užívateľovi vytvoriť objekty reprezentujúce komponenty a nadefinovať ich vlastnosti. Zvyšná funkcionalita výslednej aplikácie je implementovaná v prostredí PLAS. Záverečné hodnotenie jednoduchosti a intuitívnosti tvorby aplikácií pomocou prostredia PLAS je však na individuálnom posúdení každého užívateľa.

9.1 Výber použitých metód

Výber jednotlivých metód a postupov použitých pri vytváraní prostredia PLAS je veľmi diskutabilný. S prihliadnutím na neznámu povahu cieľovej aplikácie, rozloženia jednotlivých blokov v rámci siete, typu a veľkosti prepojovacej siete však považujem mnou vybrané metódy za optimálne čo sa týka nasaditeľnosti aplikácií respektíve limitovanie užívateľa pri ich tvorbe a nasadzovaní.

V prípade nasadzovania aplikácie do prostredia pre ktoré sú vybrané metódy nevhodné existuje, vzhľadom k modulárnosti jednotlivých častí prostredia, možnosť nahradenia modulu iným, ktorý lepšie vyhovuje konkrétnym požiadavkám.

Literatúra

[1] Tvrdík P. (2006): Paralelní systémy a algoritmy

[2] McColl W. F. (1994): BSP programming

[3] Skillicorn D. B., Hill J. M. D., McColl W. F. (1996): Question and answers about BSP

[4] Kenneth P. B. (1996): Building Secure and Reliable Network Applications

[5] Kenneth P. B. (2005): Reliable Distributed Systems - Technologies, Web Services and Applications

[6] Zave P. (1985): A Distributed Alternative to Finite-State-Machine Specifications.

[7] Rosenzweig P., Kadansky M., and Hanna S. (1998): The Java™Reliable Multicast™Service: A Reliable Multicast Library.

[8] Fenner W. (1997): Internet Group Management Protocol, Version 2, RFC 2236

[9] Comer D. (1997): Internetworking with TCP/IP

Dodatok A

Obsah média

A.1 Zdrojový kód

Na priloženom CD médiu sú v adresári `plas` uložené zdrojové kódy prostredia PLAS a konfiguračný súbor `build.xml` pre zostavovací systém Apache Ant.

A.1.1 Požiadavky

Pre využívanie prostredia resp. výsledných aplikácií užívateľom vytvorených je potrebné funkčné JRE (*Java Runtime Environment*) firmy Sun Microsystems. Prostredie PLAS bolo vyvýjané a testované s konkrétnou verziou JRE 1.6.0.

A.1.2 Kompilovanie

Na skompilovanie výsledných aplikácií sa používa zostavovací systém Apache Ant. Priložený konfiguračný súbor `build.xml` vytvára výsledný súbor typu JAR s upraveným Manifest súborom. V upravenom Manifeste je obsiahnutá direktíva *Node-Classes*, ktorá obsahuje zoznam všetkých užívateľom definovaných objektov.

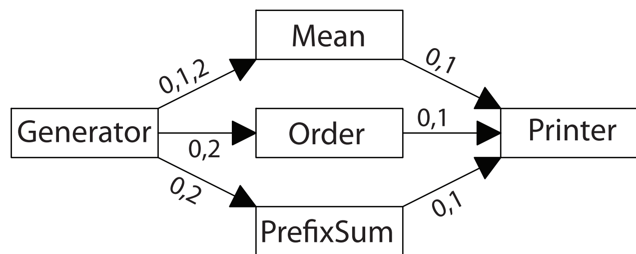
K preloženiu výslednej aplikácie a vytvorenie súboru `plas.jar` dôjde po spustení nasledovného príkazu.

```
ant build
```

A.2 Príklad použitia

V adresári `príklad` na priloženom médiu sú zdrojové kódy a preložená verzia ukážkového príkladu použitia prostredia PLAS. Tento príklad demonštruje rozdelenie úloh na bloky, ich paralelné spracovanie a ovládanie toku dát.

A.2.1 Popis aplikácie



Obr. A.1: Tvar programu

V tomto príklade je implementovaný komponentový systém podľa obrázku A.1 spracovávajúci číselné rady.

Vstupné dáta

Vstupnými dátami pre program je číselná rada reprezentovaná dátovým typom `ArrayList<Integer>`, generovaná v bloku `Generator`.

A.2.2 Bloky aplikácie

Generator – generujúci uzol pre túto aplikáciu. Vytvára trojicu datagramov (vstupných dát), ktoré majú postupne stavy 0, 1 a 2 na ktorých je demonštrované odlišné spracovanie dát toho istého tvaru.

Mean – počíta priemer číselnej rady, ktorý je následne uložený v premennej `mean`.

Order – vytvorí novú premennú `ordered`, ktorá obsajuje zotriedenú vstupnú postupnosť.

PrefixSum – spočítava prefixový súčet pre vstupnú postupnosť. V prefixovom súčte sú ku každému prvku postupnosti pripočítané hodnoty predchádzajúcich prvkov. Výsledný zoznam je uložený v premennej `prefixSum`.

Printer – vypisuje užívateľom definované premenné na štandardný výstup a je koncovým stavom pre všetky dáta. Vstupné hrany so stavom 0 sú synchronizačné, teda kópie dát prichádzajúcich po týchto hranách su pred spracovaním zjednotené.

A.2.3 Výpočet aplikácie

V tejto časti je popísaný výpočet programu pre každý z trojice datagramov odosielaných blokom Generator. Každý datagram v tomto popise obsahuje nasledovnú číselnú postupnosť.

```
list: [60, 48, 29, 47, 15]
```

Prvý datagram

Prvý datagram je z bloku Generator odosielný so stavom 0. Tento datagram je rozoslaný všetkým potomkom a z nich následne do bloku Printer po hrane so stavom 0.

V aplikácii sú týmto datagramom využité všetky bloky a pred vstupom do bloku Printer sú kópie dát z blokov Mean, Order a PrefixSum zjednotené do jednej verzie, ktorá je následne vypísaná na štandardný výstup.

Výstup:

```
Properties:
   id: 0
   ordered: [15, 29, 47, 48, 60]
   mean: 39.8
   list: [60, 48, 29, 47, 15]
   prefixSum: [60, 108, 137, 184, 199]
   synchronize: true
```

Druhý datagram

Tento datagram je z bloku Generator odosielný po hrane stavom 1, teda je odoslaný iba bloku Mean. Z tohto bloku je ďalej odoslaný bloku Printer po synchronizačnej hrane, avšak synchronizácia dát neprebehne. Datagram preskočí synchronizačnú fázu, pretože pre túto rodinu datagramov existuje iba jedna kópia pochádzajúca z bloku Mean.

Výstup:

```
Properties:
   id: 1
   mean: 39.8
   list: [60, 48, 29, 47, 15]
   synchronize: true
```

Tretí datagram

Tretí datagram je z bloku Generátor odosielaný rovnako ako prvý datagram všetkým potomkom.
Z týchto sú však jednotlivé kópie odoslané do bloku Printer po nesynchronizačných hranách so stavom 1. To znamená, že blok Printer vypíše pre túto rodinu datagramov vždy čiastočné výsledky v poradí v akom prídu, pričom obsah jednotlivých kópií je ekvivaletný vhodnej časti výstupu prvého datagramu.

Výstup:

```
Properties:
   id: 2
   mean: 39.8
   list: [60, 48, 29, 47, 15]
   synchronize: false

Properties:
   id: 2
   ordered: [15, 29, 47, 48, 60]
   list: [60, 48, 29, 47, 15]
   synchronize: false

Properties:
   id: 2
   list: [60, 48, 29, 47, 15]
   prefixSum: [60, 108, 137, 184, 199]
   synchronize: false
```