Centralizovaný systém pre výmenu správ medzi paralelnými procesmi

Diplomová práca

Roman Pauer 2007

Centralizovaný systém pre výmenu správ medzi paralelnými procesmi

DIPLOMOVÁ PRÁCA

Roman Pauer

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
KATEDRA INFORMATIKY

Študijný odbor: INFORMATIKA

Vedúci diplomovej práce
Dr. Tomáš Plachetka

BRATISLAVA 2007

Zadanie diplomovej práce

Špecifikácia, návrh a implementácia centralizovaného komunikačného systému. Systém má korektne realizovať formálne definované komunikačné operácie, ako aj dynamické pridávanie a odoberanie procesov (toto zahŕňa odolnosť voči výpadkom procesov).

Čestne vyhlasujem, že som túto diplomovú prácu vypracoval samostatne, s použitím uvedenej literatúry.

Bratislava, 2007

Roman Pauer

Ďakujem svojmu diplomovému vedúcemu Dr. Tomášovi Plachetkovi za cenné rady a pomoc pri písaní diplomovej práce.
Taktiež chcem poďakovať svojim blízkym za ich podporu počas písania tejto práce.

Abstrakt

Práca sa zaoberá špecifikáciou, návrhom a implementáciou centralizovaného systému pre výmenu správ medzi paralelnými procesmi. Inými slovami povedané, je to simulátor distribuovaného systému. Slúži okrem iného na ladenie (debuggovanie) paralelných aplikácií. Práca ďalej obsahuje implementáciu knižnice TPL (Thread Parallel Library) pomocou tohoto systému.

Kľúčové slová: centralizovaný systém, ladenie paralelných programov, paralelný systém, simulátor distribuovaného systému, výmena správ

Obsah

Zoznam obrázkov

1 Úvod

Táto práca sa zaoberá Centralizovaným systémom pre výmenu správ medzi paralelnými procesmi. Obsahuje špecifikáciu, návrh a implementáciu systému, v ktorom paralelné procesy nekomunikujú medzi sebou priamo, ale prostredníctvom centrálneho servera. Ďalej obsahuje implementáciu knižnice TPL [Pla03] pomocou tohoto systému, čo umožňuje iba zmenou knižnice pri linkovaní programu zmeniť spôsob výmeny správ cez centralizovaný systém (namiesto priamej výmeny správ medzi paralelne bežiacimi procesmi).

1.1 Motivácia

Motiváciou vytvorenia centralizovaného systému je portabilný simulátor distribuovaného systému.

Pri výmene správ v distribuovanom systéme je ťažké sledovať čo sa kde a kedy deje. Dôvodom je to, že výmena správ sa odohráva na rôznych miestach a môže sa odohrávať súčasne v tom istom čase. Tu prichádza do hry centralizovaný systém, cez ktorý prechádza všetka komunikácia. V centralizovanom systéme sa výmena správ odohráva na jedinom mieste. Týmto spôsobom je možné výmenu správ serializovať, t.j. zabezpečiť, aby sa v jednom okamihu vykonávala maximálne jedna výmena správ — a to na jedinom mieste.

Výhodou oproti distribuovanému systému je jednoduché monitorovanie všetkých udalostí, ktoré sa týkajú výmeny správ medzi paralelne bežiacimi procesmi. To znamená sledovanie výmeny správ za behu (*monitoring*), a tiež tvorbu záznamu priebehu výmeny správ (*logging*). Toto môže výrazne uľahčiť ladenie (debuggovanie) paralelných aplikácií. Môže to tiež uľahčiť pochopenie komunikačných protokolov neznámej aplikácie. Navyše to môže výrazne uľahčiť odhalenie netriviálnych chýb v zdanlivo korektne bežiacom paralelnom programe.

Medzi ďalšie prínosy centralizovaného systému patrí:

- simulácia rôznych rýchlostí prenosových liniek medzi procesmi (bez potreby cieľového hardvéru)
- hľadanie nedostatkov v algoritme resp. v použitom prostredí (napríklad ak proces čaká na správu a kým ju nedostane, tak nemôže pracovať)
- získavanie štatistických informácií o výmene správ

1.2 Cieľ

Cieľom mojej práce je špecifikovať, navrhnúť a implementovať centralizovaný systém výmeny správ. Využitie takéhoto systému je rôznorodé, v časti 1.1 sú popísané len niektoré príklady použitia.

1.3 Členenie práce

V druhej kapitole je zadefinovaný komunikačný model, ktorý centralizovaný systém používa na výmenu správ.

V tretej kapitole je ukážka programovania pomocou knižnice TPL.

Štvrtá kapitola obsahuje návrh centralizovaného systému vrátane analýzy možností implementácie, ich výhod a nevýhod a mojich rozhodnutí o tom, ktorú možnosť skutočne v implementácii použiť.

V piatej kapitole sa venujem veciam nad rámcom samotnej výmeny správ v centralizovanom systéme. A to konkrétne implementácii knižnice TPL pomocou centralizovaného systému, ďalej záznamu priebehu výmeny správ (ukážky korektných a nekorektných programov a ich príslušných záznamov, analyzátor záznamu) a možnostiam rozšírenia systému.

Šiesta kapitola obsahuje záver práce.

Práca je doplnená o slovník pojmov a v nej použitých skratiek.

1.4 Paralelné systémy a ich ladenie

Najprv spomeniem niekoľko existujúcich paralelných systémov:
PVM [PVM] (Parallel Virtual Machine) je softvérový nástroj, ktorý umožnuje použiť sieť heterogénnych počítačov ako jeden veľký paralelný počítač.
MPI [MPIa], [GLT99], [GLS99] (Message Passing Interface) je aplikačné programové rozhranie (API) pre výmenu správ na paralelných počítačoch, ktoré nezávisí na žiadnom programovacom jazyku. Existuje viacero implementácií, pričom za "generickú" sa považuje MPICH [MPIb].
PARIX [Par94] (PARallel UnIX extensions) je operačný systém používaný na systémoch Parsytec GC. Poskytuje UNIX-ovú funkcionalitu a knižničné rozšírenia pre potreby paralelného systému.
JMS [JMS] (Java Message Service) je aplikačné programové rozhranie (API) pre výmenu správ medzi dvoma alebo viacerými klientami. (Existuje viacero

implementácií.)
Apache ActiveMQ [AMQ] je sprostredkovateľ správ (*Message Broker*) s otvoreným zdrojovým kódom (*open source*), ktorý medzi iným implementuje JMS.
IBM WebSphere Application Server [WSA] je aplikačný server od firmy IBM, ktorý medzi veľa inými vecami takisto implementuje JMS.
Oracle Advanced Queuing [OAQ] je systém výmeny správ firmy Oracle, ktorý je použitý v produktoch Oracle (napr. Oracle Enterprise Service Bus).

Tieto systémy sú blízke môjmu centralizovanému systému, avšak chýba im jednak teoretické zázemie (*background*) a tiež jednoduchá možnosť zmeny komunikačného systému z centralizovaného na distribuovaný a naopak.
Pre MPI existuje štandardné rozhranie [CG99], ktoré môžu poskytovať implementácie MPI a toto rozhranie môžu využívať nástroje (debuggery) aby získavali informácie o MPI programoch (hlavne o frontách správ). Toto rozhranie však neumožňuje zmenu komunikačného systému na centralizovaný. Monitorovanie a debuggovanie je distribuované.
Klientská knižnica PVM umožňuje pri spustení programu spúšťať klientov v debuggeri namiesto normálneho spustenia. Ďalej umožňuje vytvárať záznamy udalostí týkajúcich sa výmeny správ.
Pokiaľ viem, pre PVM ani pre MPI neexistuje centralizovaný simulátor výmeny správ.

Ešte spomeniem niekoľko nástrojov, ktoré podporujú ladenie PVM/MPI programov:
TotalView [Tot] — komerčný debugger od firmy Etnus.
p2d2 [p2d] — portabilný debugger od NASA.
AIMS [AIM] — softvérový nástroj na analýzu výkonu paralelných aplikácií od NASA.

2 Špecifikácia systému

Existuje viacero teoretických komunikačných modelov pre výmenu správ, ako napríklad kanálový model [JGF96]. Tieto ale nemôžu byť priamo namapované na bežne používané komunikačné štruktúry. Konkrétne v kanálovom modeli je kanál neohraničená FIFO (*first-in-first-out*[1]) dátová štruktúra. Ale vedenia v počítačových sietach nemajú žiadnu kapacitu, takže nemôžu uchovávať dáta. Takže kanály nemôžu byť priamo namapované na vedenia a naopak.

Model, ktorý budem používať [Pla06], sa naopak dá priamo namapovať na používanú komunikačnú štruktúru. V tomto modeli prebieha celá výmena pomocou štyroch základných operácií: $CREATE$, $DESTROY$, $SEND$ a $RECV$. $CREATE$ správu vytvorí, $DESTROY$ správu zničí, $SEND$ správu pošle, $RECV$ správu prijme. Je to obdoba (teoretického) modelu používaného v transakčných databázových systémoch, v ktorom sa pre prácu s databázou používajú štyri základné operácie: $READ$, $WRITE$, $INSERT$ a $DELETE$.

Takto definovaný model je výpočtovo ekvivalentný kanálovému modelu a zároveň je výpočtovo silnejší ako MPI (Message Passing Interface). Tento model hovorí o sémantike komunikácie medzi procesmi (o sémantike základných operácií), skutočná realizácia môže byť odlišná.

Mnou použitý model je oproti originálnej definícii [Pla06] rozšírený o operácie $START$, END a o odolnosť voči chybám (*fault-tolerance*). Operácia $START$ je spustenie nového procesu a operácia END je ukončenie procesu. Odolnosť voči chybám v tomto prípade hovorí, čo sa stane keď sa proces ukončí a ešte nie sú vykonané všetky jeho operácie.

Definícia 2.1 (Predloženie základnej operácie pre výmenu správ)
Predloženie základnej operácie označuje akt predania operácie od procesu do systému pre výmenu správ.

Definícia 2.2 (Reprezentácia základných operácií)
Všetky základné operácie pre výmenu správ sú n-tice $[op, x, Y, m, f, s, t]$*, kde:*

op $\in$ *{CREATE, DESTROY, SEND, RECV, START, END};*

x *je identifikátor procesu, ktorý predkladá operáciu;*

Y *je množina identifikátorov procesov;*

m *je správa;*

[1] prvý-dnu-prvý-von

f *je booleovská funkcia definovaná na správach m (filtrovacia funkcia);*

s *je buď referencia na objekt typu semafor, ktorý je prístupný systému pre výmenu správ alebo NULL;*

t *je časová pečiatka predloženia operácie (t.j. čas, kedy bola operácia prečítaná systémom pre výmenu správ);*

Definícia 2.3 (Oblasť procesu)
Oblasť (pôsobnosť) procesu je priestor pamäte, kde sú uložené správy prislúchajúce danému procesu. Ku správe môže pristupovať iba proces, v ktorého oblasti sa správa nachádza. Proces môže vytvárať správy pomocou operácií CREATE alebo START a odstraňovať správy pomocou operácie DESTROY. Systém vytvára, odstraňuje a pristupuje ku správam iba ako je definované v tejto sekcii. Navyše systém ukladá operácie ktoré prečítal do oblasti procesu ktorý predložil danú operáciu. Oblasť procesu x budeme označovať SC(x), oblasť všetkých procesov (zjednotenie oblastí všetkých procesov) označíme SC(∗).

Pre zjednodušenie bude identifikátor m niekedy označovať správu a inokedy ukazovateľ na správu. Z kontextu bude použitie zrejmé.

Definícia 2.4 (Spracovanie predložených operácií)
Systém môže v jednom okamihu buď prečítať alebo vykonať jednu operáciu. System číta iba tie operácie, ktore boli predložené. Každá predložená operácia je systémom prečítaná iba raz. Keď systém prečíta operáciu, tak zaktualizuje jej časovú pečiatku a uloží operáciu v oblasti procesu, ktorý operáciu predložil. V ľubovoľnom čase t môže systém vykonať iba operáciu ktorá je v čase t uložená v SC(∗). Systém môže zdržať vykonanie predloženej operácie (t.j. operácie nemusia byť nutne vykonané v poradí v akom boli predložené). Nesmie však vymeniť poradie operácií SEND a RECV s operáciou START.

Definícia 2.5 (Združené operácie)
Budeme hovoriť, že dve základné operácie $BO_1 = [op_1,x_1,Y_1,m_1,f_1,s_1,t_1]$, $BO_2 = [op_2,x_2,Y_2,m_2,f_2,s_2,t_2]$, *(alebo* $BO_2 = [op_1,x_1,Y_1,m_1,f_1,s_1,t_1]$, $BO_1 = [op_2,x_2,Y_2,m_2,f_2,s_2,t_2]$*) sú združené operácie práve vtedy, keď:*

$$
\begin{gathered}
(op_1 = SEND \wedge op_2 = RECV \wedge x_1 \in Y_2 \wedge x_2 \in Y_2 \wedge f_2(m_1) \wedge \\
(\forall BO_1' = [op_1', x_1', Y_1', m_1', f_1', s_1', t_1'] \in SC(*) : (BO_1' \equiv BO_1 \vee \\
op_1' \neq SEND \vee x_1' \notin Y_2 \vee x_2 \notin Y_1' \vee \neg f_2(m_1') \vee t_1' \geq t_1)) \wedge \\
(\forall BO_2' = [op_2', x_2', Y_2', m_2', f_2', s_2', t_2'] \in SC(*) : (BO_2' \equiv BO_2 \vee \\
op_2' \neq RECV \vee x_1 \notin Y_2' \vee x_2' \notin Y_1 \vee \neg f_2'(m_1) \vee t_2' \geq t_2)))
\end{gathered}
$$

Taktiež budeme hovoriť, že BO_1 *je operácia združená s operáciou* BO_2 *a naopak.*

Neformálne: Operácia posielania BO_1 = [SEND,x_1,Y_1,m_1,f_1,s_1,t_1] je združená s operáciou prijímania BO_2 = [RECV,x_2,Y_2,m_2,f_2,s_2,t_2] práve vtedy, keď množina adresátov Y_1 obsahuje x_2, množina odosielateľov Y_2 obsahuje x_1, filtrovacia funkcia f_2 akceptuje správu m_1 a žiadna z operácií BO_1 a BO_2 nemôže byť nahradená staršou operáciou, ktorá spĺňa uvedené podmienky.

Definícia združených operácií môže byť zoslabená napríklad v prípade, ak nevyžadujeme aby správy posielané procesom inému procesu prichádzali v rovnakom poradí ako boli odoslané. V tomto prípade sú časové pečiatky ignorované a predikát v definícii 2.5 sa zmení na:

$$(op_1 = SEND \wedge op_2 = RECV \wedge x_1 \in Y_2 \wedge x_2 \in Y_2 \wedge f_2(m_1))$$

V ďalšom texte budeme používať takú definíciu, ktorá zachováva poradie (t.j. Definícia 2.5).

Definícia 2.6 (Vykonanie operácie CREATE)
Vykonanie operácie [CREATE,x,Y,m,f,s,t] pozostáva z nasledovných akcií vykonaných v atomickom kroku:

1. *Systém vytvorí novú správu m v SC(x).*
2. *Ak s ≠ NULL tak systém vykoná semaphore_signal(s).*
3. *Systém odstráni túto operáciu z SC(x).*

Definícia 2.7 (Vykonanie operácie DESTROY)
Vykonanie operácie [DESTROY,x,Y,m,f,s,t] pozostáva z nasledovných akcií vykonaných v atomickom kroku:

1. *Systém odstráni správu m z SC(x).*
2. *Ak s ≠ NULL tak systém vykoná semaphore_signal(s).*
3. *Systém odstráni túto operáciu z SC(x).*

Definícia 2.8 (Vykonanie operácie START)
Vykonanie operácie [START,x,Y,m,f,s,t] pozostáva z nasledovných akcií vykonaných v atomickom kroku:

1. *Systém vytvorí nový proces.*
2. *Systém vytvorí novú správu m v SC(x).*

3. *Systém nastaví obsah správy m na hodnotu NEW_ID=x', kde x' je identifikátor nového procesu. (Tento identifikátor je jednoznačný, t.j. odlišný od odentifikátorov všetkých dosiaľ vytvorených procesov.)*

4. *Ak $s \neq NULL$ tak systém vykoná semaphore_signal(s).*

5. *Systém odstráni túto operáciu z SC(x).*

Poznámka: Tu sa využíva skrytý predpoklad, že každý proces pozná svoj identifikátor. Čiže nový proces svoj identifikátor pozná a v správe si jeho identifikátor môže prečítať aj proces, ktorý ho vytvoril (proces, ktorý predložil operáciu START do systému).

Definícia 2.9 (Vykonanie operácie END)
Vykonanie operácie BE=[END,x,Y,m,f,s,t] pozostáva z nasledovných akcií vykonaných v atomickom kroku:

1. *Systém odstráni všetky operácie okrem BE z SC(x).*

2. *Systém odstráni všetky správy z SC(x).*

3. *Ak $s \neq NULL$ tak systém vykoná semaphore_signal(s).*

4. *Systém odstráni BE z SC(x).*

5. *Systém odstráni proces s identifikátorom x.*

Definícia 2.10 (Vykonanie operácie SEND)
Systém môže v čase t_1 vykonať operáciu BS = [SEND,x,Y,m,f,s,t] iba ak je v čase t_1 splnená aspoň jedna z nasledujúcich podmienok:

a) $Y = \emptyset \vee \exists z \in Y$: *v systéme neexistuje proces s identifikátorom z.*

b) *Existuje operácia BR = [RECV,x',Y',m',f',s',t'] $\in SC(*)$ združená s operáciou BS.*

Ak platí podmienka a), tak vykonanie operácie BS pozostáva z nasledovných akcií vykonaných v atomickom kroku:

1. *Systém odstráni správu m z SC(x).*

2. *Ak $s \neq NULL$ tak systém vykoná semaphore_signal(s).*

3. *Systém odstráni BS z SC(x).*

Ak platí podmienka b) a neplatí podmienka a), tak sa operácie BS a BR vykonajú naraz nasledovnými akciami vykonanými v atomickom kroku:

1. *Systém vytvorí novú správu m' v SC(x').*
2. *Systém skopíruje obsah správy m do obsahu správy m'.*
3. *Systém odstráni správu m z SC(x).*
4. *Ak $s' \neq NULL$ tak systém vykoná semaphore_signal(s').*
5. *Ak $s \neq NULL$ tak systém vykoná semaphore_signal(s).*
6. *Systém odstráni BS z SC(x).*
7. *Systém odstráni BR z SC(x').*

Definícia 2.11 (Vykonanie operácie RECV)
Systém môže v čase t_1 vykonať operáciu BR = [RECV,x,Y,m,f,s,t] iba ak je v čase t_1 splnená aspoň jedna z nasledujúcich podmienok:

a) $Y = \emptyset \vee \exists z \in Y$: *v systéme neexistuje proces s identifikátorom z.*

b) *Existuje operácia BS = [SEND,x',Y',m',f',s',t'] $\in$ SC($*$) združená s operáciou BR.*

Ak platí podmienka a), tak vykonanie operácie BR pozostáva z nasledovných akcií vykonaných v atomickom kroku:

1. *Systém vytvorí novú správu m v SC(x).*
2. *Systém nastaví obsah správy m na hodnotu SENDER_NOT_FOUND=z.*
3. *Ak $s \neq NULL$ tak systém vykoná semaphore_signal(s).*
4. *Systém odstráni BR z SC(x).*

Ak platí podmienka b) a neplatí podmienka a), tak sa operácie BS a BR vykonajú naraz nasledovnými akciami vykonanými v atomickom kroku:

1. *Systém vytvorí novú správu m v SC(x).*
2. *Systém skopíruje obsah správy m' do obsahu správy m.*
3. *Systém odstráni správu m' z SC(x').*
4. *Ak $s \neq NULL$ tak systém vykoná semaphore_signal(s).*

5. *Ak $s' \neq NULL$ tak systém vykoná semaphore_ signal(s').*
6. *Systém odstráni BS z SC(x').*
7. *Systém odstráni BR z SC(x).*

Definícia 2.12 (Spracovanie a vykonanie operácií)
Systém raz určite prečíta každú predloženú správu. Systém raz určite vykoná každú prečítanú operáciu typu CREATE, DESTROY, START a END. Ak sa v nejakom čase t vyskytne v SC($*$) pár združených operácií, tak systém raz určite vykoná aspoň jednu z nich.

Posledná časť definície 2.12 je vyjadrená opatrne z dôvodu podpory alternatívnych definícií združených operácií. Napríklad ak nahradíme definíciu 2.5 definíciou ktorá nepoužíva časové pečiatky, tak môže nastať nasledovný prípad. V čase t existujú v SC($*$) združené operácie BR a BS. Zároveň v čase t existuje v SC($*$) operácia BS', ktorá je taktiež združená s operáciou BR (Systém nemusí vykonať pár BR a BS okamžite, takže môže namiesto toho prečítať operáciu BS'. Čas t označuje okamih, kedy boli prečítané operácie BR, BS aj BS', ale žiadna z nich ešte nebola vykonaná). Teda časť "aspoň jednu" v definícii 2.12 umožňuje systému vykonať buď pár operácií BR, BS alebo pár BR, BS'.

3 Programovanie v TPL

TPL [Pla03] (Thread Parallel Library) je portabilná knižnica, ktorá implementuje abstraktný model popísaný v kapitole 2. TPL umožňuje pohodlný štart paralelnej aplikácie na viacerých procesoroch. Všetky funkcie TPL, ktoré sa týkajú komunikácie, sú bezpečné voči vláknam (*thread-safe*). Nasledujúci program v C demonštruje použitie knižnice TPL. Program sa skladá z dvoch klientov. Prvý klient vytvorí správu a pošle ju druhému klientovi. Druhý klient prijme správu od prvého klienta, správu dekóduje a potom zničí.

```
/* filtrovacia funkcia, ktorá akceptuje všetky správy */
int match_all(int sender, int tag, void *message)
{
  return(1);
}

int main(int argc, char **argv)
{
  int my_rank, nr_procs, rank_sender, rank_receiver;
  int sender, tag;
  void *sendbuf, *message;
  char recv_char;

  nr_procs = 2;
  rank_sender = 0;
  rank_receiver = 1;

  /* najprv inicializujeme systém a vytvoríme dvoch klientov */
  /* tu sa použije operácia START */
  tpl_initialize(&nr_procs, &my_rank, &argc, &argv);
  /* ak sa inicializácia nepodarí, */
  /* TPL zničí všetky vytvorené procesy */

  /* každý klient robí inú prácu */
  if (my_rank == rank_sender)
  {
    /* prvý klient najprv vytvorí správu - operácia CREATE */
    tpl_create(&sendbuf);
    /* potom naplní obsah správy */
    tpl_pkchar(sendbuf, "1234567890", 10);
    /* a nakoniec správu pošle druhému klientovi */
```

```
    tpl_send(&rank_receiver, 1, 0, sendbuf);
    /* operácia SEND */
  }
  else if (my_rank == rank_receiver)
  {
    /* druhý klient najprv prijme správu - operácia RECV*/
    tpl_recv(match_all, &sender, &tag, &message);
    /* potom zo správy prečíta prvý znak */
    tpl_upkchar(message, &recv_char, 1);
    /* a nakoniec správu zničí */
    tpl_destroy(message);
    /* operácia DESTROY */
  }

  /* nakoniec systém deinicializujeme a ukončíme program */
  /* tu sa použije operácia END */
  tpl_deinitialize();
  return(0);
}
```

4 Návrh systému

V ďalšom texte bude *klient* označovať jeden z paralelných procesov ktoré si vymieňajú správy, *server* bude označovať centralizovaný server pre výmenu správ a *systém* bude označovať systém pozostávajúci z paralelných procesov a centralizovaného servera pre výmenu správ.

Systém pozostáva z dvoch častí. Jedna časť je samotný server, druhá časť je klientská knižnica, ktorá implementuje funkcie rozhrania, cez ktoré klienti komunikujú (t.j. základné operácie).

Najskôr uvediem možnosti implementácie systému spĺňajúceho špecifikáciu podľa rôznych kritérií, výhody/nevýhody uvedených riešení a dôvody pre môj výber riešenia. Potom popíšem samotný návrh systému.

4.1 Možnosti implementácie

4.1.1 Viazanosť na platformu

Viazanosť na platformu je otázka, či sa sústrediť na beh systému na jednej konkrétnej platforme (t.j. operačnom systéme a/alebo hardvéri), alebo či umožniť/podporovať beh systému na viacerých platformách.

a) Viazanosť na konkrétnu platformu (systém beží iba na jednej platforme):
Výhody:

- Iba výhody konkrétnej platformy oproti ostatným platformám

Nevýhody:

- Užšia možnosť využitia systému

b) Multiplatformovosť (systém beží na viacerých platformách):
Výhody:

- Širšia možnosť využitia systému

Nevýhody:

- Nevyužitie výhod konkrétnej platformy oproti iným platformám

Keďže žiadna platforma neponúka také výhody oproti ostatným platformám, ktoré by prevážili širšiu možnosť využitia systému, tak som zvolil možnosť *b) Multiplatformovosť*.

4.1.2 Dosiahnutie multiplatformovosti

Dosiahnutie multiplatformovosti je otázka ako multiplatformovosť dosiahnuť. Multiplatformovosť môže byť dosiahnutá na úrovni zdrojového kódu (t.j. ten istý zdrojový kód sa dá skompilovať a spustiť na rôznych platformách) alebo na úrovni skompilovaného kódu (t.j. ten istý skompilovaný kód sa dá spustiť na rôznych platformách). Ďalším aspektom multiplatformovosti je otázka či pri behu systému môžu byť klienti a/alebo server na rôznych platformách alebo musia byť na rovnakej platforme.

a) Použitie programovacieho jazyka/prostredia, ktorého vykonávanie nie je závislé na platforme (Java [Java], .NET [NET], ..., t.j. skompilovaný kód nie je kód konkrétnej platformy, ale je to medzikód, ktorý je vykonávaný virtuálnym strojom):
Výhody:

- Možnosti systému nie sú obmedzené možnosťami platformy — program je vykonávaný vo virtuálnom stroji
- Počas behu programu je možné preniesť a spustiť kód z jednej platformy na druhú — napríklad filtrovacia funkcia sa môže preniesť na server a spúšťať sa tam

Nevýhody:

- Menšia rýchlosť systému oproti *b)* a *c)* — program nie je vykonávaný priamo, ale vo virtuálnom stroji
- Použitie iba na tých platformách, na ktorých je implementovaný virtuálny stroj
- Obmedzenie na jediný programovací jazyk (resp. na jedno prostredie, napr. .NET)

b) Použitie knižnice/prostredia, ktoré skrýva rozdiely medzi platformami (wxWidgets [wxW], SDL [SDL], Lazarus [Laz], ..., t.j. ten istý zdrojový kód sa dá skompilovať a spustiť na viacerých platformách):
Výhody:

- Použitie viacerých programovacích jazykov — tých, pre ktoré existuje daná knižnica
- Možnosti jednej platformy môžu byť emulované knižnicou na druhej platforme, ak ich druhá platforma nepodporuje
- Rýchlosť systému je väčšia oproti *a)* — program je vykonávaný priamo

Nevýhody:

- Rýchlosť systému je menšia oproti *c)* — program volá funkcie knižnice a tá potom volá funkcie systému
- Použitie iba na tých platformách, na ktorých je implementovaná daná knižnica
- Vo všeobecnosti nie je možné preniesť skompilovaný kód z jednej platformy na druhú a spustiť ho tam

c) Použitie aplikačného programovacieho rozhrania spoločného pre viac platforiem (POSIX [POS], t.j. ten istý zdrojový kód sa dá skompilovať a spustiť na viacerých platformách):
Výhody:

- Dostupnosť na veľa platformách (viac ako *a)* a *b)*)
- Väčšia rýchlosť systému oproti *a)* a *b)* — program je vykonávaný priamo a sú volané priamo funkcie systému

Nevýhody:

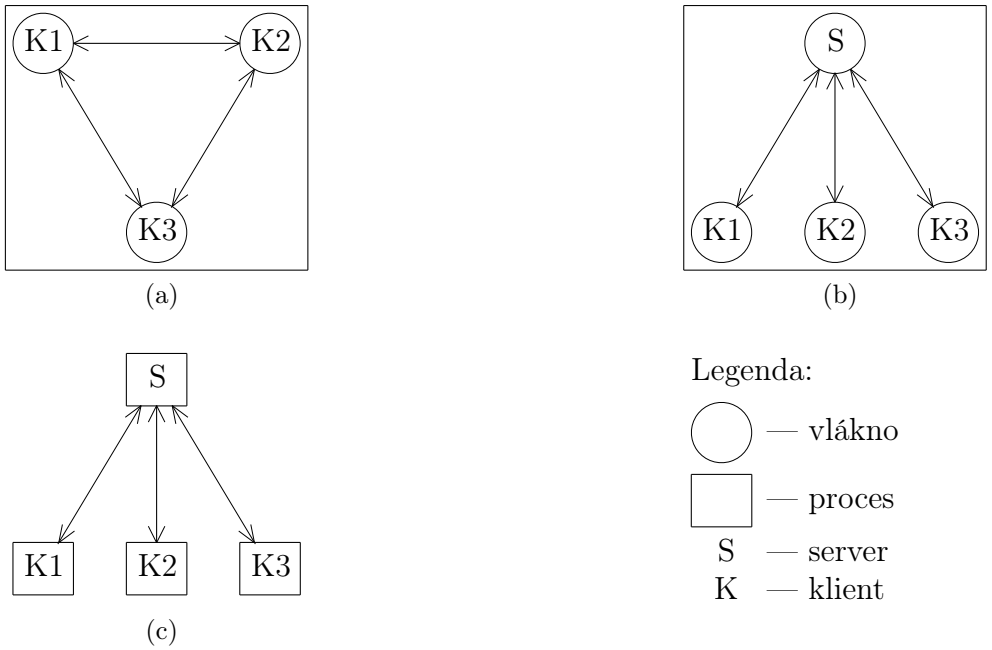
- Použitie iba na tých platformách, ktoré podporujú dané aplikačné programovacie rozhranie
- Možnosti platforiem sú obmedzené iba na definované aplikačné programovacie rozhranie
- Obmedzenie (prakticky) na jediný programovací jazyk — ANSI C
- Vo všeobecnosti nie je možné preniesť skompilovaný kód z jednej platformy na druhú a spustiť ho tam

Kvôli rýchlosti systému a lepšej podpore platforiem som zvolil možnosť *c) programovací jazyk ANSI C a aplikačné programovacie rozhranie POSIX.*

4.1.3 Architektúra systému

Architektúra systému je základná organizácia systému, ktorá určuje čo sú jeho prvky (klienti, server), ako medzi sebou komunikujú, atď. (Obrázok 4.1 – strana 15)

a) Systém je implementovaný ako jeden proces. Klienti sú vlákna daného procesu. Na prácu systému (komunikácia, oblasti procesov, ...) je použitá zdieľaná pamäť. Klienti komunikujú medzi sebou priamo, server neexistuje. (Obrázok 4.1a – strana 15)
Výhody:



Obrázok 4.1: Architektúra systému

- Najjednoduchšie na implementáciu
- Využitie štandardných prostriedkov operačného systému na komunikáciu
- Vytvorenie nového klienta znamená vytvorenie nového vlákna

Nevýhody:

- Klient musí byť bezpečný voči vláknam (*thread-safe*), t.j.
  - Nesmie používať globálne premenné
  - Všetky knižnice, ktoré používa, musia byť bezpečné voči vláknam (*thread-safe*)
- Nižšia možnosť reálneho využitia
- Nedá sa hovoriť o centralizovanom systéme, hoci neexistencia servera neznamená zmenu sémantiky komunikačných operácií na strane klienta
- Klienti si môžu (nechcene) prepisovať svoje dáta

b) Systém je implementovaný ako jeden proces. Klienti aj server sú vlákna daného procesu. (Obrázok 4.1b – strana 15)
Výhody:

- Server priamo volá filtrovacie funkcie
- Server priamo signalizuje semafory
- Na rozdiel od riešenia *a)* centralizovaný server existuje
- Vytvorenie nového klienta znamená vytvorenie nového vlákna

Nevýhody:

- Zložitejšie na implementáciu ako riešenie *a)*
- Menej efektívne ako riešenie *a)*
- Klient musí byť bezpečný voči vláknam (*thread-safe*), t.j.
  - Nesmie používať globálne premenné
  - Všetky knižnice, ktoré používa, musia byť bezpečné voči vláknam (*thread-safe*)
- Nižšia možnosť reálneho využitia
- Klienti si môžu (nechcene) prepisovať svoje dáta

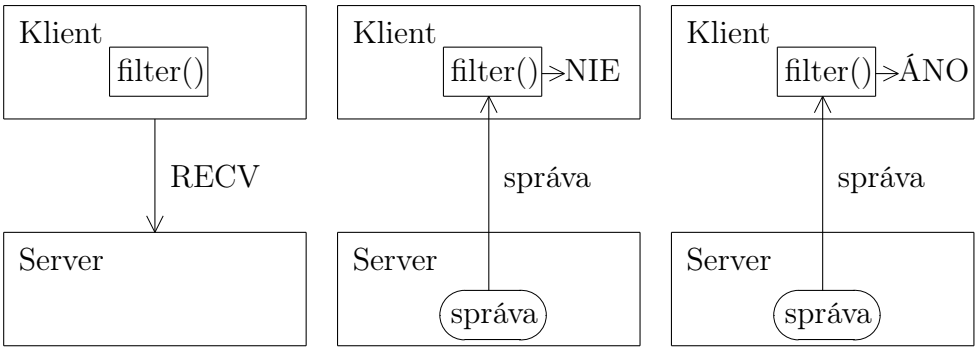
c) Server aj klienti sú samostatné procesy, ktoré medzi sebou komunikujú cez *sockety*[2]. (Obrázok 4.1c – strana 15)
Výhody:

- Klient nemusí byť bezpečný voči vláknam, t.j.
  - Môže používať globálne premenné
  - Môže používať knižnice, ktoré nie sú bezpečné voči vláknam
- Na rozdiel od riešenia *a)* centralizovaný server existuje
- Vyššia možnosť reálneho využitia
- Server a klienti môžu byť na rôznych platformách
- Klienti si nemôžu prepisovať svoje dáta

Nevýhody:

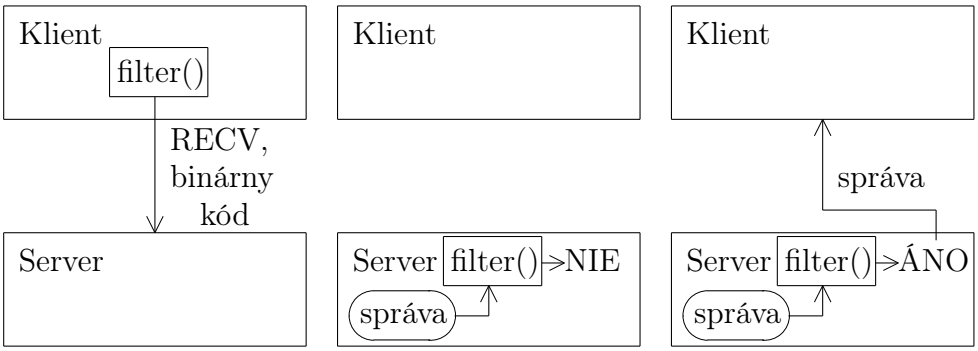
- Zložitejšie na implementáciu ako riešenia *a)* a *b)*
- Menej efektívne ako riešenia *a)* a *b)*
- Zložitejšia komunikácia ako riešenia *a)* a *b)*
- Treba vyriešiť volanie filtrovacích funkcií
- Treba vyriešiť signalizovanie semaforov

[2]Koncový bod komunikácie



(a) Operácia RECV (b) Neakceptovanie správy (c) Akceptovanie správy

Obrázok 4.2: Filtrovacie funkcie na strane klienta



(a) Operácia RECV (b) Neakceptovanie správy (c) Akceptovanie správy

Obrázok 4.3: Filtrovacie funkcie na strane servera — 1

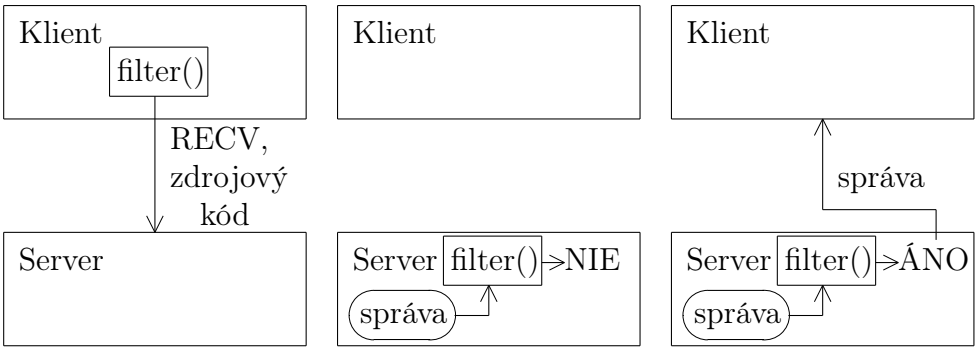
- Treba vyriešiť vytvorenie nového klienta

Z dôvodu širšieho reálneho využitia a pretože odpadá nutnosť klienta byť bezpečný voči vláknam som zvolil možnosť *c)*.

4.1.4 Volanie filtrovacích funkcií

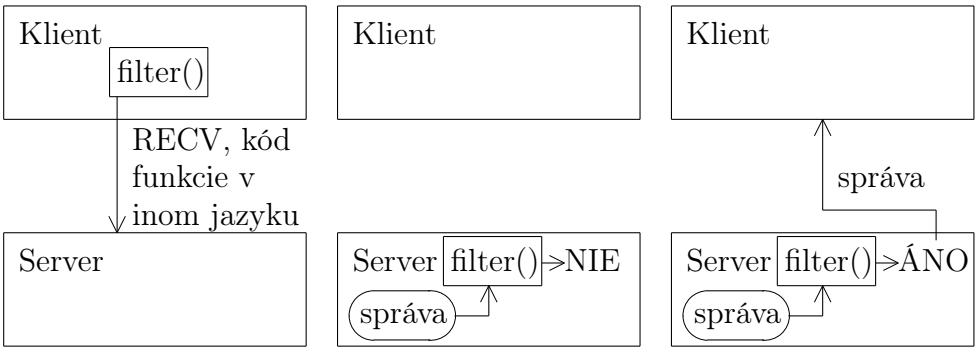
Volanie filtrovacích funkcií vyžaduje odpovede na otázky kde?, kedy? a ako? spúšťať filtrovacie funkcie.

a) Filtrovacie funkcie sa vykonávajú na strane klienta. T.j. filtrovacie funkcie sú skompilované počas kompilácie klienta, sú uložené na klientovi, nikam inam sa neprenášajú a na klientovi sa aj spúšťajú. Z toho vyplýva, že keď sa spúšťa filtrovacia funkcia, tak sa obsah správy musí najprv



(a) Operácia RECV (b) Neakceptovanie správy (c) Akceptovanie správy

Obrázok 4.4: Filtrovacie funkcie na strane servera — 2



(a) Operácia RECV (b) Neakceptovanie správy (c) Akceptovanie správy

Obrázok 4.5: Filtrovacie funkcie na strane servera — 3

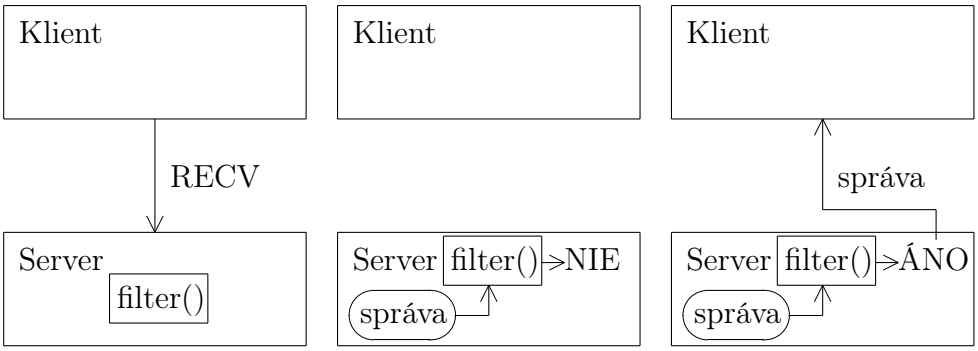
preniesť zo servera na klienta. (Obrázok 4.2 – strana 17)
Výhody:

- Najjednoduchšie z pohľadu programovania klienta a systému — netreba robiť žiadne kroky navyše
- Klient a server nemusia byť na rovnakej platforme

Nevýhody:

- Neefektívne — server posiela klientovi obsah správy, aj keď ju klient odmietne
- Nutnosť protokolu pre spúšťanie filtrovacích funkcií a návratu výsledku na server

b) Filtrovacie funkcie sa vykonávajú na strane servera.



(a) Operácia RECV (b) Neakceptovanie správy (c) Akceptovanie správy

Obrázok 4.6: Filtrovacie funkcie na strane servera — 4

b1) Na server sa pošle skompilovaný kód funkcie. T.j. filtrovacie funkcie sú skompilované počas kompilácie klienta, sú uložené na klientovi, podľa potreby sa prenesú (ich skompilovaný kód) na server a na serveri sa aj spúšťajú. (Obrázok 4.3 – strana 17)
Výhody:

- Efektívne — klientovi sa nepošle správa pokiaľ ju nechce
- Jednoduché na implementáciu v prípade, že skompilovaný kód je medzikód, ktorý vykonáva virtuálny stroj (Java, .NET umožňujú priamočiaru výmenu funkcií na úrovni binárneho medzikódu)

Nevýhody:

- Ak je skompilovaný kód skutočne vykonávaný procesorom (t.j. nie je to medzikód vykonávaný virtuálnym strojom), tak to má nasledovné nevýhody:
- Je možné iba v prípade, že klient a server sú na rovnakej platforme
- Vyžaduje špeciálnu konštrukciu filtrovacích funkcií — je potrebná znalosť práce kompilátora
  - Buď je funkcia skompilovaná v zdieľanej knižnici — v tom prípade sa na server posiela celá zdieľaná knižnica (súbor) a tá sa na serveri dynamicky nalinkuje.
  - Alebo sa na server posiela kód funkcie skopírovaný z operačnej pamäte (treba identifikovať začiatok a koniec kódu funkcie) a na serveri sa uloží do alokovanej pamäťe (treba umožniť spúšťanie kódu v dátovej oblasti).

b2) Na server sa pošle zdrojový kód funkcie. T.j. filtrovacie funkcie sú kompilované počas behu servera, sú uložené na klientovi, podľa potreby sa prenesú (ich zdrojový kód) na server a na serveri sa aj spúšťajú. (Obrázok 4.4 – strana 18)
Výhody:

- Efektívne — klientovi sa nepošle správa pokiaľ ju nechce
- Klient a server nemusia byť na rovnakej platforme
- Je možné konštruovať filtrovacie funkcie za behu

Nevýhody:

- Vyžaduje prítomnosť kompilátora a prípadných knižníc na strane servera
- Neefektívne — funkciu treba skompilovať

b3) Na server sa pošle kód funkcie napísaný v inom (platformovo nezávislom, interpretovanom) jazyku (napr. Javascript [Javb]). T.j. filtrovacie funkcie sa nekompilujú (sú interpretované počas behu servera), sú uložené na klientovi, podľa potreby sa prenesú (ich zdrojový kód) na server a na serveri sa aj spúšťajú. (Obrázok 4.5 – strana 18)
Výhody:

- Efektívne — klientovi sa nepošle správa pokiaľ ju nechce
- Klient a server nemusia byť na rovnakej platforme
- Je možné konštruovať filtrovacie funkcie za behu

Nevýhody:

- Vyžaduje prítomnosť interpretera zvoleného jazyka na strane servera
- Neefektívne — interpretovanie je pomalšie ako priame vykonávanie
- Použitie dvoch programovacích jazykov namiesto jedného

b4) Na server sa pošle identifikátor funkcie (napríklad meno). T.j. filtrovacie funkcie sú skompilované počas kompilácie servera, sú uložené na serveri, nikam inam sa neprenášajú a na serveri sa aj spúšťajú. (Obrázok 4.6 – strana 19)
Výhody:

- Efektívne — klientovi sa nepošle správa pokiaľ ju nechce

- Klient a server nemusia byť na rovnakej platforme

Nevýhody:

- Vyžaduje prítomnosť filtrovacích funkcií na strane servera

Z dôvodu vyššej efektívnosti a z dôvodu, že pri plánovanom využití systému nevadí že filtrovacie funkcie musia byť prítomné na serveri som zvolil možnosť *b4)*.

4.1.5 Signalizovanie semaforov

Signalizovanie semaforov vyžaduje odpoveď na otázku, akým spôsobom sa realizuje signalizovanie semaforov pri vykonávaní základných operácií.

a) Klient nečaká na skutočnom semafore, ale čaká na komunikáciu zo *socketu*. T.j. signalizovanie semaforu je implementované ako poslanie správy do *socketu*.
Výhody:

- Jednoduché na implementáciu

Nevýhody:

- Nie je možné asynchrónne posielanie správ
- Pokiaľ klient čaká na signalizáciu, tak server nemôže komunikovať s klientom (t.j. napr. nie je možné vykonávať filtrovacie funkcie na strane klienta)

b) Klient má (aspoň) dve vlákna. Jedno je štandardné vlákno, ktoré vykonáva kód klienta, posiela správy na server a ktoré môže čakať na semafore. Druhé je vytvorené klientskou knižnicou a prijíma správy zo servera. Zároveň, keď dostane pokyn zo servera, tak signalizuje semafor na ktorom čaká prvé vlákno.
Výhody:

- Je možné asynchrónne posielanie správ
- Server môže komunikovať s klientom aj keď klient čaká na semafore (t.j. napr. je možné vykonávať filtrovacie funkcie na strane klienta)

Nevýhody:

- Zložitejšie na implementáciu na implementáciu

- Menej efektívne — všetka výmena správ funguje pomocou komunikácie medzi dvoma vláknami

Z dôvodu väčšej univerzálnosti (server môže komunikovať s klientom aj keď klient čaká na semafore) som zvolil možnosť *b)*.

4.1.6 Vytvorenie nového klienta

Vytvorenie nového klienta vyžaduje odpovede na otázky kde?, kedy? a ako? sa vytvára nový klient (pri základnej operácii $START$).

a) Nového klienta vytvára server.

a1) Klient sa spustí na mieste kde beží server.
Výhody:

- Jednoduché na implementáciu

Nevýhody:

- Na strane servera je nutná prítomnosť klienta pre platformu na ktorej beží server
- Neefektívne — všetci klienti až na prvého bežia tam kde beží server

a2) Klient sa spustí na inom mieste ako kde beží server.
Výhody:

- Vyššia efektivita — každý klient môže bežať na inom mieste

Nevýhody:

- Vyžaduje existenciu a beh aspoň jedného ďalšieho servera, ktorý má k dispozícii klienta pre platformu na ktorej beží a vie ho na požiadanie spustiť. Alternatívou je použitie mechanizmu SSH na vytvorenie procesu na inom systéme. Toto však vyžaduje riešenie problémov spojených s autentifikáciou, mapovaním procesov na procesory atď.

b) Nového klienta vytvára klient. Vytvorenie klienta znamená pre server vygenerovanie nového identifikátora a povolenie pripojenia klienta s daným identifikátorom. Prvý klient má špeciálny identifikátor, ktorý má povolené pripojiť sa na server a ten vytvára ďalších klientov.
Výhody:

- Jednoduché na implementáciu

Nevýhody:

- Neefektívne — všetci klienti bežia na jednom mieste

Pre jednoduchosť a z dôvodu postačujúceho riešenia pre plánované využitie som zvolil možnosť *b)*.

4.2 Návrh implementácie

Najprv zopakujem v krátkosti zvolené možnosti implementácie, a potom uvediem detaily návrhu.

- Multiplatformovosť
- ANSI C + POSIX
- klient a server sú samostatné procesy
- komunikácia cez *sockety*
- filtrovacie funkcie sú uložené na serveri a tam sa aj vykonávajú
- klient má (aspoň) dve vlákna
  - jedno vykonáva prácu klienta, posiela správy na server a čaká na semafore
  - druhé prijíma správy zo servera a signalizuje semafor
- nového klienta vytvára klient

4.2.1 Klientské komunikačné rozhranie

Klientské komunikačné rozhranie je aplikačné rozhranie, pomocou ktorého klient vykonáva výmenu správ.
Rozhranie pozostáva z nasledovných troch typov funkcií.

a) Obslužné funkcie:

```
int
initialize_clients(
                    int num_clients,
                    u_long **clients_ptr,
                    char *argv[]
                    );
```

Funkcia *initialize_clients* sa spúšťa iba raz, na začiatku programu.
Vytvorí *num_clients* kópií procesu (a pripojí ich k centrálnemu serveru) a do *clients_ptr* uloží ukazovateľ na zoznam identifikátorov klientov (vytvorených procesov) (v prípade neúspešnej inicializácie sa v iniciálnom procese uloží do *clients_ptr* ukazovateľ na zoznam identifikátorov vytvorených procesov — pridané kvôli implementácii TPL).
Parameter *argv* je druhý parameter funkcie *main* (tento parameter je využitý v prípade, keď sa na vytvorenie kópií procesu používa funkcia *spawn* namiesto funkcie *fork*).
V prípade chyby funkcia vráti hodnotu 0.
Inak je návratová hodnota indexom do zoznamu identifikátorov klientov ((∗*clients_ptr*)[návratová hodnota - 1] = identifikátor klienta).

```
int
initialize_client(
                    u_long rpid
                    );
```

Funkcia *initialize_client* inicializuje proces (pripojí ho k centrálnemu serveru), tak aby jeho identifikátor bol *rpid*, kde *rpid* je nenulové číslo.
V prípade úspechu funkcia vráti hodnotu 0.
Inak je návratová hodnota záporné číslo.

V aplikačnom programe sa používa buď funkcia *initialize_clients* v prípade ak všetky procesy majú byť kópiami prvého spusteného procesu, alebo funkcia *initialize_client* ak spúšťanie procesov zabezpečuje iný mechanizmus.

```
void
deinitialize_client(
                      );
```

Funkcia *deinitialize_client* deinicializuje proces (odpojí ho od centrálneho servera).

```
u_long
getmyid(
          );
```

Funkcia *getmyid* vráti identifikátor procesu.

b) Funkcie pre prácu so správami:

```
int
mps_create(
            char *mtext,
            u_long mlength,
            u_long *message_id_ptr
            );
```

Funkcia *mps_create* pošle operáciu $CREATE$ na server a počká na jej vykonanie.
mtext_ptr je text správy (ktorý sa pošle na server).
mlength je dĺžka správy.
message_id_ptr je ukazovateľ na miesto, kam sa uloží identifikátor správy.
V prípade úspechu funkcia vráti hodnotu 0.
Inak je návratová hodnota nenulové číslo.

```
int
mps_destroy(
             u_long message_id
             );
```

Funkcia *mps_destroy* pošle operáciu $DESTROY$ na server a počká na jej vykonanie.
message_id je identifikátor správy.
V prípade úspechu funkcia vráti hodnotu 0.
Inak je návratová hodnota nenulové číslo.

```
int
mps_send(
            u_long message_id,
            u_long num_process_ids,
            u_long *process_ids
            );
```

Funkcia *mps_send* pošle operáciu $SEND$ na server a počká na jej vykonanie.
message_id je identifikátor správy.
num_process_ids je počet záznamov v zozname identifikátorov klientov (prijímateľov) *process_ids*.
V prípade úspechu funkcia vráti hodnotu 0.
Inak je návratová hodnota nenulové číslo.

```
int
mps_send_async(
                    u_long message_id,
                    u_long num_process_ids,
                    u_long *process_ids
                    );
```

Funkcia *mps_send_async* pošle operáciu $SEND$ na server a nečaká na jej vykonanie.
message_id je identifikátor správy.
num_process_ids je počet záznamov v zozname identifikátorov klientov (prijímateľov) *process_ids*.
V prípade úspechu funkcia vráti hodnotu 0.
Inak je návratová hodnota nenulové číslo.

```
int
mps_recv(
            char **mtext_ptr,
            u_long *mlength_ptr,
            u_long *message_id_ptr,
            u_long *process_id_ptr,
            u_long num_process_ids,
            u_long *process_ids,
            char *filter
            );
```

Funkcia *mps_recv* pošle operáciu $RECV$ na server a počká na jej vykonanie.
mtext_ptr je ukazovateľ na miesto, kam sa uloží text správy (lokálna kópia).
mlength_ptr je ukazovateľ na miesto, kam sa uloží dĺžka správy.
message_id_ptr je ukazovateľ na miesto, kam sa uloží identifikátor správy.
process_id_ptr je ukazovateľ na miesto, kam sa uloží identifikátor odosielateľa.
num_process_ids je počet záznamov v zozname identifikátorov klientov (odosielateľov) *process_ids*.
filter je názov filtrovacej funkcie, resp. $NULL$ ak filtrovanie nemá byť vykonávané.
V prípade úspechu funkcia vráti hodnotu 0.
Inak je návratová hodnota nenulové číslo.

c) Funkcie pre prácu s procesmi:

```
int
mps_start(
        char **mtext_ptr,
        u_long *mlength_ptr,
        u_long *message_id_ptr
        );
```

Funkcia *mps_start* pošle operáciu $START$ na server a počká na jej vykonanie.
mtext_ptr je ukazovateľ na miesto, kam sa uloží text správy (lokálna kópia).
mlength_ptr je ukazovateľ na miesto, kam sa uloží dĺžka správy.
message_id_ptr je ukazovateľ na miesto, kam sa uloží identifikátor správy.
V prípade úspechu funkcia vráti hodnotu 0.
Inak je návratová hodnota nenulové číslo.

Táto funkcia sa používa v prípade, že program používa na inicializáciu funkciu *initialize_client*.
Ak program používa na inicializáciu funkciu *initialize_clients*, tak je funkcia *mps_start* nepotrebná.

```
int
mps_end(
        );
```

Funkcia *mps_end* pošle operáciu END na server a počká na jej vykonanie.
V prípade úspechu funkcia vráti hodnotu 0.
Inak je návratová hodnota nenulové číslo.

4.2.2 Komunikačný protokol

Komunikačný protokol je interný protokol medzi klientom a serverom, pomocou ktorého medzi sebou klient a server komunikujú.
Pokiaľ nie je povedané inak, tak sú všetky zložky komunikačného protokolu typu *unsigned long* (32 bitové slovo).
$\longrightarrow$ označuje prenos z klienta na server.
$\longleftarrow$ označuje prenos zo servera na klienta.
$\sim$ označuje pokračovanie na ďalšom riadku.

Pripojenie klienta na server:

⟶ OpTp_CONNECT|CONNECTION_STRING|client_id
Odpoveď o úspešnom pripojení:
⟵ OpTp_ACCEPT_CONNECT|client_id
Odpoveď o neúspešnom pripojení:
⟵ OpTp_REJECT_CONNECT

Operácia *START*:
⟶ OpTp_START|signal_ptr|message_ptr|mlength_ptr|message_id_ptr
Odpoveď po vykonaní operácie:
⟵ OpTp_START_RET|signal_ptr|message_ptr|mlength_ptr|∼
message_id_ptr|message_id|mlength|message

Operácia *END*:
⟶ OpTp_END|signal_ptr
Odpoveď po vykonaní operácie:
⟶ OpTp_END_RET|signal_ptr

Operácia *CREATE*:
⟶ OpTp_CREATE|signal_ptr|message_id_ptr|mlength|message
Odpoveď po vykonaní operácie:
⟵ OpTp_CREATE_RET|signal_ptr|message_id_ptr|message_id

Operácia *DESTROY*:
⟶ OpTp_DESTROY|signal_ptr|message_id
Odpoveď po vykonaní operácie:
⟵ OpTp_DESTROY_RET|signal_ptr

Operácia *SEND*:
⟶ OpTp_SEND|signal_ptr|message_id|num_ids|process_ids
Odpoveď po vykonaní operácie:
⟵ OpTp_SEND_RET|signal_ptr

Operácia *RECV*:
⟶ OpTp_RECV|signal_ptr|message_ptr|mlength_ptr|∼
message_id_ptr|process_id_ptr|num_ids|process_ids|flength|function
Odpoveď po vykonaní operácie:
⟵ OpTp_RECV_RET|signal_ptr|message_ptr|mlength_ptr|∼
message_id_ptr|process_id_ptr|message_id|process_id|mlength|∼
message

Vysvetlivky:

OpTp_∗ sú identifikátory typov prenosov

CONNECTION_STRING je identifikátor systému výmeny správ

client_id je identifikátor klienta

signal_ptr je ukazovateľ na semafor, ktorý sa má signalizovať po vykonaní operácie

message_ptr je ukazovateľ na miesto, kam sa uloží správa po vykonaní operácie

mlength_ptr je ukazovateľ na miesto, kam sa uloží dĺžka správy po vykonaní operácie

message_id_ptr je ukazovateľ na miesto, kam sa uloží identifikátor správy po vykonaní operácie

message_id je identifikátor správy

process_id_ptr je ukazovateľ na miesto, kam sa uloží identifikátor procesu po vykonaní operácie

process_id je identifikátor procesu

mlength je dĺžka správy

message je text správy — *mlength* bytov

num_ids je počet identifikátorov procesov

process_ids je zoznam identifikátorov procesov dĺžky *num_ids*

flength je dĺžka názvu funkcie

function je názov funkcie — $flength$ bytov

4.2.3 Štruktúra záznamu priebehu výmeny správ (logu)

Záznam priebehu výmeny správ (log) vytvára server počas svojho behu a zapisuje do neho všetky relevantné udalosti (t.j. vykonávané operácie, ktoré boli popísané v kapitole 2).
Záznam priebehu výmeny správ pozostáva z hlavičky a udalostí.
Prvý riadok záznamu je hlavička a každý ďalší neprázdny riadok je jedna udalosť.
Hlavička a udalosti pozostávajú z hodnôt (stĺpcov) oddelených bodkočiarkou.

Hlavička:
eventid;resultid;time;pid;sync/async;h6;h7;h8;h9;h10;h11;h12;errorid;text

Stĺpce *eventid*, *resultid*, *time*, *pid*, *sync/async*, *errorid*, *text* sú spoločné pre každú udalosť.
Stĺpce *h6*, *h7*, *h8*, *h9*, *h10*, *h11*, *h12* sú špecifické pre každú udalosť.
Ak niektorý stĺpec nie je aplikovateľný pre danú udalosť, tak hodnota stĺpca je prázdny reťazec.

Spoločné stĺpce:
eventid — identifikátor udalosti — číselná hodnota:
- 1 — spustenie servera
- 2 — ukončenie servera
- 3 — pripojenie klienta k serveru
- 4 — odpojenie klienta od servera
- 5 — operácia $START$
- 6 — operácia END
- 7 — operácia $CREATE$
- 8 — operácia $DESTROY$
- 9 — operácia $SEND$
- 10 — operácia $RECV$
- 11 — vykonanie poslania a/alebo prijatia správy (mimo operácií)

resultid — výsledok udalosti — číselná hodnota:
- 0 — nastala chyba
  nemôže nastať pri udalostiach 2 a 4
- 1 — chyba nenastala — úspešné vykonanie
  (pri udalostiach 9 a 10 — správa bola okamžite doručená)
- 2 — chyba nenastala — správa nebola odoslaná
  (jeden z prijímateľov nie je pripojený k serveru)
  môže nastať iba pri udalostiach 9 a 11
- 3 — chyba nenastala — poslanie správy bolo odložené na neskôr
  (nie je komu poslať, ...)
  môže nastať iba pri udalosti 9
- 4 — chyba nenastala — správa nebola prijatá
  (jeden z odosielateľov nie je pripojený k serveru)
  môže nastať iba pri udalostiach 10 a 11
- 5 — chyba nenastala — prijatie správy bolo odložené na neskôr
  (nie je od koho prijať, ...)
  môže nastať iba pri udalosti 10

time — čas udalosti (UTC) — reťazec
formát: "YYYY-MM-DD hh:mm:ss.lll"
YYYY — rok
MM — mesiac
DD — deň
hh — hodiny
mm — minúty
ss — sekundy
lll — milisekundy
pid — identifikátor klienta — číselná hodnota
klient, ktorý inicioval udalosť
výnimka: udalosť 11 nie je iniciovaná klientom
v tomto prípade je *pid* identifikátor klienta, ktorý prijíma správu
sync/async — synchrónna/asynchrónna operácia — reťazec:
sync — operácia je z klienta vykonávaná synchrónne
async — operácia je z klienta vykonávaná asynchrónne
("synchrónne vykonávaná" = klient čaká na semafore)
errorid — chybová hodnota — číselná hodnota
ak *resultid* je 0, tak *errorid* je identifikátor chyby
text — textový popis udalosti — reťazec

Špecifické stĺpce:
Udalosť 1 (spustenie servera):
h6 — IP adresa siete, na ktorej server počúva (0.0.0.0) — reťazec
h7 — číslo portu, na ktorom server počúva (19792) — číselná hodnota
h8 — verzia logu (100) — číselná hodnota

Udalosť 2 (ukončenie servera):
žiadne špecifické stĺpce

Udalosť 3 (pripojenie klienta k serveru):
h6 — IP adresa klienta — reťazec
h7 — číslo portu klienta — číselná hodnota

Udalosť 4 (odpojenie klienta od servera):
žiadne špecifické stĺpce

Udalosť 5 (operácia $START$):
h6 — identifikátor nového klienta — číselná hodnota
h7 — identifikátor správy — číselná hodnota
h8 — dĺžka správy — číselná hodnota

Udalosť 6 (operácia END):
žiadne špecifické stĺpce

Udalosť 7 (operácia $CREATE$):
h6 — dĺžka správy — číselná hodnota
h7 — prvých najviac 16 znakov textu správy — reťazec
h8 — identifikátor správy — číselná hodnota

Udalosť 8 (operácia $DESTROY$):
h6 — identifikátor správy — číselná hodnota

Udalosť 9 (operácia $SEND$):
h6 — identifikátor posielanej správy — číselná hodnota
h7 — počet prijímateľov správy — číselná hodnota
h8 — zoznam prijímateľov správy (oddelených čiarkami) — reťazec
h9 — identifikátor klienta (skutočného príjemcu) — číselná hodnota
h10 — identifikátor prijatej správy — číselná hodnota

Udalosť 10 (operácia $RECV$):
h6 — počet odosielateľov správy — číselná hodnota
h7 — zoznam odosielateľov správy (oddelených čiarkami) — reťazec
h8 — názov filtrovacej funkcie — reťazec
h9 — identifikátor klienta (skutočného odosielateľa) — číselná hodnota
h10 — identifikátor prijatej správy — číselná hodnota
h11 — identifikátor posielanej správy — číselná hodnota
h12 — dĺžka prijatej správy — číselná hodnota

Udalosť 11 (vykonanie poslania a/alebo prijatia správy):
h6 — identifikátor klienta (odosielateľa) — číselná hodnota
h7 — identifikátor prijatej správy — číselná hodnota
h8 — identifikátor posielanej správy — číselná hodnota
h9 — dĺžka prijatej správy — číselná hodnota

4.2.4 Architektúra klientskej časti

Klientská časť sa skladá z dvoch vlákien.

Prvé vlákno je štandardné vlákno klienta a má na starosti:

- Inicializáciu a deinicializáciu klientskej časti
- Pripojenie k serveru a odpojenie od servera
- Vytvorenie a zrušenie druhého vlákna
- Vytváranie nových klientov (ako kópií svojho procesu)
- Posielanie operácií na server
- Čakanie na semafore (pri synchrónne vykonávaných operáciách)

Druhé vlákno je nové vlákno, ktoré sa vytvorí pri inicializácii klientskej časti a má na starosti:

- Prijímanie odpovedí od servera
- Vytváranie lokálnych kópií prijatých správ
- Signalizovanie semaforu v prvom vlákne (pri synchrónne vykonávaných operáciách)

4.2.5 Architektúra servera

Server sa skladá z hlavného vlákna a jedného alebo viacerých klientských vlákien.

Hlavné vlákno je štandardné vlákno servera a má na starosti:

- Inicializáciu a deinicializáciu servera
- Vytváranie klientských vlákien (podľa potreby)
- Čakanie na pripojenie klientov
- Overovanie platnosti klienta (autentifikácia)
- Registrovanie platných klientov (okrem iného pridelenie klienta klientskému vláknu)

Klientské vlákna sú vytvárané hlavným vláknom. Jedno klientské vlákno má na starosti jedného alebo viacerých pripojených klientov a robí:

- Prijímanie operácií od klienta
- Volanie filtrovacích funkcií
- Vykonávanie operácií
- Posielanie odpovedí klientovi

Vykonávanie operácií sa robí tak skoro, ako sa dá (t.j. ak sa operácia môže vykonať hneď po jej prečítaní, tak sa aj vykoná).
Ak sa pri operácii $SEND$ ($RECV$) v zozname prijímateľov (odosielateľov) vyskytne klient, ktorý nie je pripojený, tak doručenie správy zlyhá.
Dôsledkom je, že ak sa klient odpojí od servera, tak zlyhajú všetky operácie $SEND$ ($RECV$) nachádzajúce sa na serveri, ktoré ešte neboli vykonané a odpojený klient je v ich zozname prijímateľov (odosielateľov).

Hlavné vlákno, ako aj klientské vlákna zapisujú podľa potreby do záznamu priebehu výmeny správ (logu). Pre vzájomné vylúčenie vstupu do kritickej sekcie (zápis do logu) sa používa semafor.

5 Využitie systému

V tejto kapitole sa budem venovať možnostiam využitia môjho naprogramovaného systému a jeho prípadným rozšíreniam.

5.1 Knižnica TPL

Knižnica TPL [Pla03] (Thread Parallel Library) je komunikačná knižnica pre výmenu správ. Je minimalistická, bezpečná voči vláknam (*thread-safe*) a aplikácie ktoré ju využívajú môžu byť viac-vláknové (*multi-threaded*). TPL priamo implementuje základné komunikačné operácie z kapitoly 2. TPL bola úspešne použitá na implementáciu veľkých paralelných aplikácií, napríklad paralelnej verzie renderovacieho systému POV-Ray [Pla02], [Pla03]. Pre overenie správnosti môjho návrhu je logické implementovať nezmenené rozhranie TPL s použitím mojich funkcií. Prípadné chyby v mojom návrhu by sa totiž prejavili buď nemožnosťou implementácie funkcií TPL alebo zložitou implementáciou. Táto simulácia má tiež praktické využitie — môj systém sa stane použiteľný na ladenie programov využívajúcich knižnicu TPL.

Pri implementácii knižnice treba riešiť nasledovné problémy (rozdiely TPL oproti môjmu systému):

Problém: Identifikátory procesov v TPL sú v rozsahu 0..počet procesov-1, v mojom systéme to tak nie je.

Riešenie: Treba mať prevodovú tabuľku medzi identifikátormi procesov v TPL a identifikátormi v mojom systéme.

Problém: Knižnica TPL používa na identifikáciu filtrovacej funkcie ukazovateľ na jej kód, môj systém používa na identifikáciu filtrovacej funkcie jej meno.

Riešenie: Treba mať prevodovú tabuľku medzi ukazovateľmi na filtrovacie funkcie a ich menami.

Problém: Knižnica TPL používa iné parametre pre filtrovacie funkcie.

Riešenie: Treba doprogramovať na serveri druhý typ filtrovacích funkcií.

Problém: Zoznam príjemcov vo funkcii *tpl_send* (operácia SEND) v TPL znamená, že sa správa pošle každému procesu v zozname, v mojom systéme to znamená, že sa pošle prvému (podľa času) procesu zo zoznamu, ktorý ju

môže prijať.

Riešenie: V implementácia funkcie *tpl_send* treba každému procesu zo zoznamu príjemcov poslať správu zvlášť.

Problém: Funkcia *tpl_recv* (operácia RECV) v TPL nemá zoznam akceptovaných odosielateľov (tento zoznam je skrytý vo filtrovacej funkcii).

Riešenie: V implementácii funkcie *tpl_recv* treba prijímať správu od všetkých procesov pripojených k serveru.

Problém: Ak niektorý proces ukončí svoju prácu (funkcia *tpl_deinitialize*) a server ešte nevykonal všetky operácie daného klienta, tak sa tie operácie nevykonajú úspešne (pretože klient sa odpojil od servera).

Riešenie: V implementácii funkcie *tpl_deinitialize* treba pred odpojením od servera vykonať bariérovú synchronizáciu.

Problém: Ak sa nepodarí inicializácia, tak TPL zničí všetky vytvorené procesy (môj systém nie).

Riešenie: V prípade neúspešnej inicializácie treba vytvoriť zoznam vytvorených procesov a v implementácii funkcie *tpl_initialize* ich treba zničiť.

Implementácia je robená (celkom prirodzene) v rovnakom programovacom jazyku v akom je spravený môj systém — ANSI C + POSIX.

Implementácia funkcie *tpl_initialize*:

```
u_long num_clients, *clients = NULL;

void tpl_initialize(int *nr_tasks, int *rank, int *argc,
                    char ***argv)
{
  pid_t mypid, *proc_pids;
  int ret, pos;

  num_clients = *nr_tasks;
  ret = initialize_clients(num_clients, &clients, *argv);

  *rank = ret - 1;
```

```
  if ( ret == 0 )
  {
    /* pri inicializácii nastala chyba, takže */
    /* zničím všetky vytvorené procesy vrátane súčasného */

    if ( clients != NULL )
    {
      mypid = getpid();
      proc_pids = (pid_t *) clients;

      for ( pos = 0; pos < num_clients; pos++ )
      {
        if (proc_pids[pos] != 0 &&
            proc_pids[pos] != mypid)
        {
          kill(proc_pids[pos], SIGKILL);
        }
      }
    }
    exit(1);
  }
}
```

num_clients je počet procesov
clients je zoznam identifikátorov procesov (prevodová tabuľka medzi identifikátormi procesov v TPL a v mojom systéme). (V prípade chybnej inicializácie obsahuje v iniciálnom procese zoznam identifikátorov vytvorených procesov.)

Vo funkcii *tpl_initialize* sa inicializujú klienti a vytvorí sa prevodová tabuľka pre identifikátory procesov. V prípade, že neprebehne inicializácia v poriadku, tak sa zničia všetky vytvorené procesy vrátane iniciálneho procesu.

Na vytvorenie prevodovej tabuľky pre filtrovacie funkcie je potrebný zásah programátora a nová funkcia *tpl_register_function*. (Toto je jediná vec, ktorá vyžaduje zásah do zdrojového kódu pri zmene architektúry z distribuovanej na centralizovanú.)

```
typedef struct _tpl_function_ {
  MATCHING_FUNC *addr;
  char *name;
  struct _tpl_function_ *next;
```

```
} tpl_function;

tpl_function *functions = NULL;

void tpl_register_function(MATCHING_FUNC *function,
                           char *function_name);
```

Túto funkciu je potrebné zavolať po volaní funkcie *tpl_ initialize* pre každú filtrovaciu funkciu, ktorá bude použitá v programe. Týmto sa vytvorí prevodová tabuľka pre filtrovacie funkcie (*functions*).
Implementácia funkcie *tpl_ deinitialize*:

```
void tpl_deinitialize(void)
{
  u_long mid_sync;

  if ( functions != NULL )
  {
    /* zmazanie prevodovej tabuľky pre filtrovacie funkcie */
    delete_functions(functions);
    functions = NULL;
  }

  if ( clients != NULL )
  {
    /* pre ľahšiu identifikáciu koncovej synchronizácie */
    /* v logu, vytvorím správu a pokiaľ ju nezmažem, tak */
    /* beží koncová synchronizácia */
    if ( mps_create("SYNC-END", 9, &mid_sync) ) mid_sync = 0;

    /* bariérová synchronizácia */
    barrier_synchronization();

    if ( mid_sync != 0 ) mps_destroy(mid_sync);

    /* zmazanie prevodovej tabuľky pre identifikátory */
    free(clients);
    clients = NULL;
  }

  deinitialize_client();
}
```

Vo funkcii *tpl_deinitialize* sa robí bariérová synchronizácia, vymažú sa prevodové tabuľky a deinicializuje sa klient.

Funkcia *tpl_create* slúži na vytvorenie lokálnej správy a funkcie *tpl_pk** slúžia na jej naplnenie. Obdobne funkcie *tpl_upk** slúžia na prečítanie lokálnej (prijatej) správy a funkcia *tpl_destroy* na jej zmazanie. Vnútorne je prvých 32 bitov správy vyhradených na užívateľskú značku správy (*tag*). Keďže funkcia *tpl_initialize* vytvára klientov na jednom počítači (platforme), tak funkcie *tpl_pk** a *tpl_upk** nerobia žiadnu konverziu medzi formátmi dát, ale iba priamo kopírujú dodané dáta.

```
typedef struct _tpl_message_ {
  void *addr;
  u_long length, pos;
} tpl_message;

void tpl_create(void **message);
void tpl_destroy(void *message);

void tpl_pkbyte(void *message, void *p, int nitems);
void tpl_upkbyte(void *message, void *p, int nitems);
```

Implementácia funkcie *tpl_send*:

```
void tpl_send(int *recipients, int nr_recipients, int tag,
              void *message)
{
  int rec;
  u_long mid;
  tpl_message *tpl_msg;

  if ( message == NULL ) return;

  tpl_msg = (tpl_message *) message;

  if ( tpl_msg->length >= 4 && tpl_msg->addr != NULL )
  {
    *((u_long *) (tpl_msg->addr)) = htonl(tag);
  }

  for ( rec = 0; rec < nr_recipients; rec++ )
```

```
  {
    if ( mps_create(tpl_msg->addr, tpl_msg->length, &mid) )
      continue;

    if ( mps_send_async(mid, 1, &(clients[recipients[rec]])) )
      continue;
  }

  tpl_destroy(message);
}
```

Vo funkcii *tpl_send* sa na vyhradené miesto v správe zapíše jej užívateľská značka (*tag*), pre každého z príjemcov sa na serveri vytvorí kópia správy, pošle sa príjemcovi (asynchrónne) a na konci sa lokálna kópia správy vymaže.

```
void tpl_recv(MATCHING_FUNC *match, int *sender, int *tag,
              void **message)
{
  tpl_message *tplmsg;
  u_long mid, msglen, pid;
  char *msg;

  if ( message == NULL ) return;

  if ( mps_recv(&msg, &msglen, &mid, &pid, num_clients,
                clients, find_function_name(match)) )
  {
    *sender = -1;
    *tag = -1;
    *message = NULL;
    return;
  }

  mps_destroy(mid);

  if ( msglen == 0 ||
       strncmp(msg, "SENDER_NOT_FOUND=", 17) == 0 )
  {
    free(msg);
    *sender = -1;
```

```
    *tag = -1;
    *message = NULL;
    return;
  }

  tplmsg = (tpl_message *) malloc(sizeof(tpl_message));
  if ( tplmsg == NULL )
  {
    free(msg);
    *sender = -1;
    *tag = -1;
    *message = NULL;
    return;
  }

  *sender = find_rank(pid);
  *tag = (msglen >= 4)?ntohl(*((u_long *) (msg))):0;
  *message = tplmsg;
  tplmsg->addr = msg;
  tplmsg->length = msglen;
  tplmsg->pos = (msglen >= 4)?4:0;
}
```

Vo funkcii *tpl_recv* sa nájde podľa ukazovateľa na kód filtrovacej funkcie a prevodnej tabuľky názov filtrovacej funkcie. Potom sa prijme správa od všetkých odosieľateľov, t.j. od prvého (podľa časovej pečiatky) zo zoznamu všetkých procesov v systéme. Táto správa sa následne vymaže zo servera. Nakoniec sa lokálna kópia správy skonvertuje do formátu pre knižnicu TPL.

5.2 Analyzátor záznamu priebehu výmeny správ (logu)

Analyzátor logu je nástroj, ktorý spracúva informácie z logu a prezentuje ich následne užívateľovi. Nástroj pracuje automaticky, t.j. bez zásahu užívateľa. Informácie, ktoré prezentuje užívateľovi (v súčasnej verzii) sú nasledovné. Po prvé sú to chyby, ktoré nastali v priebehu výmeny správ. Po druhé sú to štatistické informácie o priebehu výmeny správ.

Kontrole chýb sa venujem v časti 5.2.2.

Štatistické informácie sú počítané zvlášť pre každého klienta. Patria medzi ne tieto informácie:

- kedy a ako dlho bol klient pripojený k serveru
- počet, celková/priemerná/minimálna/maximálna dĺžka prijatých / odoslaných správ
- celkové/priemerné/minimálne/maximálne čakanie pri prijímaní / odosielaní (synchrónnom) správ

5.2.1 Príklad (logu)

V tejto časti ukážem a okomentujem záznam priebehu výmeny správ, ktorý vznikol pri spustení ukážkového programu v kapitole 3.

∼ označuje pokračovanie na ďalšom riadku.

Tu začína záznam priebehu výmeny správ.
eventid;resultid;time;pid;sync/async;h6;h7;h8;h9;h10;h11;h12;errorid;text
Hlavička logu.

1;1;2007-04-09 07:30:40.250;;;0.0.0.0;19792;100;;;;;;server startup
Štart práce servera.

3;1;2007-04-09 07:30:44.987;1;;127.0.0.1;1761;;;;;;;client connect
Pripojí sa úvodný klient (identifikátor 1) a začne sa inicializácia systému.

7;1;2007-04-09 07:30:44.991;1;sync;11;0x49,0x4e,0x49,0x54,0x2d,0x50,0x4f,∼
0x53,0x49,0x58,0x00;1;;;;;0;operation create
Klient vytvorí správu so špecifickým textom ("INIT-POSIX") a kým túto správu (identifikátor 1) nezmaže, tak všetky jeho operácie sú súčasťou inicializácie. ("INIT-POSIX" znamená, že inicializácia bude robená pomocou funkcie *fork*. Alternatívny spôsob inicializácie, v ktorom sa namiesto funkcie *fork* používa funkcia *spawnv* je indikovaný správou s textom "INIT-MINGW".)

5;1;2007-04-09 07:30:44.992;1;sync;2;2;9;;;;;0;operation start
8;1;2007-04-09 07:30:44.993;1;sync;2;;;;;;;0;operation destroy
Ďalej klient vytvorí (zaeviduje na serveri) operáciou START nového klienta (identifikátor 2). Následne zmaže správu, v ktorej mu bol oznámený identifikátor nového klienta.

8;1;2007-04-09 07:30:44.994;1;sync;1;;;;;;;0;operation destroy
Klient vymaže správu (identifikátor 1), takže v inicializácii už nebude robiť ďalšie operácie.

4;1;2007-04-09 07:30:44.995;1;;;;;;;;;;client disconnect - client side disconnect
Následne sa klient odpojí (kvôli jednoduchosti implementácie) a fyzicky vytvorí druhého klienta.

3;1;2007-04-09 07:30:45.013;1;;127.0.0.1;1765;;;;;;;client connect
Potom sa znovu pripojí k serveru a začne počiatočnú synchronizáciu.

7;1;2007-04-09 07:30:45.023;1;sync;10;0x53,0x59,0x4e,0x43,0x2d,0x49,0x4e,∼
0x49,0x54,0x00;3;;;;;0;operation create
Vytvorí správu so špecifickým textom ("SYNC-INIT") a kým túto správu (identifikátor 3) nezmaže, tak všetky jeho operácie sú súčasťou počiatočnej synchronizácie.

7;1;2007-04-09 07:30:45.029;1;sync;6;0x53,0x54,0x41,0x52,0x54,0x00;4;;;;;0;∼
operation create
9;3;2007-04-09 07:30:45.030;1;async;4;1;1;;;;;0;operation send
10;4;2007-04-09 07:30:45.031;1;sync;2;1,2;;2;5;;19;0;operation recv
8;1;2007-04-09 07:30:45.032;1;sync;5;;;;;;;0;operation destroy
Počiatočná synchronizácia (prvý klient).

3;1;2007-04-09 07:30:45.035;2;;127.0.0.1;1768;;;;;;;client connect
Pripojenie druhého klienta.

10;1;2007-04-09 07:30:45.132;1;sync;2;1,2;;1;6;4;6;0;operation recv
Počiatočná synchronizácia (prvý klient).

7;1;2007-04-09 07:30:45.134;2;sync;10;0x53,0x59,0x4e,0x43,0x2d,0x49,0x4e,∼
0x49,0x54,0x00;7;;;;;0;operation create
Druhý klient vytvorí správu so špecifickým textom ("SYNC-INIT") a kým túto správu (identifikátor 7) nezmaže, tak všetky jeho operácie sú súčasťou počiatočnej synchronizácie.

8;1;2007-04-09 07:30:45.135;1;sync;6;;;;;;;0;operation destroy
10;5;2007-04-09 07:30:45.136;2;sync;1;1;;;8;;10;0;operation recv
7;1;2007-04-09 07:30:45.137;1;sync;6;0x53,0x54,0x41,0x52,0x54,0x00;9;;;;;0;∼

operation create
9;1;2007-04-09 07:30:45.138;1;sync;9;1;2;2;8;;;0;operation send
Počiatočná synchronizácia (obaja klienti).

8;1;2007-04-09 07:30:45.139;1;sync;3;;;;;;;0;operation destroy
Prvý klient vymaže správu (identifikátor 3), takže končí s počiatočnou synchronizáciou.

7;1;2007-04-09 07:30:45.140;1;sync;14;0x00,0x00,0x00,0x00,0x31,0x32,0x33,∼
0x34,0x35,0x36,0x37,0x38,0x39,0x30;10;;;;;0;operation create
Prvý klient (odosielateľ, identifikátor 1, identifikátor v TPL 0) vytvorí pomocou operácie CREATE na serveri správu (identifikátor 10).

8;1;2007-04-09 07:30:45.141;2;sync;8;;;;;;;0;operation destroy
Počiatočná synchronizácia (druhý klient).

8;1;2007-04-09 07:30:45.141;2;sync;7;;;;;;;0;operation destroy
Druhý klient vymaže správu (identifikátor 7), takže končí s počiatočnou synchronizáciou.

9;3;2007-04-09 07:30:45.142;1;async;10;1;2;;;;;0;operation send
Prvý klient odošle vytvorenú správu (identifikátor 10) pomocou operácie SEND druhému klientovi. Keďže druhý klient ešte nevložil do systému operáciu RECV, ktorá túto správu prijme, tak je vykonanie operácie SEND odložené na neskôr. Týmto práca prvého klienta skončila a môže začať robiť záverečnú synchronizáciu a ukončiť svoju prácu.

10;1;2007-04-09 07:30:45.143;2;sync;2;1,2;;1;11;10;14;0;operation recv
Druhý klient (prijímateľ, identifikátor 2, identifikátor v TPL 1) prijme pomocou operácie RECV od prvého klienta správu (identifikátor 10) a vytvorí si jej kópiu (identifikátor 11). Zároveň sa vykoná operácia SEND a správa s identifikátorom 10 sa vymaže.

7;1;2007-04-09 07:30:45.144;1;sync;9;0x53,0x59,0x4e,0x43,0x2d,0x45,0x4e,∼
0x44,0x00;12;;;;;0;operation create
Prvý klient vytvorí správu so špecifickým textom ("SYNC-END") a kým túto správu (identifikátor 12) nezmaže, tak všetky jeho operácie sú súčasťou záverečnej synchronizácie.

8;1;2007-04-09 07:30:45.146;2;sync;11;;;;;;;0;operation destroy

Druhý klient prijatú správu (identifikátor 11) pomocou operácie DESTROY na serveri vymaže. Týmto práca druhého klienta skončila a môže začať robiť záverečnú synchronizáciu a ukončiť svoju prácu.

```
10;5;2007-04-09 07:30:45.147;1;sync;1;2;;;13;;9;0;operation recv
```
Záverečná synchronizácia (prvý klient).

```
7;1;2007-04-09 07:30:45.148;2;sync;9;0x53,0x59,0x4e,0x43,0x2d,0x45,0x4e,∼
  0x44,0x00;14;;;;;0;operation create
```
Druhý klient vytvorí správu so špecifickým textom ("SYNC-END") a kým túto správu (identifikátor 14) nezmaže, tak všetky jeho operácie sú súčasťou záverečnej synchronizácie.

```
7;1;2007-04-09 07:30:45.149;2;sync;15;0x52,0x45,0x51,0x55,0x45,0x53,0x54,∼
  0x5f,0x46,0x49,0x4e,0x49,0x53,0x48,0x00;15;;;;;0;operation create
9;1;2007-04-09 07:30:45.150;2;async;15;1;1;1;13;;;0;operation send
10;5;2007-04-09 07:30:45.151;2;sync;1;1;;;16;;15;0;operation recv
8;1;2007-04-09 07:30:45.152;1;sync;13;;;;;;;0;operation destroy
7;1;2007-04-09 07:30:45.153;1;sync;7;0x46,0x49,0x4e,0x49,0x53,0x48,0x00;∼
  17;;;;;0;operation create
9;1;2007-04-09 07:30:45.154;1;sync;17;1;2;2;16;;;0;operation send
8;1;2007-04-09 07:30:45.155;2;sync;16;;;;;;;0;operation destroy
```
Záverečná synchronizácia (obaja klienti).

```
8;1;2007-04-09 07:30:45.157;2;sync;14;;;;;;;0;operation destroy
```
Druhý klient vymaže správu (identifikátor 14), takže končí so záverečnou synchronizáciou.

```
6;1;2007-04-09 07:30:45.157;2;sync;;;;;;;;0;operation end
```
Druhý klient ukončuje pomocou operácie END svoju prácu...

```
4;1;2007-04-09 07:30:45.157;2;;;;;;;;;;client disconnect - end operation disconnect
```
...a je na základe toho odpojený od servera.

```
8;1;2007-04-09 07:30:45.158;1;sync;12;;;;;;;0;operation destroy
```
Prvý klient vymaže správu (identifikátor 12), takže končí so záverečnou synchronizáciou.

```
4;1;2007-04-09 07:30:45.259;1;;;;;;;;;;client disconnect - client side disconnect
```
Prvý klient sa odpája od servera — nerobí to pomocou operácie END, pretože

klient s identifikátorom 1 má špeciálne postavenie (ostatní klenti sa odpájajú pomocou operácie END).

2;1;2007-04-09 07:30:46.893;;;;;;;;;;server shutdown
Ukončenie práce servera.
Tu končí záznam priebehu výmeny správ.

5.2.2 Detekcia chýb pomocou analyzátora

Chyby, ktoré nastali v priebehu výmeny správ môžu byť v podstate dvoch druhov:

- Za prvé sú to systémové chyby zapríčinené napríklad nedostatkom systémových prostriedkov (málo voľnej pamäťe, ...), chybou pri prenose dát po sieti, atď. Tieto chyby sú z pohľadu ladenia paralelných programov nezaujímavé a pri korektnom behu samotného systému by sa nemali vyskytovať.
- Za druhé sú to programátorské chyby zapríčinené nekorektnou implementáciou programu. Tieto chyby sa snažíme ladením programov nájsť a eliminovať ich.

Na tomto mieste uvediem chyby, ktoré analyzátor odhaľuje, ukážky programov, ktoré ich spôsobujú a ako sa daná chyba prejaví v logu.

Chyba: Poslanie správy klientovi (resp. prijatie správy od klienta), ktorý nie je pripojený k serveru.

Programová ukážka:

```
ret = initialize_clients(2, &clients, argv);
if ( ret == 0 ) return -1;
/* inicializujem dvoch klientov */

client_pid = clients[1] + 1;
/* client_pid je identifikátor nepripojeného klienta */
if ( ret == 1)
{
  /* jeden klient pošle správu nepripojenému klientovi */
  mps_create("test", 5, &mid);
  mps_send(mid, 1, &client_pid);
}
```

```
else
{
  /* druhý klient prijme správu od nepripojeného klienta */
  mps_recv(&msg, &msglen, &mid, &pid, 1, &client_pid, NULL);
  mps_destroy(mid);
  free(msg);
}

free(clients);
deinitialize_client();
return 0;
```

Ukážka logu:

```
7;1;2007-04-15 14:52:31.328;1;sync;5;0x74,0x65,0x73,0x74,0x00;10;;;;;0;~
  operation create
9;2;2007-04-15 14:52:31.343;1;sync;10;1;3;3;;;;0;operation send
```

Chyba sa prejaví tak, že hodnota stĺpca *resultid* v operácii SEND je dva.

```
10;4;2007-04-15 14:52:31.343;2;sync;1;3;;3;11;;19;0;operation recv
8;1;2007-04-15 14:52:31.343;2;sync;11;;;;;;;0;operation destroy
```

Chyba sa prejaví tak, že hodnota stĺpca *resultid* v operácii RECV je štyri.

Túto chybu možno takisto odhaliť tak, že sa pri čítaní logu udržiava zoznam klientov pripojených k serveru a daný prijímateľ resp. odosielateľ sa v zozname nenachádza.

Chyba: Vytvorenie / prijatie správy, ktorá nebola neskôr zmazaná / odoslaná.

Programová ukážka:

```
ret = initialize_clients(1, &clients, argv);
if ( ret == 0 ) return -1;
/* inicializujem jedného klienta */

mps_create("test", 5, &mid);
/* vytvorím správu, ale nič s ňou nebudem robiť */

free(clients);
```

```
deinitialize_client();
return 0;
```

Ukážka logu:

```
7;1;2007-04-15 15:23:40.046;1;sync;5;0x74,0x65,0x73,0x74,0x00;1;;;;;0;∼
  operation create
4;1;2007-04-15 15:23:40.156;1;;;;;;;;;client disconnect - client side disconnect
```

Chyba sa dá odhaliť tak, že sa pri čítaní logu udržiava pre klienta zoznam správ, ktoré má vo svojej oblasti a keď je tento zoznam pri odpájaní klienta od servera neprázdny, tak nastala táto chyba.

Chyba: Odoslanie / zmazanie správy, ktorú klient nevytvoril / neprijal.

Programová ukážka:

```
ret = initialize_clients(2, &clients, argv);
if ( ret == 0 ) return -1;
/* inicializujem dvoch klientov */

mid = 123;
/* mid je identifikátor neexistujúcej správy */
if ( ret == 1)
{
  /* jeden klient posiela neexistujúcu správu */
  mps_send(mid, 1, &(clients[1]));
}
else
{
  /* druhý klient maže neexistujúcu správu */
  mps_destroy(mid);
}

free(clients);
deinitialize_client();
return 0;
```

Ukážka logu:

```
9;0;2007-04-15 15:56:31.828;1;sync;123;1;2;;;;;-2;∼
```

operation send - error executing operation

Chyba sa prejaví tak, že hodnota stĺpca *resultid* v operácii SEND je nula a hodnota stĺpca *errorid* je mínus dva.

Ukážka logu:

8;0;2007-04-15 15:56:31.828;2;sync;123;;;;;;;-2;∼
operation destroy - error executing operation

Chyba sa prejaví tak, že hodnota stĺpca *resultid* v operácii DESTROY je nula a hodnota stĺpca *errorid* je mínus dva.

Táto chyba sa dá takisto odhaliť tak, že sa daná správa nenachádza v zozname správ v oblasti klienta (viď predchádzajúca chyba).

Chyba: Poslanie správy, ktorú nikto neprijme resp. prijatie správy, ktorú nikto neodošle. T.j. operácie SEND bez združenej operácie RECV resp. operácie RECV bez združenej operácie SEND.

Pri tejto chybe je rozdiel keď program komunikuje pomocou knižnice TPL (na konci sa robí bariérová synchronizácia), alebo keď komunikuje mojimi funkciami (na konci sa nerobí nič).

Programová ukážka (moje funkcie):

```
ret = initialize_clients(3, &clients, argv);
if ( ret == 0 ) return -1;
/* inicializujem troch klientov */

if ( ret == 1)
{
  /* prvý klient pošle správu tretiemu klientovi */
  mps_create("test", 5, &mid);
  mps_send(mid, 1, &(clients[2]));
}
else if ( ret == 2)
{
  /* druhý klient prijme správu od tretieho klienta */
  mps_recv(&msg, &msglen, &mid, &pid, 1, &client_pid, NULL);
  mps_destroy(mid);
```

```
  free(msg);
}
/* tretí klient nerobí nič */

free(clients);
deinitialize_client();
return 0;
```

Ukážka logu:

```
7;1;2007-04-16 07:35:13.281;1;sync;5;0x74,0x65,0x73,0x74,0x00;14;;;;;0;~
  operation create
9;3;2007-04-16 07:35:13.281;1;sync;14;1;3;;;;;0;operation send
10;5;2007-04-16 07:35:13.281;2;sync;1;3;;;15;;5;0;operation recv
6;1;2007-04-16 07:35:13.281;3;sync;;;;;;;;0;operation end
4;1;2007-04-16 07:35:13.281;3;;;;;;;;;;client disconnect - end operation disconnect
11;2;2007-04-16 07:35:13.281;3;;1;;14;;;;;0;execute send/recv
11;4;2007-04-16 07:35:13.281;2;;3;15;;19;;;;0;execute send/recv
8;1;2007-04-16 07:35:13.281;2;sync;15;;;;;;;0;operation destroy
```

Chyba sa prejaví tak, že po odpojení tretieho klienta sa vykonajú operácie RECV, pre ktoré je on odosieľatelom a operácie SEND, pre ktoré je on prijímateľom — samozrejme sa vykonajú neúspešne (viď prvá spomenutá chyba).

Túto chybu možno takisto odhaliť tak, že sa pri čítaní logu udržiava pre klienta zoznam jeho operácií SEND a RECV, ktoré ešte neboli vykonané, a keď je tento zoznam pri odpájaní klienta od servera neprázdny, tak nastala táto chyba.

Pri použití knižnice TPL je rozdiel pri posielaní správy a pri prijímaní správy.

Pri poslaní správy, ktorú nikto neprijme (v programe) sa žiadna chyba neprejaví, pretože pri vykonávaní záverečnej synchronizácii sa správa prijme a odignoruje sa. Jediný spôsob ako túto chybu zistiť je rozlišovať pri čítaní logu, či klient robí synchronizáciu alebo normálnu komunikáciu. Potom ak jeden klient pošle správu pri normálnej komunikácii a druhý klient ju prijme pri synchronizácii tak nastala chyba.

Pri prijímaní správy, ktorú nikto nepošle (v programe) nastane zablokovanie systému (*deadlock*), pretože jeden proces čaká na správu od druhého, ale ten

už robí synchronizáciu a čaká na sychronizačnú správu od prvého. Chyba sa dá odhaliť iba tak, že keď klient, ktorý robí normálnu komunikáciu čaká na správu od klienta, ktorý robí synchronizáciu, tak nastala táto chyba.

Chyba: Zablokovanie systému (*deadlock*)

Túto chybu analyzátor neodhaľuje, pretože to nie je (vo všeobecnosti) možné. Potenciálny *deadlock* je možné hľadať tak, že sa spraví orientovaný graf klientov, ktorí čakajú na iného klienta. Ak sa v tomto grafu vyskytne cyklus, tak to môže, ale nemusí byť *deadlock*. *Deadlock* to nemusí byť v prípade ak klient, ktorý je súčasťou cyklu má viac ako jedno vlákno, ktorým komunikuje, takže vlákno, ktoré nie je súčasťou cyklu môže cyklus prerušiť. Druhá možnosť kedy to nemusí byť *deadlock* je, ak klient ktorý je súčasťou cyklu nečaká na správu iba od klienta, ktorý je tiež súčasťou cyklu ale čaká na správu aj od klienta, ktorý súčasťou cyklu nie je — tento klient môže cyklus prerušiť.

Programová ukážka:

```
ret = initialize_clients(2, &clients, argv);
if ( ret == 0 ) return -1;
/* inicializujem dvoch klientov */

if ( ret == 1)
{
  /* prvý klient čaká na správu od druhého klienta */
  mps_recv(&msg, &msglen, &mid, &pid, 1, &(clients[1]), NULL);
  mps_destroy(mid);
  free(msg);
}
else
{
  /* druhý klient čaká na správu od prvého klienta */
  mps_recv(&msg, &msglen, &mid, &pid, 1, &(clients[0]), NULL);
  mps_destroy(mid);
  free(msg);
}

free(clients);
deinitialize_client();
return 0;
```

Ukážka logu:

```
10;5;2007-04-16 09:12:34.375;1;sync;1;2;;;10;;6;0;operation recv
10;5;2007-04-16 09:12:34.375;2;sync;1;1;;;11;;6;0;operation recv
```

Poznámka: Niektoré chyby môžu nastať aj pri korektnej inicializácii / synchronizácii. Tieto chyby analyzátor odhalí, ale užívateľovi ich nezobrazí.

5.3 Rozšírenie systému

V tejto časti sa venujem možnostiam rozšírenia/vylepšenia implementovaného systému v jednotlivých častiach.

5.3.1 Server

Sledovanie za behu (*monitoring*):
Rozšíriť systém o sledovanie za behu je relatívne jednoduchá záležitosť. Treba na to spraviť nasledovné tri veci:

- Zadefinovať v serveri potrebné štruktúry/premenné pre udalosti, ktoré chceme sledovať.
- Ak nastane nejaká udalosť, tak dané premenné aktualizovať (toto je triviálne, pretože pri udalostiach sa už vytvára záznam priebehu výmeny správ — stačí ho rozšíriť o aktualizovanie premenných).
- Sprostredkovať získané informácie užívateľovi. Jeden variant je umožniť poskytovať tieto informácie inému programu cez ďalší *socket*. (Treba samozrejme zadefinovať na to ďalší protokol.)

Simulácia rôznych rýchlostí prenosových liniek medzi procesmi:
Simulovať rôzne rýchlosti prenosových liniek medzi procesmi je možné viacerými spôsobmi, ale asi najjednoduchší a najrozumnejší je nasledovný spôsob. Pre každú dvojicu procesov (zdroj a cieľ) sa zadefinuje časové zdržanie — latencia siete. Potom pri vykonávaní dvojice operácií $SEND/RECV$ sa odloží poslanie odpovedí na daných klientov (signalizovanie semaforov, ...) tak, aby čas medzi prečítaním operácie $SEND$ a odoslaním odpovedí bol aspoň taký ako definované časové zdržanie. Toto sa dá spraviť dvoma spôsobmi:

- Vykonať zdržanie v tom vlákne, ktoré vykonáva dané operácie. Toto je jednoduché na implementáciu, ale nevýhodou je že sa tým potencionálne zdrží vykonávanie ďalších operácií (ak sú na serveri a má ich vykonať dané vlákno).

- Vytvoriť nové vlákno, ktoré sa bude starať o zasielanie odpovedí na klientov tak, aby bolo dodržané zadefinované časové zdržanie. Toto je zložitejšie na implementáciu, ale nebude sa tým zdržovať vykonávanie iných operácií.

Možnosť pozastaviť a krokovať prácu servera:
Implementácia pozastavenia a krokovania práce servera už nie je jednoduchá záležitosť. Ako prvé treba implementovať samotné pozastavenie a krokovanie. To znamená:

- Zadefinovať stavové premenné, ktorými sa bude server riadiť pri práci.
- Implementovať pozastavenie servera podľa zadefinovaných stavových premenných.
- Umožniť užívateľovi prístup k stavovým premenným. Konkrétne pomocou komunikácie s iným programom cez ďalší *socket* a ďalší protokol.
- Naprogramovať program, ktorým bude užívateľ riadiť prácu servera.

Okrem toho treba ešte umožniť automaticky pozastaviť server podľa užívateľsky zadaných parametrov. Čiže:

- Umožniť užívateľovi definovať počas behu body prerušenia (*breakpoint*). T.j. definovať masku na operácie a na správy, na ktorých sa má server pozastaviť. Takisto umožniť tieto masky ukladať na disk (načítavať z disku).
- V serveri počas behu testovať všetky operácie a správy, či súhlasia s niektorou maskou a prípadne pozastaviť server.

Toto všetko ale umožní iba pasívne sledovať prácu systému. Takže ďalší logický krok je umožniť interaktívnu manipuláciu s operáciami a so správami (pridávanie, prepisovanie, mazanie) počas behu na serveri.

Načítavanie nastavení zo súboru:
Server sa v súčasnom stave riadi stavovými premennými, ktorých obsah je naplnený natvrdo v programe a nemení sa (napr. číslo portu na ktorom server počúva, ...). Takže by bolo asi vhodné doprogramovať načítavanie týchto premenných zo súboru, aby sa dala práca servera prispôsobovať bez rekompilácie programu. Obdobne to platí aj pre klienta (číslo portu na ktorý sa klient pripája, IP adresa na ktorú sa klient pripája, ...).

Optimalizácia:
Pri práci s centralizovaným systémom nie je kladený dôraz na rýchlosť resp. na pamäťové nároky, pretože je určený skôr na ladenie programov a nie na ich rýchle/efektívne vykonávanie (to je skôr úlohou distribuovaného systému). Napriek tomu by sa v mojej implementácii našli miesta, ktoré je možné optimalizovať. Hlavne ide o manipuláciu s operáciami a správami resp. ich ukladanie/hľadanie/mazanie do/v/z oblasti procesu.

5.3.2 Knižnica TPL

Zmena prideľovania identifikátorov klientom na serveri:
Ak by sa mal môj systém používať na ladenie programov v TPL, tak by bolo vhodné zmeniť prideľovanie identifikátorov klientom na serveri, aby boli identifikátory v mojom systéme a identifikátory v TPL (zhruba) rovnaké. Bude prehľadnejšie pre užívateľa, a tiež to sprehľadní a mierne urýchli systém. Keďže 0 nemôže byť identifikátor (je to špeciálna hodnota), tak by mali byť identifikátory klientov v rozsahu 1..počet procesov. Tým sa eliminuje použitie prevodných tabuliek medzi identifikátormi a prevod bude znamenať pripočítanie resp. odpočítanie jednotky. (Tu sa myslí hlavne urýchlenie prevodu identifikátora v mojom systéme na identifikátor v TPL, pretože tento prevod je v súčasnej implementácii logaritmický — binárne vyhľadávanie v utriedenom zozname.)

Konverzia formátu dát:
Ako som už spomínal pri popise implementácie knižnice TPL, pri čítaní a zapisovaní správ sa nerobí žiadna konverzia formátu základných dátových typov (napr. konverzia kódovania typu *integer* z *little-endian* na *big-endian*), keďže to v súčasnosti nie je potrebné. Toto sa v budúcnosti môže zmeniť, takže potom bude treba konverziu formátu dát pri čítaní a zapisovaní správ robiť. Jedna z možností je na to použiť knižnicu XDR.

5.3.3 Analyzátor záznamu priebehu výmeny správ (logu)

Štatistika výmeny správ medzi jednotlivými procesmi:
V súčasnom stave počíta analyzátor logu štatistické údaje správ iba pre jednotlivé procesy (t.j. štatistika všetkých správ, ktoré proces poslal/prijal). Pre niektoré účely by bolo vhodné počitať aj štatistické údaje správ medzi dvoma procesmi (t.j. štatistika všetkých správ, ktoré proces poslal konkrétnemu procesu resp. prijal od konkrétneho procesu).

Grafický výstup:
Pre lepšiu prezentáciu logu užívateľovi, by bolo vhodné spraviť grafické znázornenie priebehu výmeny správ. V čase písania tejto práce beží ročníkový projekt, ktorého cieľom je vytvorenie interaktívneho prezentačného programu [Bar07].

6 Záver

Cielmi mojej práce boli špecifikácia, návrh a implementácia centralizovaného komunikačného systému. Tieto ciele sa mi podarilo splniť — špecifikácia je popísaná v kapitole 2, návrh je popísaný v kapitole 4 a implementácia je súčasťou príloh tejto práce. Ďalej táto práca obsahuje implementáciu knižnice TPL pomocou mnou navrhnutého rozhrania systému. Táto implementácia je popísaná v kapitole 5, konkrétne v časti 5.1.

Hoci je moja implementácia plne funkčná, neznamená to že je konečná. Možným rozšíreniam a vylepšeniam sa venujem v časti 5.3.

Implementácia rozhrania TPL pomocou mnou navrhnutých funkcií bola priamočiara, t.j. preklad žiadnej funkcie TPL nevyžadoval použitie neefektívnych programovacích techník či použitie dodatočnej pamäte. Inými slovami, simulácia funkcií TPL mojimi funkciami je "rozumná" v zmysle *invariance thesis* [vEB90] (hoci formálny dôkaz pravdepodobne presahuje rámec tejto práce). Optimalizácie uvedené v časti 5.3 nie sú v skutočnosti príliš významné.

V časti 5.2 sa ďalej venujem hľadaniu programátorských chýb za použitia môjho systému a analyzátora logu. V distribuovanom systéme je odhaľovanie programátorských chýb zvyčajne veľmi obtiažne. Môj analyzátor odhalí väčšinu typických chýb automaticky. Avšak aj pri chybách, ktoré sa vo všeobecnosti automaticky odhaliť nedajú (napríklad *deadlock*), môže programátorovi záznam udalostí v logu výrazne pomôcť.

Funkcionalita systému (a implementácie knižnice TPL) bola okrem iného overená aj úspešným skompilovaním a spustením existujúcich programov využívajúcich rozhranie TPL. Niektoré tieto programy sú súčasťou príloh tejto práce. Všetky programy bežali v mojom systéme korektne, t.j. výstup žiadneho programu sa napriek serializácii nelíšil od výstupu toho programu bežiacom na viacprocesorovom počítači (samozrejme s výnimkou údajov súvisiacich s meraním času).

Portabilita implementácie bola overená úspešným skompilovaním a spustením v rôznych operačných systémoch a na rôznych hardvérových platformách. Konkrétne:

- OS: Windows, platforma: x86, kompilátor: MinGW (+ knižnica Pthreads)
- OS: Windows, platforma: x86, kompilátor/knižnica: Cygwin

- OS: Linux, platforma: x86, kompilátor: gcc
- OS: Linux, platforma: ARM big-endian, kompilátor: gcc
- OS: Solaris, platforma: x86, kompilátor: gcc

7 Slovník pojmov a skratiek

Táto príloha obsahuje abecedne zoradený zoznam skratiek a pojmov, ktoré sa objavujú v rámci práce, spolu s popisom ich významu.

7.1 Zoznam skratiek

Skratka	Význam skratky
POSIX	*P*ortable *O*perating *S*ystem *I*nterface for uni*X* je kolektívne meno pre rodinu príbuzných štandardov, ktoré definujú aplikačné programovacie rozhranie pre softvér kompatibilný s variantami operačného systému UNIX. Napriek názvu je POSIX štandard aplikovateľný na ľubovolný operačný systém, nie len pre UNIX.
SDL	*S*imple *D*irectMedia *L*ayer je otvorená, multiplatformová knižnica, ktorá vytvára abstrakciu nad grafickými, zvukovými, vstupnými, ... aplikačnými programovacími rozhraniami rôznych platforiem.
SSH	*S*ecure *Sh*ell je množina štandardov a sieťový protokol, ktorý umožňuje vytvoriť bezpečný komunikačný kanál medzi lokálnym a vzdialeným počítačom. (Okrem iných vecí umožnuje prenášať súbory medzi lokálnym a vzdialeným počítačom a spúštať príkazy na vzdialenom počítači.)
UTC	*U*niversal *T*ime, *C*oordinated je čas definovaný na základe medzinárodného atómového času (je nezávislý na rotácii Zeme). Skratka vznikla ako kompromis medzi anglickou skratkou CUT (co-ordinated universal time) a francúzskou skratkou TUC (temps universel coordonné).
XDR	e*X*ternal *D*ata *R*epresentation je štandard a knižnica, ktorá umožňuje obaliť dáta platformovo nezávislým spôsobom, aby sa dali prenášať medzi heterogénnymi počítačovými systémami.

7.2 Slovník pojmov

Pojem	Význam pojmu
ANSI C	Taktiež ISO C alebo Standard C. Implementácia programovacieho jazyka C, ktorá spĺňa štandardnú špecifikáciu jazyka C.

Bariérová synchronizácia	Typ synchronizačnej metódy. Bariéra pre skupinu vlákien alebo procesov znamená, že každé vlákno/proces sa musí v tomto bode zastaviť a nemôže pokračovať pokým všetky ostatné vlákna/procesy nedosiahnu bariéru.
Bod prerušenia	Breakpoint — používa sa na úmyselné pozastavenie práce programu za účelom ladenia. Počas prerušenia preskúma programátor testovacie prostredie (logy, pamäť, súbory, ...), aby zistil či program funguje podľa jeho predstáv.
C++	Objektovo-orientovaný programovací jazyk. Rozšírenie programovacieho jazyka C.
Cygwin	Port kompilátora gcc pre operačný systém Windows. Súčasťou (nutnou k behu programov) je knižnica cygwin1.dll, ktorá implementuje aplikačné rozhranie POSIX.
Delphi	Objektovo orientovaný programovací jazyk založený na jazyku Pascal a vývojové prostredie, vytvorený spoločnosťou Borland.
Free Pascal	Otvorený, multiplatformový kompilátor jazyka programovacieho jazyka Pascal.
.NET	Softvérové prostredie vytvorené spoločnosťou Microsoft. Podporuje viacero programovacích jazykov, ktorých kód sa kompiluje do platformovo-nezávislého medzijazyka (bytecode).
Java	Objektovo-orientovaný programovací jazyk, vytvorený spoločnosťou Sun Microsystems. Je to platformovo-nezávislý jazyk kompilovaný do tzv. bytecode. Syntaxou podobný jazyku C++.
JavaScript	Skriptovací programovací jazyk, vytvorený spoločnosťou Netscape Communications. Je to platformovo-nezávislý jazyk. Syntaxou podobný jazyku C.
Lazarus	Otvorená, multiplatformová sada knižníc pre Free Pascal ktorá emuluje Delphi.
MinGW	Minimalist GNU for Windows — port kompilátora gcc pre operačný systém Windows. K behu programov nevyžaduje žiadne knižnice (okrem knižníc systému Windows). T.j. nepodporuje aplikačné rozhranie POSIX.
Socket	Myslí sa TCP socket — koncový bod sieťovej komunikácie.
Vlákno	Thread, tok riadenia v procese. (Vo všeobecnosti môže mať proces niekoľko nazávislých tokov riadenia, ktoré zdieľajú pamäť toho procesu.)

wxWidgets	Otvorená, multiplatformová knižnica pre budovanie grafického prostredia, pre prístup k databázam cez ODBC, pre sieťovú komunikáciu, ...
Zablokovanie systému	Deadlock — situácia, keď dve alebo viac konkurenčných akcií čaká na vykonanie tej druhej, takže sa nevykoná ani jedna z nich.

8 Prílohy

K práci je priložené CD so zdrojovými kódmi, skompilovaná verzia pre Windows (kompilátor MinGW) a elektronická verzia tejto práce.

Adresárová štruktúra a popis:
/bin/ — skompilované verzie programov
/bin/Windows (MingW)/ — verzia pre Windows (kompilátor MinGW)
/lib/ — knižnice potrebné pre kompiláciu a spúštanie
/lib/pthreads-win32/ — implementácia knižnice pthreads pre Windows
/src/ — zdrojové kody
/src/MPSClient/ — klientská knižnica
/src/MPSLogAnalyzer/ — analyzátor logu
/src/MPSServer/ — server
/src/TPL/ — programy využívajúce rozhranie TPL
/src/TPL/LIBWORK/ — knižnica, ktorú používa program PI
/src/TPL/PI/ — program PI (popis nižšie)
/src/TPL/SENDRECV/ — program SENDRECV
/src/TPL/THR_PINGPONG/ — program THR_PINGPONG
/text/ — elektronická verzia tejto práce

Popis programov využívajúcich rozhranie TPL:

SENDRECV:
Program má dva parametre — počet správ a veľkosť správy. Program vytvorí dvoch klientov. Jeden klient pošle druhému zadaný počet správ (zadanej veľkosti). Druhý klient prijme zadaný počet správ.

THR_PINGPONG:
Program má dva parametre — počet správ a veľkosť správy. Program vytvorí dvoch klientov. Každý z nich vytvorí dve vlákna. V jednom vlákne pošle tomu druhému klientovi zadaný počet správ (zadanej veľkosti). V druhom vlákne prijme od druhého klienta zadaný počet správ.

PI:
Program počíta číslo π pomocou integrálu ($\frac{\pi}{4} = \int \frac{dx}{1+x^2}$). Program má dva parametre — počet klientov a počet dielov integrálu. Prvý klient prideľuje prácu ostatným klientom a ráta súčet vypočítaných dielov integrálu. Ostatní klienti počítajú pre pridelené diely čiastočné sumy integrálu a výsledok pre každý pridelený diel posielajú prvému klientovi.

Referencie

[AIM] *AIMS — Automated Instrumentation and Monitoring System.* http://www.nas.nasa.gov/Groups/Tools/AIMS/.

[AMQ] *Apache ActiveMQ.* http://activemq.apache.org/.

[Bar07] Laura Bartová. Ročníkový projekt, Univerzita Komenského, Fakulta matematiky, fyziky a informatiky, Bratislava, 2007.

[CG99] James Cownie and William Gropp. A standard interface for debugger access to message queue information in MPI. In Jack Dongarra, Emilio Luque, and Tomàs Margalef, editors, *PVM/MPI*, volume 1697 of *Lecture Notes in Computer Science*, pages 51–58. Springer, 1999.

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* Scientific and Engineering Computation. Second edition, November 1999.

[GLT99] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface.* Scientific and Engineering Computation. November 1999.

[Java] *Java.* http://java.sun.com/.

[Javb] *Javascript.* http://developer.mozilla.org/en/docs/About_JavaScript.

[JGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 295–308, New York, NY, USA, 1996. ACM Press.

[JMS] *JMS — Java Message Service.* http://java.sun.com/products/jms/.

[Laz] *Lazarus.* http://www.lazarus.freepascal.org/.

[MPIa] *MPI — Message Passing Interface.* http://www.mpi-forum.org/.

[MPIb] *MPICH2*.
http://www-unix.mcs.anl.gov/mpi/mpich/.

[NET] *.NET*.
http://msdn.microsoft.com/netframework/.

[OAQ] *Oracle Advanced Queuing*.
http://www.oracle.com/technology/products/aq/htdocs/aq9i_overview.html.

[p2d] *p2d2 — Portable Parallel/Distributed Debugger*.
http://www.nas.nasa.gov/Groups/Tools/p2d2/.

[Par94] Parsytec. *Parix V1.3 PowerPC Software Documentation*, 1994.

[Pla02] Tomas Plachetka. Perfect load balancing for demand-driven parallel ray tracing. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par*, volume 2400 of *Lecture Notes in Computer Science*, pages 410–419. Springer, 2002.

[Pla03] Tomas Plachetka. *Event-Driven Message Passing and Parallel Simulation of Global Illumination*. PhD thesis, University of Paderborn, Faculty of Electrotechnics, Informatics and Mathematics, 2003.

[Pla06] Tomas Plachetka. Unifying framework for message passing. In Jirí Wiedermann, Gerard Tel, Jaroslav Pokorný, Mária Bieliková, and Julius Stuller, editors, *SOFSEM*, volume 3831 of *Lecture Notes in Computer Science*, pages 451–460. Springer, 2006.

[Pol96] Stanislav Polák. Simulácia a monitorovanie transputerovských sietí. Diplomová práca, Univerzita Komenského, Fakulta matematiky, fyziky a informatiky, Bratislava, 1996.

[POS] *POSIX*.
http://www.opengroup.org/onlinepubs/009695399/toc.htm.

[PVM] *PVM — Parallel Virtual Machine*.
http://www.csm.ornl.gov/pvm/.

[Rep96] Michal Repa. Simulácia a debuggovanie transputerovských sietí. Diplomová práca, Univerzita Komenského, Fakulta matematiky, fyziky a informatiky, Bratislava, 1996.

[SDL] *SDL — Simple DirectMedia Layer.*
http://www.libsdl.org/.

[Tot] *TotalView.*
http://www.etnus.com/TotalView/.

[vEB90] Peter van Emde Boas. Machine models and simulation. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 1–66. 1990.

[WSA] *IBM WebSphere.*
http://www.ibm.com/websphere.

[wxW] *wxWidgets.*
http://www.wxwidgets.org/.