ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická

Diplomová práce

Simulátor linkové vrstvy Profibus protokolu

Vypracoval: *Karásek Vladimír*

Vedoucí DP: *Dr.Ing. Zdeněk Hanzálek*

Datum: *28. února 2004*

Praha, 2004

1 Anotace

Výukový simulátor je určen všem zájemcům o hlubší pohled do problematiky průmyslových sběrnic, konkrétně sběrnice Fieldbus. Účelem simulátoru je snaha usnadnit pochopení specifikace PROFIBUSU[1], konkrétně jeho linkové vrstvy. Odtud pochází název FDL (Field Data Link layer) simulátor. Studium protokolu ze specifikace vyžaduje soustředění, trpělivost a rovněž jistý stupeň znalosti z oboru. Tento nástroj tyto požadavky redukuje. Pomocí simulátoru studenti mohou na vlastně vytvořených příkladech sběrnice zjistit její chování v různých situacích. To zahrnuje sledování komunikace mezi uzly sítě a stav uzlů v libovolném okamžiku. Je zde možnost navodit chybový stav sběrnice např. deadlock nebo ztrátu tokenu.

2 Anotation

The Simulator of the FDL(Field Data Link) layer of PROFIBUS protocol is created for educational purposes. It is intended for those who desire extend their knowledge in the field of industrial communication. This simulator helps to understand the PROFIBUS specification [1]. Reading specification requires concentration, patience and a certain level of a field bus background. The simulator reduces all these demands on people willing to get through specification. Students can observe behavior of the bus in various situations. It includes state transfer of nodes and communication between nodes. It is possible to achieve error conditions of the bus, for example deadlock or token loss.

Obsah

Seznam obrázků

3 Prohlášení k DP

PROHLÁŠENÍ

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č.121/2000 Sb. , o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne ..... Podpis studenta:.....

4 Struktura diplomové práce

Úvodní kapitola 5 přináší všeobecně známá fakta o PROFIBUSU a jeho technickou charakteristiku. Dále je třeba udělat analýzu problému a vytýčit požadavky, kterých jsem se snažil v diplomové práci dosáhnout. To je náplní kapitoly 6. Popis řešení najdeme v kapitole 7. Sekce Stavy 7.1 a sekce Přechody 7.2 nejenom popisují chování sběrnice v dané situaci podle specifikace [1], ale rovněž uvádí, do jaké míry odpovídá simulace v daném stavu již zmiňované specifikaci. V dalších sekcích je popsán způsob programování, testování software a přípravy dokumentace. V závěru nemůžou chybět hodnocení a postřehy týkající se diplomové práce.

5 Úvod o sběrnici Profibus

Profibus je otevřený komunikační protokol, který vznikl v roce 1989 v Německu. Vytvořila ho skupina firem působící v oblasti automatizace. Teď je standardizován evropskou normou EN50170.

Technická specifikace:

- Rychlost sběrnice se pohybuje od 9.6 Kbps do 12 Mbps a je nepřímo úměrná délce vedení.
- Největší délka sběrnice až 4.8km s použitím opakovačů a 1.2km bez použití opakovačů. Mezi dvěma sousedními stanicemi může být použito nejvíce tři opakovačů.
- Podporované topologie jsou sběrnice, hvězda, kruh. První je nejrozšířenější.
- Přenos je asynchronní, základem je standard RS485, kroucená dvoulinka, aktivita signálu se určí rozdílem napětí na jednotlivých vodičích. UART přidává na začátku START bit (log. 0) pak následuje 8 informačních bitů a na závěr paritní bit a STOP bit (log. 1). Celkem tedy 11 bitů pro přenos jednoho bajtu. Po přenosu paketu zůstává sběrnice po dobu 33 bitů v klidu. Stav sběrnice logická 1.
- Na delší vzdálenosti se může použit optické vlákno. V nebezpečném prostředí pak fyzická vrstva odpovídá standardu IEC 1158-2.
- Kódování Non Return to Zero (NRZ), jednoduché kódování. Při posloupnosti stejných znaků úroveň signálu zůstává stejná jako při jediném znaku.

- Hammingová vzdálenost HD=4
- Maximální počet stanic 128
- Je implemntován na čipech ASIC, které jsou dodávané na trh několika výrobci.
- Typické oblasti použití: procesní automatizace, distribuované řídící systémy a aplikace běžící v reálném čase.

Profibus se snaží pokryt celé spektrum aplikací. K tomu má vytvořené **komunikační profily**, které se specializují na odlišné oblasti použití.

- **Profibus - DP (Decentralized Periphery)** Původně byl určen pro připojení několika jednoduchých zařízení, jako jsou senzory a akční členy k jednomu masteru. Je to situace, kdy jedna řídící stanice ovládá sběrnici a cyklický kontroluje své podřízené členy, posílá jim žádostí a přijímá data. V režimu multi-master je podobný FMS.
- **Profibus - FMS (Field Message Specification)** je víc zaměřený na distribuované aplikace s větším počtem inteligentních stanic (masterů). Tyto stanice tvoří logický kruh ve kterém si předávají token, právo vysílat na sběrnici.
- **Profibus - PA (Process Automation**) Typická oblast použití je řízení pomalejších procesů. Je vhodný do výbušného prostředí, má možnost napájení stanic ze sběrnice.
- PROFInet Protokol vytvořený za účelem komunikace Profibus zařízení na Ethernetové síti.

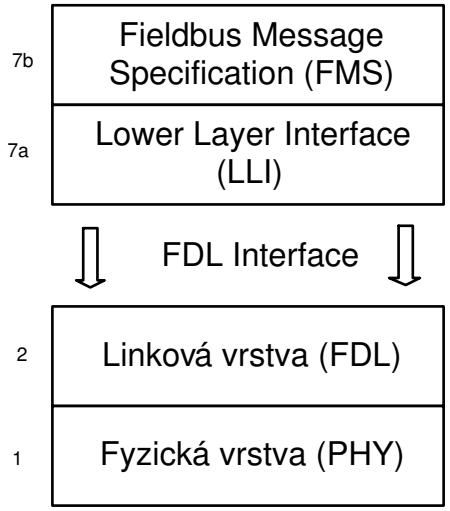
Výhody Profibusu oproti jiným sběrnicím:
Nejrozšířenější průmyslový komunikační protokol v Evropě. Z toho vyplývá nezávislost zákazníka na dodavatelích hardwaru. Produkty různých výrobců implementující daný protokol by měly být kompatibilní. Sběrnice je schopna přenášet relativně velké množství dat při vysokých rychlostech a na velké vzdálenosti. Široké spektrum použití v automatizaci.

Nevýhody Profibusu:
Jen u profilu PA napájení zařízení je součástí sběrnice. Adresa zařízení se nemůže měnit dynamicky, musí se nastavit ručně.

Profibus implementuje tři vrstvy ISO/OSI modelu:

- **Fyzická vrstva**. Většina její parametrů již byla popsaná v technické charakteristice výše.
- **Linková vrstva** definuje format paketů, zajišťuje integritu dat a řídí přístup k médiu. Nabízí vyšším vrstvám čtyři typy služeb. Tři jsou asynchronní a jedna cyklická, provádí tzv. polling.
  - SDA Send Data with Acknowledge
  - SDN Send Data with No Acknowledge
  - SRD Send and Request Data with Reply
  - CSRD Cyclic Send and Request Data with Reply
- **Aplikační vrstva** je dále rozdělená na Fieldbus Message Specification (FMS) a Lower Layer Interface (LLI)

Protože Profibus implementuje jen tyto tři vrstvy, postrádá schopnost směrování paketů. Propojení několika sítí musí být provedeno na aplikační úrovni.



Obrázek 1: ISO/OSI model Profibusu

5.1 Použité zkratky a dohody

LAS List of Active Stations, seznam stanic, které mohou být v logickém kruhu (seznam masterů)

Master Stanice, která má právo být v logickém kruhu. Dostává token. Tím je jí zaručeno právo vysílat na sběrnici po jistý časový interval. Tento pojem používám v počeštěné formě, takže skloňování tohoto slova nepovažují za prohřešek.

GAP Rozsah adres, na počátku kterého stojí TS a na konci NS.

TS This station. Adresa aktuální stanice.

NS Next station. Adresa následující aktivní stanice podle LAS.

Tr Target rotation time. Časový úsek, který vyjadřuje nejdelší dobu, za jakou token proběhne logický kruh.

Rr Real rotation time. Časový úsek, který vyjadřuje skutečnou dobu, za jakou token proběhne logický kruh.

Poll List Seznam stanic, které budou dotazované masterem, během jejich cyklické adresace pomocí paketu "Send and Request Data low"

V textu často užívám obecný termín stanice, node, případně master nebo slave. Ve většině případu se jedná o FDL kontrolér dané stanice.

6 Rozbor problematiky

6.1 Funkčnost simulátoru

Prvním problém, který musíme vyřešit je stanovit kompromis mezi náročnosti ovládání simulátoru a ochotou studentů vyzkoušet simulátor. Jedním z hlavních kritérii bude množství inicializačních parametrů, které uživatel bude muset zadávat před samotnou simulaci. Jako ideální se jeví možnost mít přednastavené parametry na výchozí hodnoty a poskytnout uživateli možnost jejich editace. Je nutné zanedbat druhořadé parametry aniž by utrpěla funkčnost systému. Simulátor by měl splňovat následující požadavky:

1. Možnost vytvoření vlastní sítě a její následná editace (např. libovolný počet masterů a slavů, pokud není v rozporu se specifikací)
2. Možnost konfigurace jednotlivých uzlů
3. Možnost nepřetržitého sledování stavů jednotlivých uzlů
4. Možnost simulace zasílání zprav mezi uzly
5. Zobrazení celkového průběhu simulace v čase
6. Automatické nebo ruční posouvání v časové oblasti a sice jak dopředu tak i dozadu
7. Možnost navodit chybový stav a sledovat, jak sběrnice na něj bude reagovat

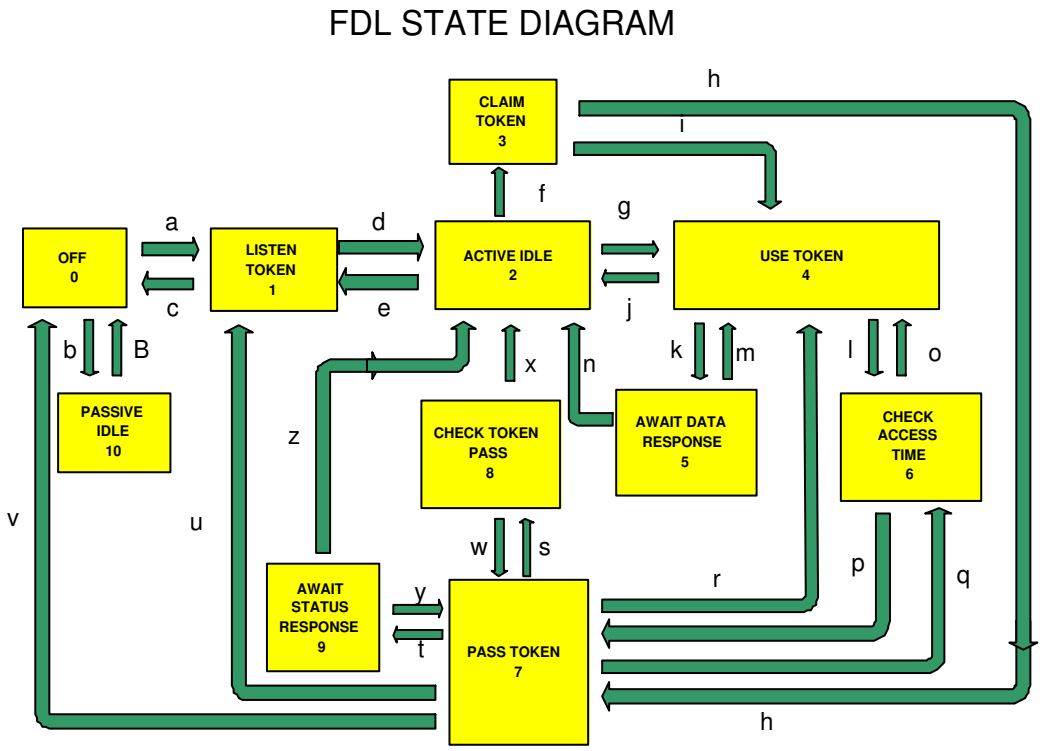
6.2 Prostředky realizace

Prostředky realizace simulátoru musí odpovídat naším záměrům. Cílem je zpřístupnit simulátor co nejširší vrstvě případných uživatelů. Je to problém, který musí řešit každý výrobce software. S tím rozdílem, že profesionální výrobci musí svůj software prodat, já chci jen nabídnout možnost snadnějšího studia. Výsledkem by měl být produkt nezávislý na platformě uživatele, který by nepůsobil problémy s instalací a měl by mít "user-friendly" ovládání. V dnešní době přístup k internetu již není problémem pro většinu lidí a pro studenty to platí dvojnásobně. Internet se jeví jako ideální cesta k šíření plodů své práce. Musíme jen vybrat vhodnou technologii, abychom zajistili již zmiňovanou nezávislost. Rozhodli jsme se pro programování simulátoru použit javovský applet. Neboť ten poběží v každém www prohlížeči podporujícím javu. Problém by mohl být pokud Java Virtual Machine běžící na

lokálním počítači byla staršího data výroby. Jelikož při programování appletu není důvod používat nejnovější vymoženosti jazyka Java, myslím si, že tento problém bude ve většině případu eliminován.

7 Řešení

Jádrem řešení je stavový diagram uzlů, na kterém jsou zobrazeny změny stavu a podmínky, které musí být splněné, aby se změna stavů uskutečnila. Tento stavový diagram je součástí specifikace [1] a je zobrazen na obr.2. Simulátor se zaměřuje na práci v režimu multi-master.



Obrázek 2: Stavový diagram stanice na úrovní linkové vrsrvy

7.1 Stavy

Offline V tomto stavu uzel nedostává pakety ani je nepřenáší (nereaguje na žádosti), do tohoto stavu se dostane zařízení po zapnutí.

Listen_Token V tomto stavu zařízení analyzuje token frames pakety a vytváří si seznam aktivních stanic na sběrnici (LAS List of Active Stations). Poté, co token proběhne dvě kola a LAS se nezmění, to znamená, že žádná stanice se nepřipojila ani se neodpojila od sběrnice, master je připraven vstoupit do dalšího stavu ActiveIdle. Pro tento přechod ještě

musí být splněno několik předpokladu. Nejprve musí dostat "Request FDL Status" požadavek od svého předchůdce v token ringu. Stanice musí zareagovat zprávou "Ready to enter logical ring" (Jsem připraven vstoupit do logického kruhu). A čekat dokud na sběrnici se neobjeví token frame určený dané stanici. Po obdržení tokenu přechází do ACTIVE_IDLE stavu.
Při generování LAS stanice neodpovídá ani ne potvrzuje příjetí požadávků.

Active_Idle V tomto stavu stanice odpovídá, případně potvrzuje "action frames" pakety jiných stanic. Pokud dostane token frame, ve kterém je adresa příjemce právě adresa této stanice, pak následuje přechod do USE_TOKEN stavu. Při vypršení time-outu, kdy stanice nezaregistrovala žádnou aktivitu na sběrnici, přechází do CLAIM_TOKEN stavu.

Claim_Token V tomto stavu stanice se snaží buď začít inicializaci logického kruhu od začátku nebo provést tzv. re-inicializaci. Rozdíl je v tom, že při re-inicializaci LAS a GAP jsou stále k dispozici a stanice tak může přejít přímo do USE_TOKEN stavu. Při úplné inicializaci musí nejprve stanice poslat sama sobě token a to dvakrát, aby ostatní stanice si do svých LASu zaregistrovaly danou adresu. Potom stanice musí vytvořit vlastní LAS a GAP, a to provede pomocí paketu FDL_STATUS_REQUEST, který posílá v AWAIT_STATUS_RESPONSE stavu.

Use_Token Tento stav je v programu poněkud pozměněn a neodpovídá striktně specifikaci [1]. Důvodem změny byla snaha zjednodušit rozhodovací proces v tomto stavu. Podle specifikace hlavní roli v tomto stavu hrají parametry Tr(TargetRotationTime) a Rr(RealRotationTime) a jejich rozdíl určí nám, zda ještě zbývá čas provést činnosti s nižšími prioritami nebo se to odloží na příště. Výpočet Tr a Rr zahrnuje několik parametrů, které na sebe nabalují další a vynucují zavedení a definici cyklu zprav s vysokou a nízkou prioritou. Vzorec můžeme najít ve specifikaci [1] na str.104. Nejprve jsem zavedl tyto parametry a počítal jsem je pomocí zjednodušených vzorců. Vzorcem je suma různých parametrů. Tím ,že jsem zanedbal nějaké členy tohoto součtu, výsledný parametr by byl zdrojem dalších chyb. Proto jsem vyřadil tyto parametry z rozhodovacího procesu. Výsledkem je fakt, že master postupně posílá DATA_REQUEST požadavky stanicím, které má v GAP listu a přechází do AWAIT_DATA_RESPONSE stavu.

Await_Data_Response Stav ve kterém stanice setrvává jeden slot-time a

očekává odezvu nebo potvrzení "action frame" paketu. Po obdržení očekávaného paketu následuje přechod do USE_TOKEN stavu. V případě, že na sběrnici se objeví jiný paket, sice platný, ale ne ten očekávaný, stanice přechází do ACTIVE_IDLE stavu.

Check_Access_Time Kvůli zjednodušení, které bylo vysvětleno v popisu USE_TOKEN stavu, stav CHECK_ACCESS_TIME plní pouze symbolickou úlohu. Má uvnitř naprogramovanou logiku, která umožňuje návrat do USE_TOKEN stavu, ale tu řídí pouze lokální proměnné, které se nemění. Takže při následujícím časovém tiku přechází stanice vždy do PASS_TOKEN stavu.

Pass_Token V tomto stavu stanice předává token svému nástupci v LAS, pokud takový existuje. Ale nejprve musí zkontrolovat jednu adresu ve svém GAP listu, zda se neobjevila nová stanice s touto adresou. Proto mění svůj aktuální stav na Await_Status_Response stav. Tady na moment odbočím. Správně kontrola nové stanice musí proběhnout po vypršení GAP Update Time. V rámci zjednodušení tento časovač nebyl brán v úvahu a kontrola nové stanice probíhá vždy.
Po návratu z tohoto stavu zpět do Pass_Token stavu provede předání tokenu a odebere se do Check_Token_Pass stavu. V případě, že zjistí, že je jedinou aktivní stanicí na sběrnici, tak přejde přímo do Use_Token stavu.

Check_Token_Pass V tomto stavu stanice čeká po dobu než uplyne Slot-time, zda se na sběrnici objevil nějaký paket, který by signalizoval, že aktivitu spolu s tokenem převzala jiná stanice.

Await_Status_Response Stav ve které stanice čeká po dobu než uplyne Slot-time, zda jiná stanice, které jsme poslali FDL_Status_Request paket odpoví nebo ne. Pokud odpoví a tato stanice není uvedena v našem GAP listu, pak je to nováček na sběrnici. Pokud odpověď zní READY_TO_ENTER_LOGICA tak je to nový master. V tom případě naše stanice (ta co posílala request) musí zkrátit svůj GAP list tak, aby NS byla právě nalezená stanice. Jinak je to slave a bude zařazen do GAP listu.

Passive_Idle To je stav ve kterém se nachází slave, pokud je inicializovaný. Poslouchá a čeká na request od nějakého mastera.

7.2 Přechody

V této sekci najdeme vysvětlení za jakých podmínek k aktivaci jednotlivých přechodu dochází. Termínem aktivace přechodu mám na mysli skutečný pře-

chod stanice z jednoho stavu do druhého přes daný přechod. Na začátku je uvedeno písmeno, které symbolizuje přechod na obr.2

a) OFFLINE – LISTEN_TOKEN Po uplynutí inicializace operačních parametrů přechází master do stavu LISTEN_TOKEN

b) OFFLINE – PASSIVE_IDLE Po uplynutí inicializace operačních parametrů přechází slave do stavu PASSIVE_IDLE.

c) LISTEN_TOKEN – OFFLINE Pokud při generování LAS (List of Active Station) stanice zaznamená paket, ve kterém zdrojová adresa se bude shodovat s její, tak vyvodí závěr, že stanice s danou adresou již existuje a proto přejde do stavu OFFLINE.

d) LISTEN_TOKEN – ACTIVE_IDLE Pokud dostane master od svého předchůdce PS (Previous stationo) Request FDL Status a odpoví zprávou "Ready to enter logical ring", tak při dalším přijetí token paketu může stanice přejít do stavu ACTIVE_IDLE.

e) ACTIVE_IDLE – LISTEN_TOKEN Pokud stanice dostane token paket se svoji adresou uvedenou jako cílovou adresu, ale rozhodne se, že nechce zůstat v logickém kruhu, pak se vrací zpět do LISTEN_TOKEN stavu.

f) ACTIVE_IDLE – CLAIM_TOKEN Pokud na sběrnici není žádná aktivita po dobu rovnající se time-outu stanice, přechází stanice do stavu CLAIM_TOKEN.

g) ACTIVE_IDLE – USE_TOKEN Pokud stanice dostane token frame se svoji adresou uvedenou jako cílovou adresu, pak přechází do USE_TOKEN stavu.

h) CLAIM_TOKEN – PASS_TOKEN Při inicializaci stanice přechází do stavu PASS_TOKEN aby předala token sama sobě a dělá to dvakrát po sobě.

i) CLAIM_TOKEN – USE_TOKEN Při re-inicializaci stanice přechází do stavu USE_TOKEN, jelikož GAP a LAS už má vytvořený. Tento přechod v programu není zahrnut. Tam v případě poruchy probíhá vždy úplná inicializace logického kruhu.

j) USE_TOKEN – ACTIVE_IDLE Ze specifikace nevyplývá, kdy dochází k aktivaci tohoto přechodu.

k) USE_TOKEN – AWAIT_DATA_RESPONSE K přechodu dojde, pokud stanice vyšle action frame.

l) USE_TOKEN – Check_Access_Time Podle specifikace k přechodu dojde když ve stavu USE_TOKEN nejsou naplánované procesy (message cycles) s vysokou prioritou, nebo po vykonání těchto procesu, nebo po vykonání procesu s nízkou prioritou. V simulátoru k přechodu dojde, jakmile stanice skončí "polling" stanic z GAP listu.

m) AWAIT_DATA_RESPONSE – USE_TOKEN Podle specifikace k přechodu dojde pokud dostane "acknowledgement" nebo "response" paket v době než vyprší slot-time nebo když vyprší slot-time u prvního pokusu získat odezvu..

n) AWAIT_DATA_RESPONSE – ACTIVE_IDLE K přechodu dojde pokud dostane platný paket, jiný než v případě m) v době než vyprší slot-time nebo když vyprší slot-time u opakovaného pokusu získat odezvu.

o) Check_Access_Time – USE_TOKEN Při výpočtu rozdílu Tr (TargetRotationTime) a Rr (RealRotationTime)výsledkem je hodnota, která určuje, zdá stanice může ještě zůstat v USE_TOKEN stavu. V případě, že je kladná, vrací se stanice do USE_TOKEN stavu. Kvůli zanedbání dvou výše zmíněných parametrů tento přechod v simulátoru není aktivní.

p) Check_Access_Time – PASS_TOKEN Kvůli zanedbání parametrů Tr a Rr k tomuto přechodu dochází vždy, když se stanice ocitne v Check_Access_Time stavu. Detaily jsou v sekci 7.1 USE_TOKEN.

r) PASS_TOKEN – USE_TOKEN K tomuto přechodu dochází v případě, že stanice ve svém LASu nemá jiného mastera, proto nemusí nikomu předávat token, takže si ho ponechá a vstupuje znovu do stavu USE_TOKEN.

q) PASS_TOKEN – Check_Access_Time O tomhle přechodu jsem ve specifikaci nenašel žádnou zmínku. Logické vysvětlení by bylo, že stanice v PASS_TOKEN stavu musí ověřovat, zda má k dispozici čas pro provedení updatu svého GAP listu. A dělá to když vyprší GAP Update time, který se rovná nějakému násobku Target Rotation time. Tento přechod není naprogramován.

t) PASS_TOKEN – AWAIT_STATUS_RESPONSE Tento přechod je využit vždy poté, co se node dostal do PASS_TOKEN stavu z CHECK_ACCESS_TIME stavu.

s) PASS_TOKEN – CHECK_TOKEN_PASS Tento přechod se aktivuje po návratu stanice z AWAIT_STATUS_RESPONSE stavu do PASS_TOKEN stavu. Nebo po neúspěšném pokusu o předání tokena a návratu z CHECK_TOKEN_PASS stavu do PASS_TOKEN stavu.

u) PASS_TOKEN – LISTEN_TOKEN Tento přechod se aktivuje, pokud dojde k situací, kdy majitel tokenu pošle tokenFrame svému nástupci, ale zjistí, že na sběrnici je poškozený paket. Toto může být dočasná chyba přijímače, vysílače nebo sběrnice. Proto při první detekci vadného paketu, stanice přechází do CHECK_TOKEN_PASS stavu jako kdyby vše bylo v pořádku. Pokud nedojde k žádné aktivitě na sběrnici, pak to znamená chybu i tokenFrame se posílá ještě jednou. V případě, že stanice opět zaregistruje poškozený paket na sběrnici, tak přechází do Listen_Token stavu a hlásí chybu managementu.

v) PASS_TOKEN – OFFLINE Tento přechod se aktivuje, pokud dojde k chybě na straně vysílací stanice. Detekce chyby spočívá v tom, že přestože stanice pošle tokenFrame paket (token), tak ho "nevidí" na sběrnici. Závěr je takový, že buď je vadný vysílač a paket nebyl odeslán nebo je vadný přijímač. V každém případě stanice mění svůj stav na OFFLINE a hlásí to managementu. Hlášení v simulátoru není implementováno.

x) CHECK_TOKEN_PASS – ACTIVE_IDLE K přechodu dojde, pokud v době slot-time je zaznamenána na sběrnici hlavička platného paketu. Pak se předpokládá, že předání tokenu proběhlo v pořádku. Pokud je zaznamenán neplatný paket, pak se dojde k závěru, že aktivní je jiná stanice než se předpokládalo a stanice stejně přejde do stavu ACTIVE_IDLE. Takže závěr zní: Pokud se objeví jakýkoliv paket na sběrnici, ať už očekávaný nebo neočekávaný, přechod do stavu ACTIVE_IDLE je jistý.

w) CHECK_TOKEN_PASS – PASS_TOKEN K přechodu dojde (nebo přesněji řečeno k návratu dojde), pokud stanice v době slot-time nezaznamená žádný paket na sběrnici.

y) AWAIT_STATUS_RESPONSE – PASS_TOKEN Přechod se aktivuje, když stanice dostane potvrzovací (acknowledgement) paket od tázané stanice a také když nedostane žádný paket nebo poškozený.

z) AWAIT_STATUS_RESPONSE – ACTIVE_IDLE Přechod se aktivuje, když stanice dostane jakýkoliv platný paket s výjimkou potvrzovacího. Pak to znamená, že token je ve vlastnictví nějaké jiné stanice.

7.3 Sporné body

Pokud v ListenToken stavu dostane master token se svoji adresou, tak přejde do ActiveIdle stavu. A protože stanice, která předává token a nachází se v CheckPassToken stavu v době, která se říká slot-time, nezjistí žádnou aktivitu, tak podle specifikace se pokusí ještě jednou poslat token. Mezitím naše stanice je už v ActiveIdle stavu a čeká na další token, aby mohla přejít do UseToken stavu. Tento další token jí přijde, jako druhý pokus prvního tokenu. Takto je naprogramován simulátor, protože to odpovídá specifikaci. Je tu ovšem velká pravděpodobnost, že zamyšlen byl jiný postup. A sice, že master předávající token, to udělá jen jednou. Naše stanice přejde do ActiveIdle stavu a master vzápětí předá token další v pořadí stanici. Takže naše stanice může zůstat ve stavu ActiveIdle ještě jedno kolo tokenu.

7.4 Funkční celky simulátoru

V souladu s požadávky na simulátor, které jsou detailně popsáné v 6.1 uživatel má k dispozici několik funkčně odlišných ploch.

Katalog Zde jsou na výběr tři prvky. Sběrnice, uzel typu "Master" a uzel typu "Slave". Pokud uživatel, chce přidat nějaký z těchto prvku do editoru, pak stačí označit příslušnou iknou kliknutím a následně kliknout v editoru na místo, kde si přejeme mít daný prvek.

Editor Plocha na které je zobrazena sběrnice se všemi připojenými uzly. Při připojování nových uzlu ujistěte se, že v editoru je již sběrnice.

Ovládací panel Zde jsou tlačítka, pro krokování v čase a také pro spuštění a zastavení simulace. Najdete tu rovněž posouvací lištu, která určuje rychlost simulace. Jeden krok můžete nastavit v rozsahu 1 až 5 sekund.

Informační panel Zde je několik tabulek. V jedné najdeme informace o stavu všech uzlů. V další tabulce jsou detailní informace o jednom právě vybraném uzlu. Najdeme zde stav uzlu, jeho adresu, typ a v případě, že je to master ještě LAS a GAP.

Textová konzole Vypisuje průběh simulace a poskytuje jiné užitečné informace

Inicializační panel Umožňuje nastavit parametry jednotlivých uzlů.

Panel pro zasílání zprav Umožňuje poslat zprávu od jednoho uzlu jinému.

7.5 Programování

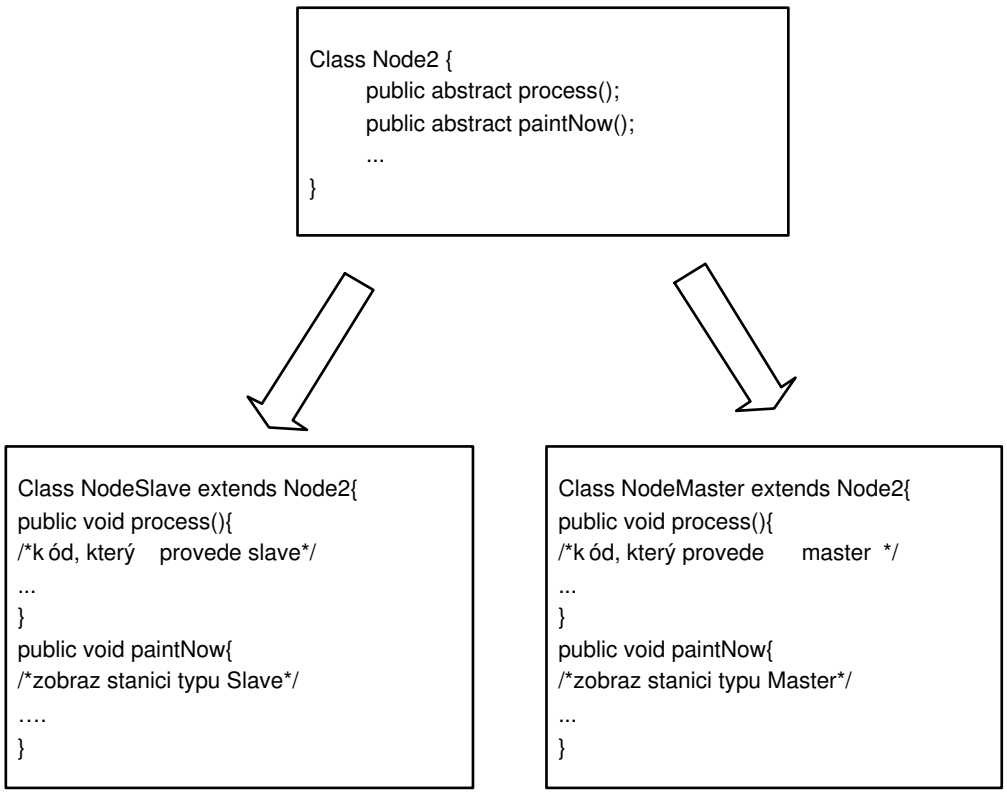
7.5.1 Java, applet, swing

Applet je napsán v jazyce Java. Applet je program, který je k dispozicí na webovém serveru a který můžeme spustit pomocí www prohlížeče. Prohlížeč musí mít zabudovanou podporu Javy. Před pár léty všichni výrobci webových prohlížečů slibovali zabudovanou podporu Javy ve svých produktech, Microsoft ovšem se odklonil od původních plánů a dnes to uživatele MS Exploreru musí řešit instalací java plug-inu.

Pro vytvoření grafického uživatelského prostředí posloužila knihovna SWING, která je součástí JFC (Java Foundation Classes). Největší problémy mi dělalo rozmístění komponent na obrazovce. Každá komponenta musí být součástí kontejneru. U kontejneru můžeme specifikovat tzv. Layout manager. Je to správce rozmístění komponent v daném kontejneru. Uživatel pak pouze přidává komponenty do kontejneru a tento správce je rozmísťuje svým typickým způsobem. Máme k dispozici celou řadu těchto správců, od nejjednodušších jako např. FlowLayout nebo BorderLayout po komplexní jako například GridBagLayout. V knížce "Thinking in Java" [2] se doporučuje dávat přednost kombinací panelů s více jednoduchými správci před použitím GridBagLayoutu. Já jsem využil modifikace GridLayout manageru. Pozměněný správce má jméno GridLayout2 manager. Tento správce umísťuje stejně jako jeho rodič komponenty do mřížky. Na rozdíl od standardního správce sloupce mohou mít vzájemně odlišnou šířku a řádky se mohou lišit výškou. Více informací ohledně modifikovaného správce a jeho zdrojové kódy naleznete v tomto článku [4]. Výsledná kompozice se skládá z několika kontejnerů typu Panel, které používají převážně GridLayout2 manager. Tyto panely pak jsou rozmístěné v kontejneru JAppletContentPane, což je základní kontejner appletu. Při použití GridLayout2 manageru v JAppletContentPane

Při vytváření appletu jsem využil hlavní výhody objektového programování. A sice dědění, polymorfismus a použití vláken. Na principu **dědění** jsou vytvořené třídy *NodeMaster* a *NodeSlave*, mají totiž mnoho společných rysů, proto jejich základem se stala třída *Node2*. Tato třída je abstraktní, protože obsahuje několik abstraktních metod, jako jsou *paintNow()* nebo *process()*. Jsou to metody, které v každé zděděné třídě vykonávají stejnou funkci odlišně. Například metoda *paintNow()* vykresluje ikonu na ploše editoru. Pro třídu *NodeMaster* musí vykreslit obrázek mastera a pro třídu *NodeSlave* zobrazit stanici typu slave. Metoda *process()* na základě informací o aktuálním stavu stanice a okolních požadavků může změnit aktuální stav stanice nebo

poslat nějaký paket na sběrnici.Tady se dostáváme k **polymorfismu**. Nejlepší bude uvést konkrétní příklad. Stanice typu master a slave jsou obě ve stavu OFFLINE, obě již mají inicializované operační parametry a jsou připravené změnit svůj stav. Obě třídy *NodeMaster* a *NodeSlave* musí mít nadefinovanou metodu *process()*, kdyby nebyla, hlásil by kompilátor chybu. Zavolám li metodu *process()* pro instanci třídy *NodeSlave*, změní tato stanice svůj aktuální stav z OFFLINE na PASSIVE_IDLE. V případě, že zavolám metodu process() pro instanci třídy *NodeMaster*, změní svůj aktuální stav na LISTEN_TOKEN.



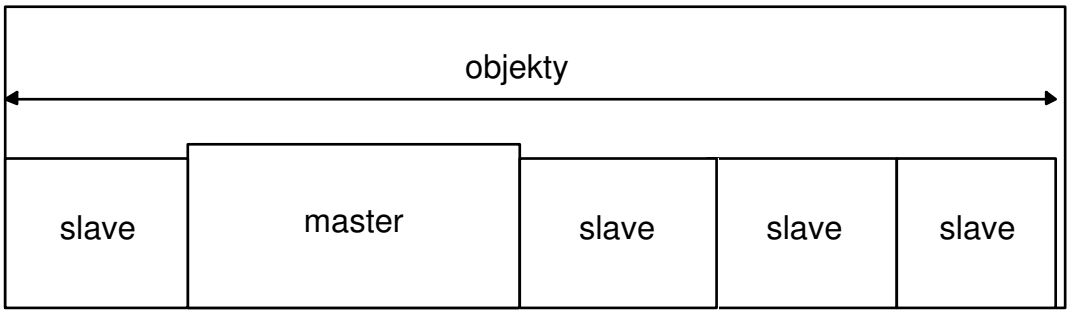
Obrázek 3: Dědění a polymorfismus

A teď přichází ta zmiňovaná výhoda. Všechny stanice mám uložené ve struktuře ArrayList. Tato struktura může obsahovat libovolné množství různých objektů. Pro každý objekt z instance této struktury mohu po jeho přetypování na Node2 zavolat metodu process() a vím, že se provede správná metoda, aniž bych přesně specifikoval třídu daného objektu.

```
/*
 *  nodes je instance třídy ArrayList, kde jsou uložené
```

```
 *  všechny stanice, jak masters tak i slaves
 */
ListIterator lit = nodes.listIterator();
Node2 n2=null;
while(lit.hasNext()){
    n2 = (Node2)lit.next();
    /*process and generate response*/
    n2.process();
}
```

Tento přístup je velice elegantní, vyžaduje však promyšlenou strukturu tříd.



Obrázek 4: ArrayList

Použití vláken

Tato problematika je velice obsáhlá a zajímavá. Odborníci doporučují vyhnout se použití vláken, pokud to není nezbytně nutné. Je třeba zvážit přínos použití vláken. Typickými příklady použití je vyčkávání na vstupně-výstupní operace. Další oblastí jsou úlohy náročné na čas zpracování. Pokud ji můžeme rozdělit na relativně nezávislé části, které mohou běžet paralelně a máme k dispozici víceprocesorový systém, tak se můžeme dočkat zrychlení výpočtů. Já jsem použil vlákna z důvodu vytvoření citlivého uživatelského rozhraní. Umožňuje zasáhnout do probíhajícího procesu, případně ho zastavit. V simulátoru máme možnost spustit časovač, který v rozmezí jedné až pěti sekund automaticky zvyšuje uplynulý čas. Rychlost určíme posuvníkem vedle časomíry. Abychom zastavili časovač tlačítkem Stop, musí proces časovače ve svém průběhu předávat řízení na nějakou dobu aplikaci. To je podstata využití více vláken v procesu. Máme zdání paralelního průběhu několika podprocesů. Ve skutečnosti se podprocesy sekvenčně střídají ve využití času

procesoru. Při běhu aplikace existuje vlákno, které se jednou za čas probudí, v případě potřeby zvýší čas o jednotku a opět usne. Tím předá řízení zpět aplikaci. Vlákno je řešeno jako soukromá vnitřní třída, která je zděděna od třídy Thread. Metoda run(), což je základní metoda vlákna, je zde uvedená.

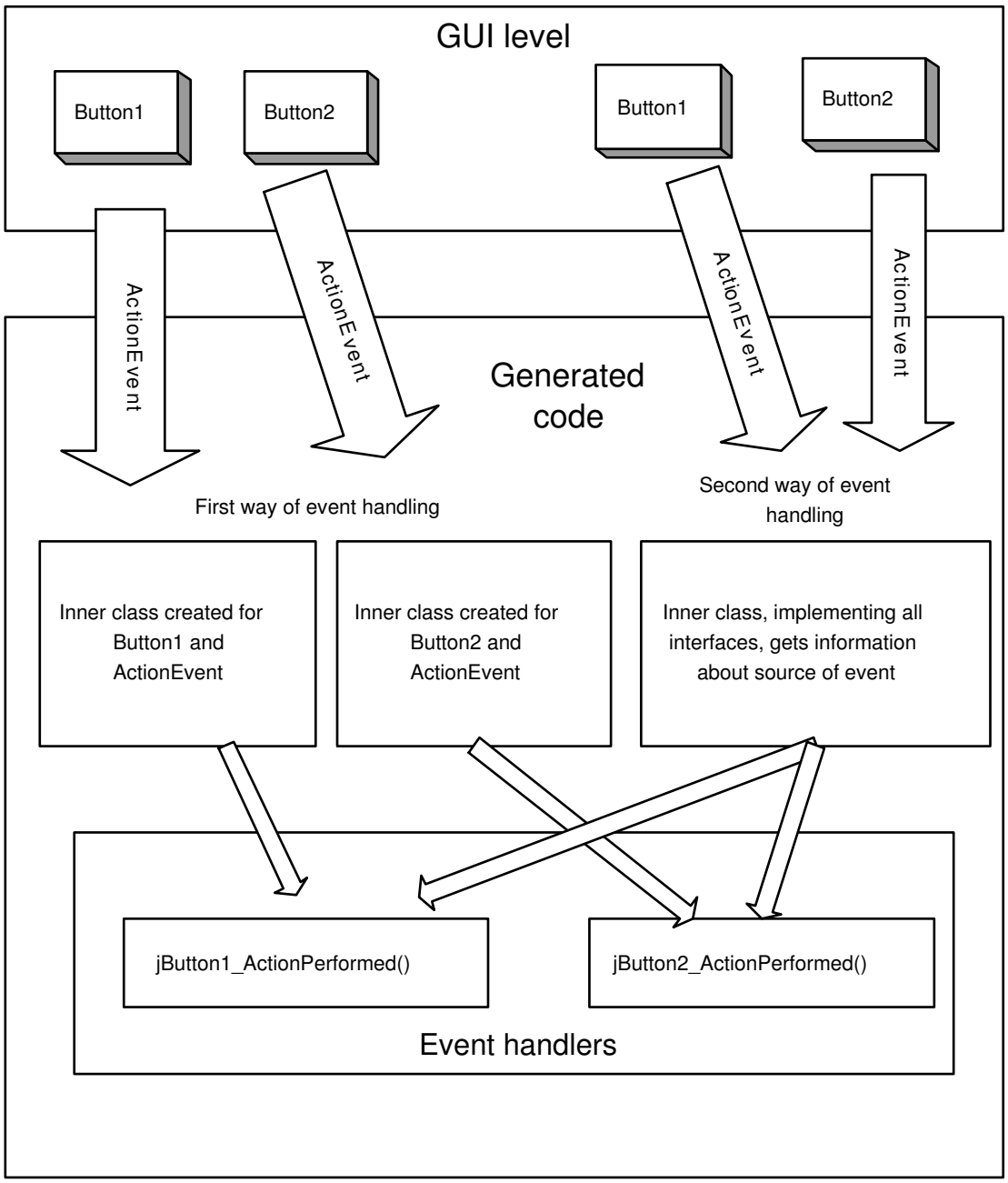
```
public void run() {
while (true) {
  try {
    System.out.println("Time to sleep: "+time);
    sleep(time);
  } catch (InterruptedException ie) {
    ie.printStackTrace();
  }
  if(ivjJButtonSimulator.getText().equals("Stop")){
    /* timer is on, we have to increment its value*/
    /* press button "TimeUP" */
    ActionEvent ae=new ActionEvent(ivjJButtonTimeUp,1000,"nic");
    jButtonTimeUp_ActionPerformed(ae);
    }
 }
}
```

Všechno je jednoduché. Nejsou tu semafóry nebo zámky.

Princip reakce na události při použití knihovny SWING.
Grafické komponenty, které vidíme na obrazovce musí mezi sebou nějakým způsobem komunikovat. Knihovna SWING definuje řadu událostí, které mohou být generovány jednotlivými komponentami. Komponenta si zaregistruje posluchače svých událostí. Jako posluchače nemůže zaregistrovat libovolnou komponentu. Posluchač musí být schopen danou událost přijmout. Tato podmínka je realizována pomocí rozhraní, které posluchač musí implementovat. Každá událost má své rozhraní. Například objekty tříd *Checkbox* nebo *Choice* mohou generovat událost typu *ItemEvent*. Pokud nějaký objekt chce být posluchačem této události, musí implementovat rozhraní *ItemListener* a musí být zaregistrován u zdroje události. Toto rozhraní má pouze jednu metodu. Jiný rozhraní mají metod více, a každá metoda reaguje na jiný typ události Existuje analogie mezi jménem události a příslušným jménem rozhraní, které implementuje posluchač. Pokud událost se jmenuje *XxxEvent*, potom rozhraní má jméno *XxxListener* a registraci provádí objekt pomocí

metody addXxxListener(). Co se stane, když máme dvě komponenty stejného typu, například tlačítka, a obě si zaregistrují stejného posluchače? Obě tlačítka vysílají stejný typ události. Nezbývá nic jiného než rozlišovat původce událostí. Pavel Herout ve své knize [3] doporučuje pro každý zdroj události vytvořit vnitřní třídu, která implementuje odpovídající rozhraní a které bude událost zasílána. Má to navíc výhodu, že vnitřní třída může přistupovat ke všem metodám a proměnným vnější třídy.

Simulátor byl naprogramován s použitím vývojového prostředí Visual Age for Java (VAJ) od firmy IBM. Jsou tam uplatněné charakteristické rysy pro Rapid Application Development. Tento vývojový nástroj generuje dobře čitelný kód a mohl jsem vybrat, zda chci vytvořit pro každý zdroj události zvláštní anonymní vnitřní třídu nebo mít jednu vnitřní třídu ve které se zjišťuje zdroj události. V prvním případě každá anonymní třída volá určitou metodu, ve které je již zdrojový kód, který vložil programátor, jakýsi handler události. Například pro první tlačítko v appletu existuje metoda *jButton1_ActionPerformed(java.awt.event.ActionEvent actionEvent)*, pro druhé existuje metoda *jButton2_ActionPerformed(java.awt.event.ActionEvent actionEvent)*. V druhém případě ve vnitřní třídě se určí zdroj události pomocí metody *getSource()* a poté se zavolá příslušná metoda. Pro první tlačítko je to znovu metoda *jButton1_ActionPerformed(java.awt.event.ActionEvent actionEvent)*. Takže "navenek" pro programátora obě možnosti vypadají stejně, žádné výhody nevidím. Pro snadnější pochopení tohoto odstavce je tu obrázek 5, který má pomoci získat správnou představu.



Obrázek 5: Generování kódu ve VAJ, rozdíl při použití jedné společné vnitřní třídy a zvláštní vnitřní třídy pro každou komponentu a událost

7.5.2 Test Driven Development

Funkční jádro aplikace, a tím je třída *Node* spolu s zděděnými třídami *NodeMaster* a *NodeSlave* byly vytvořené metodikou "Test Driven Development (TDD)". Tato metodika programování zasluhuje větší pozornost a proto jsem ji věnoval jednu podkapitolu.

Jak se programuje?

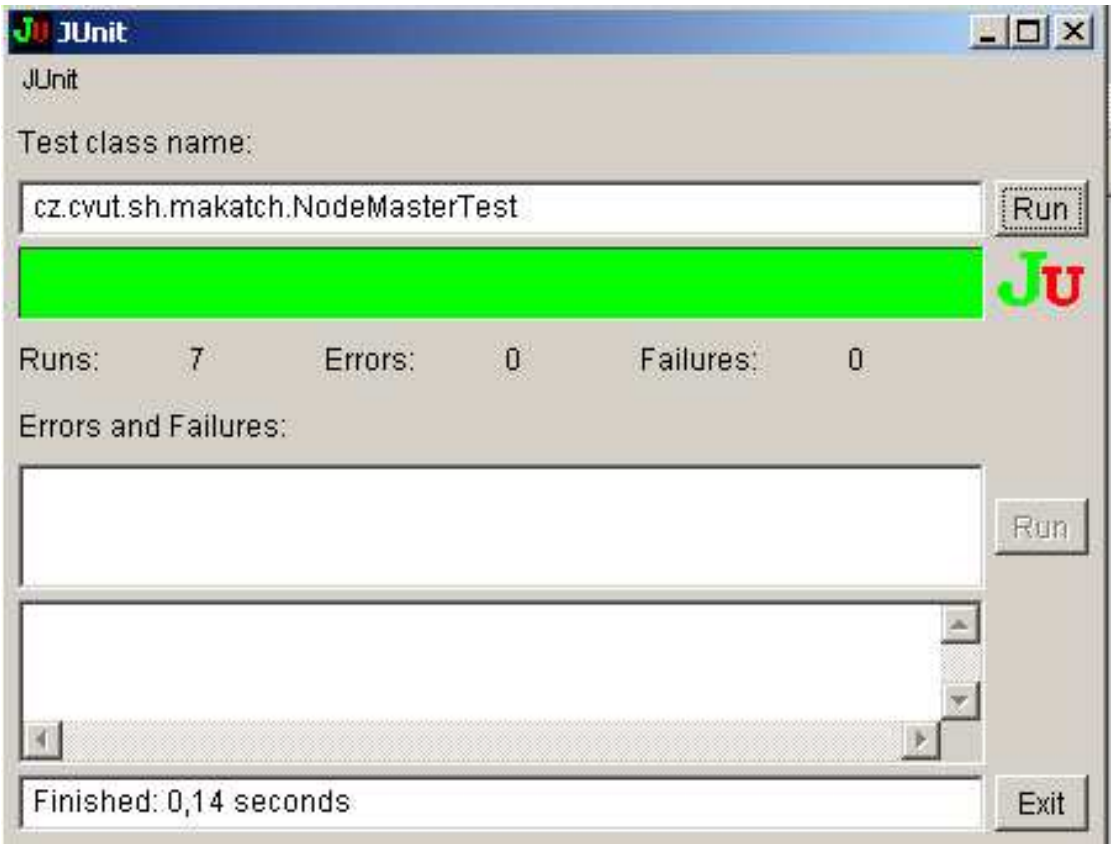
Nejprve programátor musí napsat **test**, té třídy, případně metody, kterou hodlá naprogramovat. V testu porovnává výsledky dosažené použitím budoucí třídy s výsledky předpokládanými. Pokud ještě nemáme naprogramovanou třídu nebo metodu, je jasné, že test dopadne negativně. Potom programátor začíná psát požadovanou třídu. Programování třídy končí ve chvíli, kdy je splněn test. Toto je okamžik pro vytvoření následujícího testu, který klade další nároky na naší třídu a vzápětí dodělat případně pozměnit naší třídu tak, aby splňoval rovněž tento test. Výsledkem je cyklické psaní testu a kódu, které splní příslušné testy. Vývoj postupuje malými kroky, to však neznamená že pomalu, každý krok navíc je kontrolován příslušným testem.

Výhody

Tato metoda přináší tu výhodu, že v jakémkoliv okamžiku vím, jaká část kódu je již funkční, protože správně napsané testy mi tuto funkčnost zaručí. Kód, který jen částečně splňuje počáteční záměry aplikace, ale spolehlivě vykonává již hotové částí přináší víc spokojenosti, než kód který se snaží od samého počátku o úplnost ale nefunguje nebo funguje nespolehlivě. Osobně největší výhodou vidím v tom, že se nebojím zasáhnout do již funkčního kódu a přepsat jej, například pro odstranění duplikací, provést tzv. refactoring. Okamžitě zjistím, jestli změny neporušili funkčnost a pokud ano, tak vím přesně, kde hledat chybu. Při programování se mohu soustředit jen na dílčí problém, cílem je splnit test. Vyjmenoval jsem jen hlavní výhody, které jsem ocenil na vlastní zkušenosti. V odborné literatuře [5] lze najít spoustu dalších důvodů, které programování významně zefektivní.

Nevýhody

Zezačátku to vypadá jako plýtvání časem. Člověk musí mít velkou motivaci, aby vyzkoušel tuto metodu. Musí porušit svoje zaběhnuté zvyky, ne každý je ochoten to udělat. TDD slibuje programátorům zbavit jich stresových situací. Jsou to případy, kdy programátor věnoval mnoho úsilí a času nějakému kódu, blíží se termín odevzdání a program stále nefunguje. Výsledkem je strach z toho, že nebude co předvést a řešení je nejisté a v nedohlednu. Zbavit se takových situací je rozhodně velká motivace. Ještě větší však stává

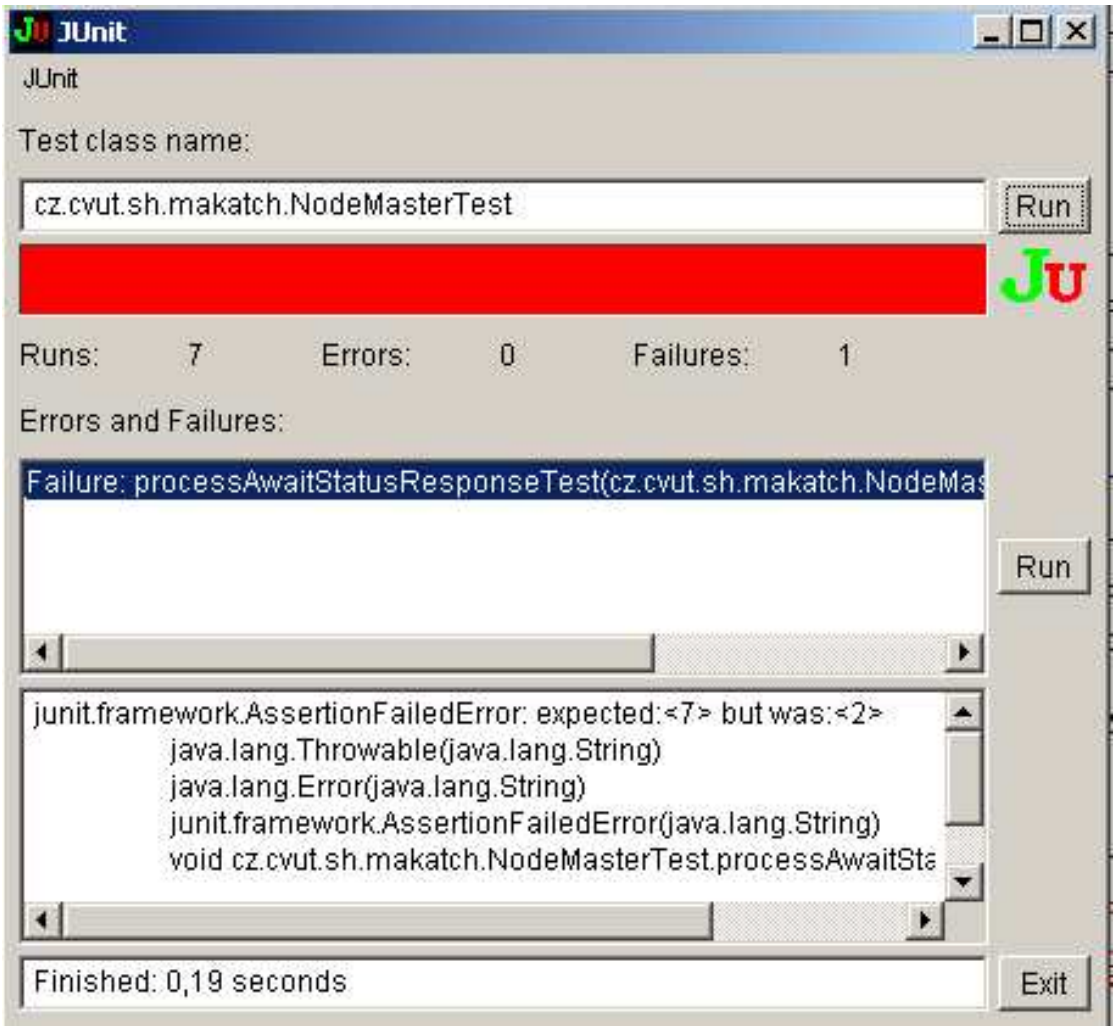


Obrázek 6: Junit, úspěšný test

v okamžiku, kdy programátor se do ní dostane. Je však nutné věnovat čas pro studium alespoň základů této metodiky. A času není nazbyt. Vzniká dilema, pokračovat dále ve stejné taktice, nebo nasměrovat svoje úsilí na Test-Driven Development?

Výsledky použití TDD

Znalci této metodiky programování by označili moje testy za vzdálené od ideálů, neboť jim chybí úplnost. Autoři této metody, definují ideální test takový, který odhalí zakomentování libovolné řádky kódu, případně změnu hodnoty proměnné. Pokud k tomu nedojde, pak mají dvojí vysvětlení. Buď je tato řádka kódu zbytečná nebo test není úplný.
Nemohu tvrdit, že tato metoda ušetří čas. Záleží jak kvalitní kód produkuje programátor. Nastanou-li potíže, pak vyhledání chyby je snadnou záležitosti. Psaní testů je investice do budoucna, obzvláště u větších projektů. Tato metodika slibuje mnoho pozitiv, ovšem za cenu obtížnějšího přechodu.



Obrázek 7: Junit, neúspěšný test

Junit je jednoduché prostředí pro automatické provádění opakovaných testů. Pokud chceme testovat nějakou třídu A, tak vytvoříme třídu *testA*, ve které budou metody ověřující předpokládanou funkčnost metod třídy A. Pak vytvoříme objekt třídy *junit.framework.TestSuite*, pojmenujeme ho *suite*. Do tohoto objektu zařadíme ty metody *testA* třídy, které mají být v testu a pak pouze spustíme Junit program, který vidíme na obr.6 nebo na obr. 7. Do textového pole napíšeme jméno naše testovací třídy, tedy *testA* a to včetně balíku a tlačítkem Run spustíme testování. Výsledný zelený pruh, znamená, že všechny testy, které jsou zařazené do objektu suite proběhly bez chyb.

Pokud se objeví červený pruh, pak to znamená, že jeden nebo více testu jsou neúspěšné.

7.5.3 Hlavní třídy simulátoru

Tato sekce by měla poskytnou základní orientaci v naprogramovaných třídách.

Třída *FDLSimulator* je hlavní a nejrozsáhlejší třídou simulátoru, je odvozená od třídy *JApplet*. Obsahuje grafické uživatelské rozhraní a handlery pro zpracování událostí. V ní jsou uloženy všechny objekty, buď přímo nebo jako součást jiných objektů, které se podílejí na simulaci.

Třídy *NodeMaster* a *NodeSlave* obsahují vlastní algoritmus přechodů, který vidíme na obr.7.1.

Třída *LASandGAP* reprezentuje dvě části simulátoru. První je LAS List of Active Stations, druhá pak je GAP popisuje rozsah adres od dané stanice (TS This Station) do následující aktivní stanice (NS Next Station). Obě části jsou řešeny jako pole hodnot boolean, o maximálním rozsahu, který si určí uživatel sám. Podle specifikace je to hodnota 128, ovšem pro takový rozsah adres simulace se stává zdlouhavou a jednotvárnou, proto je tu možnost tuto hodnotu snížit. V simulátoru tento parametr najdeme pod označením "Range" a je přednastaven na hodnotu 20.

Třída *FDLTimer* obsahuje tři časovače, které hrají důležitou úlohu v algoritmu při změně stavů.

7.5.4 Dokumentace

Vytvoření programu bez dokumentace není v dnešní době přípustné. Ale existují výjimky o kterých se zmíním později. Vytvořil jsem dokumentaci pomocí utility javadoc, což je pro programovací jazyk java standardní způsob. Výsledkem jsou soubory v html formátu, které obsahují hierarchickou strukturu tříd, hlavní parametry a krátký popis datových členů a metod. Dokumentace se generuje automaticky z komentářů zdrojového kódu. Aby komentář byl zahrnut do dokumentace, musí být na správném místě a používat tagy k tomu určené. Uvedu jen jeden příklad. Pokud chci specifikovat jméno autora v dokumentaci, pak komentář vypadá takto

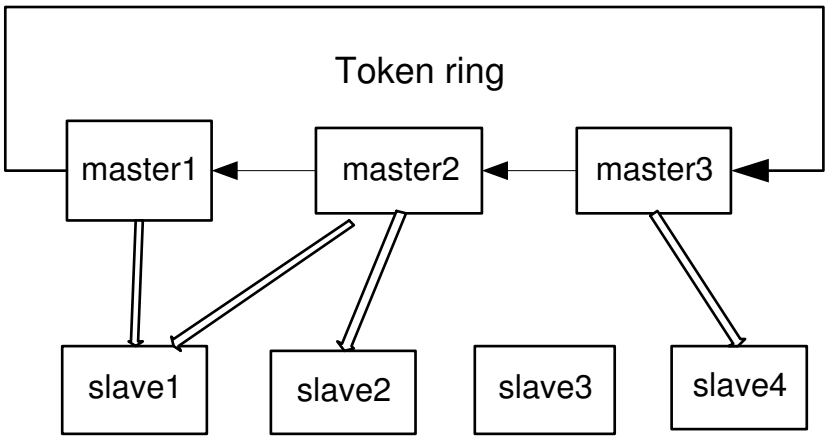
```
/**
* @author: Vladimir Karasek
*/
```

Více informací je k dispozici na příslušných internetových stránkách [6] společnosti Sun. Protože jsem se tady zmíňoval o Test-Driven programování, tak je na

místě uvést, jak je to s dokumentaci v TDD. Zastánci TDD tvrdí, že nejlepší dokumentací jsou testy. Taková dokumentace je vždy aktuální. Pokud probíhá vývoj, tak se nevyplatí psát dokumentaci pro zdrojový kód, jelikož proces změn je velice frekventovaný. Dokumentace se dodává na konci vývoje pro základní objekty.

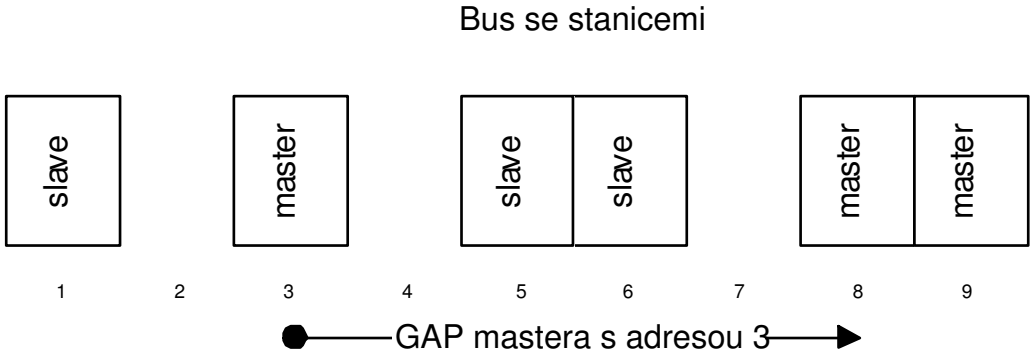
8 Vyhodnocení

Simulátor je určen pro práci sběrnice v režimu multi-master viz. obr. 8 V tomto režimu je na sběrnici několik stanic typu master a musí se střídat v přístupu na sběrnici. Pokud je konfigurace systému taková, že máme pouze jeden master, pak celková aktivita na sběrnici degraduje na cyklickou komunikaci masteru se svými podřízenými stanicemi. V tomto případě simulace reálného chování sběrnice značně pokulhává. Důvodem je fakt, že cyklická výměna dat, není zde řešená takovým způsobem, aby to odpovídalo specifikaci. Při simulaci master v době, kdy má přidělenou sběrnici, postupně zasílá svým podřízeným stanicím DATA_REQUEST požadavky a neuvolní sběrnici dokud neskončí. Podle specifikace [1], každé stanici typu master je zaručeno provedení jednoho cyklu s vysokou prioritou. Vysoká priorita může být stanovená jen u služeb SDA, SDN a SRD. Zde vzniká rozpor, jelikož v simulaci k "pollingu" dojde při každém stavu USE_TOKEN. Z toho plyne, že tento cyklus sběru dat od svých podřízených stanic může být považován za cyklus s vysokou prioritou. Polling používá typ služby CSRD a té nemůže být přidělená vysoká priorita. Další odklon od specifikace je nezavedení Poll



Obrázek 8: Multi-Master systém

Listu, což je seznam stanic, u kterých se budou žádat data. Funkci Poll Listu zastupuje GAP. O data se žádá pouze u těch stanic, které jsou v daném rozsahu GAP, viz.9. To neodpovídá skutečnosti, kdy několik masterů může číst data u stejné stanice typu slave, bez ohledu na to, že zapisovat data může jen jeden master. V simulátoru není zmínka o typech přenosových služeb (SDA,SDN,SRD a CSRD) viz. kapitola 5. Pokud master zkoumá, zda v jeho GAP listu se neobjevil nový člen, tak na danou adresu posílá

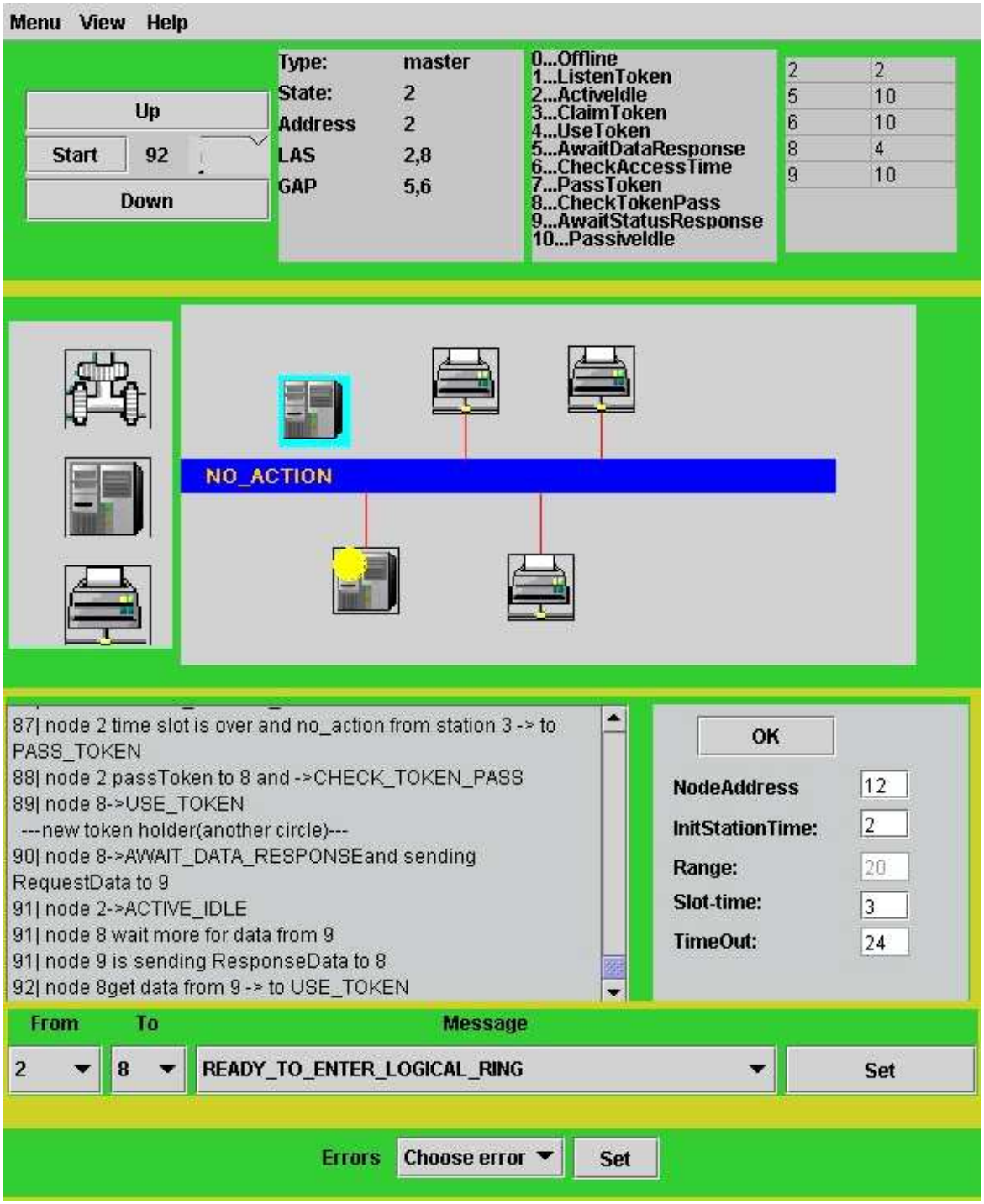


Obrázek 9: Vysvětlení pojmu GAP

FDL_STATUS_REQUEST zprávu a předpokládá se, že to je SDA zpráva. V případě, že se na dané adrese skutečně objevila nová stanice, první stanice dostane potvrzení (acknowledgement) od nováčka.

Tyto nedostatky jsou výsledkem kompromisu o kterém jsem se zmínil na začátku v kapitole 6.1.

V simulátoru není možnost vrátit se nazpět v čase, přestože zezačátku tato možnost byla naplánována. Řešení by bylo možné provést dvěma způsoby. Prvním je ukládat informace o stavu všech stanic v každém časovém okamžiku. Toto řešení by bylo možné jen pro kratší simulace. Druhým elegantnějším řešením by bylo vytvoření inverzních funkcí pro každý přechod. Ukládání vstupu od uživatele by se stejně stalo nezbytným předpokladem návratu simulátoru nazpět v čase. Důležitou roli v simulátoru hrají časovače. Jsou tu tři: Init, Time-out a Time-slot timer. Časovače by museli mít doprogramovánou možnost návratu zpět. Byl by to velký zásah do systému, ne však nemožný. Pokud bych už od začátku programoval třídu *FDLSimulátor*, což je hlavní třída celého appletu, metodou Test-Driven Development, tak bych se nebál provést radikální změny.



Obrázek 10: Simulátor

8.1 Kvantitativní úspěšnost zadání

Porovnání požadavků na simulátor z kapitoly 6.1 a dosažených výsledků podle jednotlivých bodů dopadlo následovně:

1. Sběrnici můžeme vytvořit s libovolným počtem stanic, omezení je dáno pouze velikosti editační plochy. To by se dalo vyřešit zavedením posuvných lišt, ale myslím si, že je to zbytečné. Pro pochopení komunikace mezi mastery, případně sledování pohybu tokenu stačí tři stanice a k ním samozřejmě můžeme přidat několik stanic typu slave.

2. Konfigurace jednotlivých uzlů spočívá v nastavení adresy, inicializačního času, u stanic typu master pak ještě doby čekání na odezvu (Slot-time) a time-outu.

3. Pro sledování aktuálního stavů stanic slouží dva panely. Jeden znázorňuje aktuální stav všech stanic. Druhý přináší detailní informace ohledně jedné vybrané stanice. K tomu ještě aktuální informace se vypisují na konzoli.

4. Tento bod je rovněž splněn. Uživatel může simulovat zaslání jakékoliv zprávy libovolné stanici podle svého uvážení. Pokud označí nějakou stanici, tak podle jejího typu (master nebo slave), tak v komboboxu se přednastaví vhodné pro danou stanici zprávy.

5. Celkový průběh simulace můžeme odvodit z výpisu konzole, kde na každém řádku je čas a událost, která se tehdy odehrála. Každá změna stavu uzlu se projeví na konzoli.

6. Časový posun a to ruční a automatický, ten poslední s nastavením rychlosti změny času je k dispozici pouze v dopředném směru.

7. Možnost navození chybového stavu tu je. Ale jedna se jen o jeden problém, a sice ztrátu tokenu.

Toto je porovnání spíše kvantitativní, co se týče kvality odvedené práce, to posoudí jiní.

Applet byl úspěšně vyzkoušen na lokálním počítači s danou konfigurací:
Java^{TM}Plug-in: Version 1.4.1_02
Using JRE version 1.4.1_02 Java HotSpot^{TM}Client VM

8.2 Použití LaTeXu

Diplomová práce je vytvořená sázecím systémem LaTeX. Je to typografický systém specializující se na psaní vědeckých a matematických textu na profesionální úrovni. Jeho základem je formátovací program TeX. Nejsložitější na tomto systému je jeho uvedení do provozu. Obzvláště pokud chceme používat češtinu musíme se blíže seznámit s typografickými konvencemi a filozofii TeXu. Chvíli trvá než člověk si udělá přehled, co je funkční jádro, co je nástavba jádra a co je distribuce. Vyjmenuji jenom základní věci, které se musí udělat pro to, abychom začali vytvářet dokument v češtině. Především musíme vybrat distribuci, kterou hodláme používat. Pro vytvoření diplomové práce jsem použil MiKTeX. Je to nejrozšířenější distribuce LaTeXu běžící pod operačním systémem MS Windows. Je nabízena za tu nejlepší cenu, a sice zdarma. Pak je třeba nainstalovat české fonty a vygenerovat format csLaTeX. Potom vybrat vhodný editor TeXových dokumentů, osobně doporučuji WinEdt, jedná se však o sharewarový produkt. Vyplatí se přidat do WinEdt český slovník pro kontrolu pravopisu. Pro kontrolu výstupu se používá program Yap, což je freewarový prohlížeč dvi souborů a je součástí MiKTeX distribuce. Tento program nabízí i možnost zpětného vyhledání té části kódu, kterou označíme v prohlížeči. Shledávám docela obtížné v některých případech umístit obrázek na žádané místo. Pro obrázek 5 systém nemohl nalézt české fonty, výsledný obrázek měl vynechané znaky s diakritikou. Je zajímavé, že pro jiné obrázky stejné fonty k dispozici byly. Jako nejschůdnější řešení jsem považoval překlad textu na obrázku do angličtiny.

Diplomová práce je kratšího rozsahu, neboť dokumentace a zdrojový program je na přiloženém CD. Není třeba plýtvat papírem, ale šetřit lesy a čas zkoušejících :-)

Reference

[1] PROFIBUS Specification, *Normative Parts of PROFIBUS -FMS, -DP, -PA according to the European Standard EN 50 170 Volume 2* Edition 1.0 March 1998

[2] Eckel,Bruce *Thinking in Java* Prentice Hall 2000

[3] Herout,Pavel *Java - grafické uživatelské prostředí a čeština* Kopp, 2001

[4] Dorohonceaunu,Bogdan *Flex your grid layout* JavaWorld [online]. December 14,2001 Dostupné z < http://www.javaworld.com/javaworld/javatips/jw-javatip121.html >

[5] Beck,Kent *Test-Driven Development By Example* Addison Wesley November 08, 2002

[6] Article, *How to Write Doc Comments for the Javadoc™ Tool* [online] Dostupné z http://java.sun.com/j2se/javadoc/writingdoccomments/

[7] T. Oetiker , *A not very Short Introduction to LaTeX* 2_ϵ, Version 1.1, November, 1994

[8] Doob,Michael *Jemný úvod do TeXu* Manuál pro samostatné studium, Československé sdružení uživatelů TeXu, Praha 1993