

Automatizace testování software

Automation of Software Testing

Bc. Radek Liška

Diplomová práce
2014



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2013/2014

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Bc. Radek Liška**
Osobní číslo: **A12494**
Studijní program: **N3902 Inženýrská informatika**
Studijní obor: **Informační technologie**
Forma studia: **kombinovaná**

Téma práce: **Automatizace testování software**

Zásady pro vypracování:

1. Seznamte se s komponentou TGC (Test Gateway Component) a testovacím systémem TTT (TCL Testing Tool). TGC slouží k překladu protokolů a testuje se s ní, zda komponenty z ní odvozené správně plní svou funkci. TTT je nástroj určený na systémové testování produktů.
2. Nastudujte požadavky na testování TGC a na testovací nástroje.
3. Navrhněte a implementujte LINK simulátor do TTT prostředí. LINK je protokol využívaný pro komunikaci více systémů TGC mezi sebou se sníženou režii ve srovnání s překladem zprávy do běžně využívaných protokolů.
4. Dále navrhněte a implementujte jeho disektor pro analyzátor síťových protokolů Wireshark a v prostředí TTT vytvořte testy využívající tento protokol, kterými bude ověřována správná funkcionality komponenty TGC.
5. Diskutujte nad dosaženými výsledky, nároky na údržbu implementace a možnostmi případného rozšíření.

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. HUCHENSON, Marnie L. **Software Testing Fundamentals: Methods and Metrics**. Indianapolis (USA): Wiley, 2003, 432 s. ISBN 047143020X.
2. KANER, Cem, James BACH a Bret PETTICHORD. **Lessons Learned In Software Testing: A Context-Driven Approach**. Indianapolis (USA): Wiley, 2001, 352 s. ISBN 0471081124.
3. CRISPIN, Lisa a Janet GREGORY. **Agile testing: A Practical Guide for Tester and Agile Teams**. Boston: Addison-Wesley, 2009, 576 s. ISBN 978-032-1534-460.
4. VAN VEENENDAAL, Erik a Brian WELLS. **Test Maturity Model integration TMMi: Guidelines for Test Process Improvement**. 's-Hertogenbosch (Nizozemí), Uitgeverij Tutein Nolthenius, 2012, 352 s. ISBN 9490986100.
5. CARNEGIE MELLON UNIV. SOFTWARE ENGINEERING INST. **The Capability Maturity Model: Guidelines for Improving the Software Process**. Boston: Addison-Wesley, 1994, 456 s. ISBN 02-015-4664-7.
6. OREBAUGH, Angela et al. **Wireshark & Ethereal: Network Protocol Analyzer Toolkit**. Gilbert Ramirez. Boston: Syngress, 2007, 448 s. ISBN 15-974-9073-3.
7. FLYNT, Clif. **Tcl/Tk: A Developer's Guide**. 3. vyd. Waltham (USA): Morgan Kaufmann, 2012, 816 s. ISBN 0123847171.

Vedoucí diplomové práce:

Ing. Michal Bližňák, Ph.D.

Ústav informatiky a umělé inteligence

Datum zadání diplomové práce:

21. února 2014

Termín odevzdání diplomové práce:

20. května 2014

Ve Zlíně dne 21. února 2014

prof. Ing. Vladimír Vašek, CSc.
děkan



doc. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

ABSTRAKT

Tato práce si klade za cíl zařadit do softwarového ekosystému společnosti Acision vnitřní nedostatečně používaný komunikační protokol a pro jeho rozvoj poskytnout podpůrné nástroje. V práci je rozebrán obecný postup vývoje softwaru, možné přístupy a typy testování softwaru a konkrétní implementace těchto postupů ve společnosti Acision. V praktické části je popsán protokol LINK, který se používá ke komunikaci s nízkou režii, jeho simulátor a disektor pro analyzátor síťového provozu Wireshark. Důraz je kladen na prakticky využitelný výstup těchto nástrojů.

Klíčová slova: UGC, Universal Gateway Component, TGC, Test Gateway Component, TTT, Tcl Test Tool, Wireshark, disektor, LINK, Acision, testování, software, Linux

ABSTRACT

This thesis aims to include an underused internal communication protocol LINK into the Acision software ecosystem and to offer support tooling to accelerate its adoption. General approaches to software development are detailed and then the available types of software testing and specific implementations of these processes in Acision. The practical part of the thesis describes the LINK protocol that is used for low-overhead communication, its Tcl simulator and a dissector for Wireshark network analyser. Strong focus on usable output of these tools is expressed.

Keywords: UGC, Universal Gateway Component, TGC, Test Gateway Component, TTT, Tcl Test Tool, Wireshark, dissector, LINK, Acision, testing, software, Linux

Rád bych poděkoval mému vedoucímu práce Ing. Michalu Bližňákovi, Ph.D. za cenné rady, připomínky a odborné vedení.

Díky zasluhují také můj vedoucí-konzultant Karel Berkovec a kolegové Tomáš Majer, Aleš Kuchař a Ondra Pančocha za podporu a poskytnutí možnosti věnovat se tématu z praxe.

Děkuji také mé rodině a všem ostatním, kteří mě během přípravy práce podporovali.

„Umění není nikdy dokončeno, pouze opuštěno.“

LEONARDO DA VINCI

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně

.....
podpis diplomanta

OBSAH

OBSAH	7
ÚVOD	10
I. TEORETICKÁ ČÁST	11
1 VÝVOJ SOFTWARE	12
1.1 KLASICKÉ PŘÍSTUPY	12
1.1.1 VODOPÁDOVÝ MODEL	12
1.1.2 V-MODEL	12
1.1.3 W-MODEL	12
1.1.4 INKREMENTÁLNÍ VÝVOJ	13
1.2 AGILNÍ PŘÍSTUPY	14
1.2.1 EXTRÉMNÍ PROGRAMOVÁNÍ	14
1.2.2 SCRUM	15
1.2.3 KANBAN.....	15
2 TESTOVÁNÍ SOFTWARE	16
2.1 TESTOVACÍ METODY	17
2.2 SKŘÍŇKOVÝ PŘÍSTUP	17
2.3 ÚROVNĚ TESTOVÁNÍ	18
2.4 TYPY TESTOVÁNÍ	19
2.5 TESTOVÁNÍ V RÁMCI AGILNÍCH METODIK	21
3 SPECIFIKA VÝVOJE A TESTOVÁNÍ SOFTWARE V ACISIONU	22
4 SÍTĚ	23
4.1 SÍŤOVÉ VRSTVY A PROTOKOLY	23
4.1.1 SÍŤOVÝ MODEL ISO/OSI.....	23
4.1.2 SÍŤOVÝ MODEL TCP/IP (DoD)	26
4.2 SÍŤOVÁ ANALÝZA	27
4.2.1 ANALÝZA SÍŤOVÉHO PROVOZU.....	28
5 WIRESHARK	29
II. PRAKTICKÁ ČÁST	33
6 TGC A UGC	34
6.1 OBECNÉ INFORMACE	34
6.2 KONFIGUROVATELNOST	35
7 LINK	38

7.1	OBEČNÉ INFORMACE	38
7.2	STRUKTURA	39
7.3	OPERAČNÍ KÓDY	39
7.3.1	KEEP-ALIVE.....	40
7.3.2	BIND REQUEST/RESPONSE	41
7.3.3	RELEASE REQUEST/RESPONSE, CLOSE REQUEST	41
7.3.4	NAMES UPDATE	42
7.3.5	DATA REQUEST/RESPONSE	42
7.4	VÝSLEDKY OPERACÍ.....	43
8	TTT.....	44
8.1	SIMULÁTORY.....	44
8.2	TESTOVACÍ SCÉNÁŘE.....	45
8.3	KONFIGURAČNÍ SCÉNÁŘE	46
8.4	PLÁNOVÁNÍ.....	46
8.5	VÝSTUP	47
8.6	AUTOMATIZACE.....	47
9	LINK SIMULÁTOR PRO TTT.....	49
9.1	OBEČNÉ API	49
9.1.1	GBGLINK_I_SETUP.....	49
9.1.2	GBGLINK_RELEASE.....	49
9.1.3	GBGLINK_I_SEND	50
9.1.4	GBGLINK_RCV	50
9.1.5	GBGLINK_SEND_NAMESUPDATE.....	50
9.1.6	GBGLINK_RECEIVE_NAMESUPDATE.....	50
9.1.7	GBGLINK_DUMP_PACKET	50
9.2	KLIENSKÉ API.....	50
9.2.1	GBGLINK_OPEN	51
9.2.2	GBGLINK_CLOSE.....	51
9.2.3	GBGLINK_CLIENT_BIND.....	51
9.2.4	GBGLINK_CLIENT_KEEAPLIVE	51
9.2.5	GBGLINK_CLIENT_DATA	51
9.3	SERVEROVÉ API	52

9.3.1	GBGLINK_LISTEN.....	52
9.3.2	GBGLINK_UNLISTEN	52
9.3.3	GBGLINK_SERVER_BIND.....	52
9.3.4	GBGLINK_SERVER_KEEPLIVE.....	53
9.3.5	GBGLINK_SERVER_DATA.....	53
9.4	INTERNÍ A POMOCNÉ API	53
9.4.1	GBGLINK_I_VERIFY_OPCODE	53
9.4.2	GBGLINK_I_ENCODE_HEADER.....	53
9.4.3	GBGLINK_I_DECODE_HEADER_OPCODE	54
9.4.4	GBGLINK_I_DECODE_HEADER_DATASIZE	54
9.4.5	GBGLINK_I_DECODE_HEADER.....	54
9.5	TESTOVACÍ API.....	54
9.5.1	CL_GBGLINK_APITEST.....	54
9.6	UKÁZKA POUŽITÍ V PRAXI	55
10	DISEKTOR PROTOKOLU LINK PRO WIRESHARK.....	58
10.1	ZÁKLADNÍ FUNKCE.....	59
10.1.1	PROTO_REGISTER_UGCLINK	59
10.1.2	PROTO_REG_HANDOFF_UGCLINK	59
10.1.3	DISSECT_UGCLINK.....	59
10.2	POMOCNÉ FUNKCE	60
10.2.1	UGCLINK_PARSE_ENDPOINTS	60
10.2.2	UGCLINK_PARSE_NAMES.....	60
10.2.3	UGCLINK_PARSE_PARAMS.....	60
10.3	UKÁZKA POUŽITÍ V PRAXI	60
ZÁVĚR		62
ZÁVĚR V ANGLIČTINĚ.....		64
SEZNAM POUŽITÉ LITERATURY.....		66
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK.....		68
SEZNAM OBRÁZKŮ		71
SEZNAM TABULEK.....		72
SEZNAM PŘÍLOH.....		73

ÚVOD

Ve světě narůstající komplexity softwaru a s tím spojeného rozmachu pomocných vývojových nástrojů zůstává testování stále poměrně nerozvíjenou, takřka opomíjenou aktivitou. Tester je ale po vlastníkovvi produktu a vývojáři dalším, kdo má moc rozhodnout o dalším osudu a směru vývoje softwaru. Je to jeho přístup k testování, který odhaluje nevyřešené problémy a který může změnit priority vývoje.

Cílem této práce je zdokumentovat proprietární komunikační protokol LINK, který je k dispozici uvnitř firemního prostředí, a vytvořit podpurné nástroje, které usnadní jeho nasazení do praxe. Zejména je nutno vytvořit simulátor jeho klienta a serveru, které umožní testovací provoz před zavedením u zákazníka, a disektor pro analyzátor síťových protokolů Wireshark, který bude možno použít pro identifikaci případných chyb. Protokol poté bude možné použít místo jiných řešení, která jsou pro daný scénář použití méně vhodná.

Práce obsahuje popis jednotlivých přístupů k vývoji softwarových produktů a také metody jejich testování. Jsou popsány jak konzervativní metody používané již desítky let, tak i poměrně nové agilní metody, které se snaží odstranit některé z nedostatků konvenčních přístupů. Dále je zpracována konkrétní aplikace těchto metod ve společnosti Acision, jejich odlišnosti, používané procesy a výhled, jak se v budoucnu budou tyto přístupy ve společnosti vyvíjet.

V praktické části diplomové práce je popsán vývoj simulátoru klienta i serveru protokolu LINK pro platformu Tcl a jsou zdokumentována všechna rozhraní určená k použití uživatelem, je definován soubor jednotkových testů, které ověřují správnou funkci simulátoru. Následuje návrh a popis všech náležitostí potřebných pro implementaci disektoru protokolu LINK pro analyzátor síťového provozu Wireshark. Je popsán seznam dalších případných vylepšení a možností rozvoje vytvořených nástrojů, které přesahují rámec této práce.

I. TEORETICKÁ ČÁST

1 VÝVOJ SOFTWARE

Pro úspěšný vývoj softwaru v týmech je třeba adoptovat určitou vývojovou metodologii. Během let vzniklo mnoho takových postupů. Při vývoji softwaru je třeba projít následujícími kroky [1]:

- Požadavky – specifikace nároků uživatele na software – co má software umět
- Design – analýza požadavků a architektonický návrh – jak se to bude dělat
- Implementace – programování, vytváření samotného kódu
- Verifikace – tvorba testovacích plánů, testů, ověření správné funkčnosti softwaru
- Údržba – nasazení softwaru u zákazníka a následné udržování funkčnosti

1.1 Klasické přístupy

1.1.1 Vodopádový model

Ve vodopádovém modelu [2] se jednotlivé kroky vývoje vykonávají po sobě. Práce na další aktivitě začínají, jakmile je předchozí dokončená. V případě, že se v libovolném bodě narazí na problém, je třeba se vrátit do předchozí fáze, aby mohl být opraven. Celý projekt je vytvořen v jednom cyklu. Model dostal své jméno podle toho, že přechody mezi jednotlivými aktivitami připomínají kaskádový vodopád.

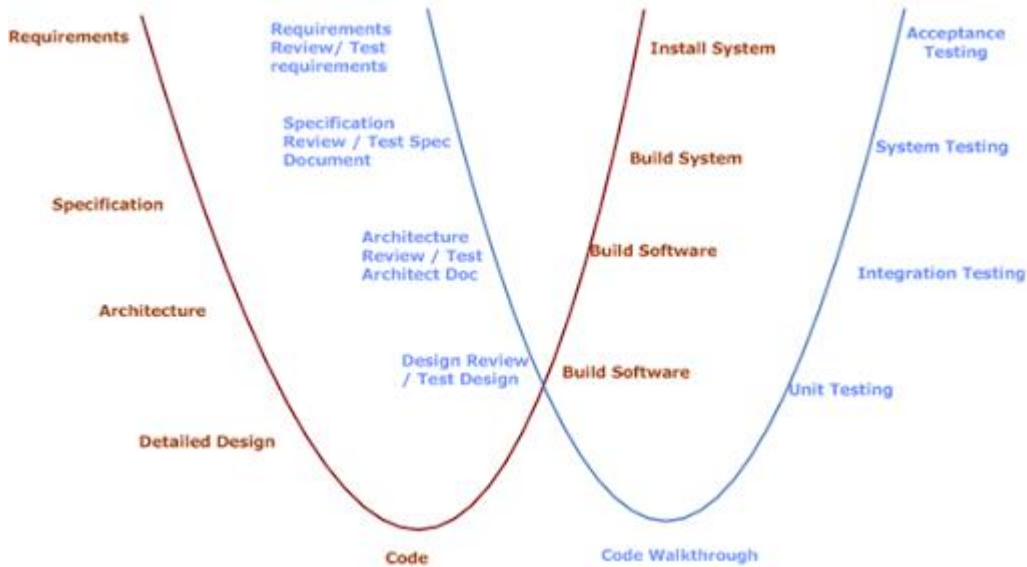
1.1.2 V-model

V-model [2] rozvíjí vodopádový model tím, že klade větší důraz na testování. To již není jen jednou z několika krabiček na schématu, ale tvoří reakci na každý vývojový krok. Jak napovídá název modelu, lze si představit, že analýza, návrh a tvorba kódu postupně probíhají po hlavní diagonále písmene V, na jeho hrotu je hotový zdrojový kód, po vedlejší diagonále pak pokračují jednotlivé kroky testování. Jako ukázkou V-modelu lze použít Obr. 1, ze kterého je třeba uvažovat jen vnější ramena hypotetického písmene W.

1.1.3 W-model

V-model stále ještě trpí na poměrně vysokou cenu opravy chyby zjištěné v pozdějších fázích testování, jelikož testování sice dostává svůj prostor, ale stále se dostává na řadu až po dokončení vývoje. Na potlačení tohoto problému byl navržen W-model [3]. Na obrázku

(Obr. 1) lze vidět, že ověřování a kontroly vytvořených návrhů, dokumentů a zdrojového kódu probíhají souběžně s vývojem a dochází k určitému překryvu těchto aktivit. Díky tomu lze chybu zaznamenat dříve a snížit tak náklady na její opravu.



Obr. 1. Schéma W-modelu [3].

1.1.4 Inkrementální vývoj

Inkrementální vývoj [2] softwaru se snaží k celku přidávat jednotlivé hotové části, podobně jako se z cihel staví zeď [32]. Postup lze ilustrovat na obrázku (Obr. 2). Lze říct, že tvorba projektu inkrementálním způsobem připomíná postup pomocí mnoha malých vodopádových modelů.



Obr. 2. Znázornění inkrementálního vývoje [32].

1.2 Agilní přístupy

Agilní přístupy využívají iterativního vývoje k dosažení vyšší efektivity práce. Iterativní vývoj nepracuje s částmi, jako spíše s celkem, který postupným vývojem vylepšuje celý najednou. Rozdíl proti inkrementálnímu vývoji ilustruje obrázek (Obr. 2). Díky tomu, že lze vést dialog se zákazníkem ohledně momentálního směřování vývoje, lze lépe naplnit jeho požadavky.



Obr. 3. Znárodnění iterativního vývoje [32].

Agilní vývoj je charakterizován svým manifestem [4], který stojí na následujících principech:

- „jednotlivci a interakce před procesy a nástroji
- fungující software před vyčerpávající dokumentací
- spolupráce se zákazníkem před vyjednáváním o smlouvě
- reagování na změny před dodržováním plánu“

Agilní metodiky často bývají mylně vykládány jako metodiky, ve kterých se nepracuje na dokumentaci. Je proto důležité si uvědomit, že vysoká důležitost hodnot vlevo neznamená, že hodnoty vpravo by měly být zanedbány či dokonce vynechány.

1.2.1 Extrémní programování

Extrémní programování [2] (XP) se soustředí na zvýšení kvality softwaru za současného zrychlení reakcí na vývoj požadavků zákazníka. Pro XP jsou proto podstatné krátké vývojové cykly, aby bylo možné udržovat častou komunikaci se zákazníkem. Dalšími charakteristickými rysy jsou trvalé ověřování napsaného kódu (v praxi aplikované jako programování ve dvojicích), pokrývání kódu jednotkovými testy, mělká struktura

managementu a odklady implementace nevyžadovaných vlastností projektu. Nevýhodou je složitá tvorba celkového návrhu vzhledem k trvale se měnící formě projektu.

1.2.2 Scrum

Scrum [5] (původně pojem pro skrumáž při rozehrávce v ragby) vznikl původně jako metodologie v automobilovém průmyslu a postupně byl aplikován do softwarové oblasti. Základním principem je podobně jako v XP uvědomění si, že problematika projektu na jeho začátku nemůže být plně pochopena ani definována, takže se vývoj posouvá po malých krůčcích, mezi kterými dochází k úpravě vnějších požadavků.

Ve scrumové metodologii se štěpí následující role: Product Owner (vlastník produktu, který reprezentuje zákaznickovy potřeby), vlastní vývojový tým a Scrum Master. Jeho povinností je pracovat na dodržování pravidel Scrum procesu, chránit vývojový tým před rušivými vlivy a umožnit tak týmu co nejefektivnější práci. Někdy proto bývá označován jako služebník-vůdce, jelikož svou moc využívá k upřednostnění potřeb ostatních členů týmu.

Scrum probíhá v iteracích (tzv. sprintech), které trvají obvykle jeden až čtyři týdny. Před každým sprintem se děje plánovací porada, kdy se připraví rámeček práce, který bude vykonán během sprintu. Během sprintu probíhá každodenní Scrum meeting, kde si členové týmu vyměňují novinky a zkušenosti z vývoje. Na závěr sprintu připadá ukázka odvedené práce vlastníkovému produktu. Ten pak poskytuje zpětnou vazbu, díky které lze rychle reagovat na změny požadavků. V závěru každého sprintu by měl software být ve stavu vhodném k vydání, i když se vydávat nemusí.

1.2.3 Kanban

Kanban [7] (z japonštiny doslova „billboard“ [6]) stejně jako scrum původně pochází z jiného než softwarového prostředí a sdílí podobné agilní principy. Hlavní motivací je snaha omezit množství započaté práce a na výrobcích pracovat až ve chvíli, kdy jsou potřeba, dle systému JIT (Just-in-time). Kanban používá tabuli pro vizualizaci probíhajících prací, omezuje množství započatých aktivit a není dělený do daných časových období [8]. Plán probíhajících prací lze kdykoliv měnit.

2 TESTOVÁNÍ SOFTWARE

Softwarové testování je proces ověřování a udržování kvality softwaru. Kvalita je definována jako úroveň splnění požadavků zainteresovaných stran, zpravidla zákazníka. Jinými slovy, testování ukazuje, zda je výsledný produkt vhodný pro dané použití, které pro něj bylo specifikováno.

Testování lze také chápat nikoliv jako kvalitativní proces, ale i čistě jako hledání chyb. Od tohoto pohledu se ale v moderních metodikách vývoje softwaru upouští, jak se testování od svých začátků jako součást debugingu (lze přeložit jako „odstraňování chyb“) postupně stává jednou ze součástí procesu vývoje softwaru [9].

Jsou zaznamenány tyto fáze vývoje softwarového testování [10]:

- Do roku 1956 – debugging – „nebyl jasně daný rozdíl mezi testováním a debuggingem“
- 1957–1978 – demonstrace – „testování a debugging jsou jasně odděleny, pomocí testů se ukázalo, že software splňuje požadavky“
- 1979–1982 – destrukce – „cílem bylo najít chyby“
- 1983–1987 – ohodnocení – „je poskytnuto ohodnocení produktu a změření kvality“
- 1988–2000 – prevence – „testy demonstrovaly, že software splňuje specifikace, rozpoznávaly chyby a předcházely chybám“

Testování lze rozdělit dle několika kritérií a úhlů pohledu. Existuje základní rozdělení na metody, kde se lze setkat se statickým a s dynamickým testováním. Dále je možno dělit pomocí skříňkového přístupu na testování metodou černé, bílé či šedé skříňky. Další rozdělení je pomocí úrovně, kde se testování provádí, tj. jednotkové, integrační, testování rozhraní komponent a akceptační testování. Konečné rozdělení je pak na typy testování. V každém rozdělení existuje několik metod či přístupů, které se navzájem doplňují. Není tedy vyloučeno použití více přístupů a metod najednou, naopak ve většině případů není možné zákazníkovi dokázat dostatečnou kvalitu softwaru, aniž by nebyl otestován z více úhlů pohledu.

2.1 Testovací metody

Testování lze na nejvyšší úrovni dělit na statické a dynamické, neboli verifikaci, respektive validaci.

Verifikace je podle normy ČSN EN ISO 9000:2006 [11] definována jako „potvrzení prostřednictvím poskytnutí objektivních důkazů, že specifikované požadavky byly splněny“. Laicky lze říct, že verifikace je ověřování, zda software byl vytvořen správně dle požadavků zákazníka. Mezi akce verifikace patří review a inspekce kódu, kontrolní čtení a statická analýza kódu překladačem či jiným nástrojem.

Validaci definuje norma ČSN EN ISO 9000:2006 [11] jako „potvrzení prostřednictvím poskytnutí objektivních důkazů, že požadavky na specifické zamýšlené použití nebo na specifickou aplikaci byly splněny“. Validaci lze tedy laicky popsat jako ověřování, zda byl vytvořen správný software, neboli jsou správně definované požadavky zákazníka, podle kterých byl software vytvářen. Validací akce jsou spouštění kódu a ověřování funkcí softwaru.

2.2 Skříňkový přístup

K testování lze přistupovat pomocí metody skříňek, které definují znalost uspořádání a hloubku přístupu k testovanému softwaru. Existuje přístup bílé, černé a šedé skříňky [12].

1. Bílá skříňka (white-box testování)

Testování pomocí bílé skříňky [12] – strukturní testování – provádí tester, který má k dispozici úplnou specifikaci zdrojového kódu a přesně zná jeho činnost. White-box paradigma říká, že software jako průhledná skříňka má viditelné všechny vnitřní části a v kteroukoliv chvíli lze sledovat pohyb vstupních dat a jejich transformaci na výstupní data. Jsou známy všechny použité algoritmy, datové struktury i podmínky větvení toku kódu. Jednotlivé testy mohou ověřovat například jednotlivé větve, do kterých se tok programu může rozdělit, případně jednotlivé rozsahy vstupních hodnot, které jsou pro kód či jeho dílčí části platné. Pomocí jednotkových testů lze zajistit ověření funkcionality v naprosto přesně zvolitelných bodech kódu, které pomocí end-to-end testování (neboli systémového testování) nemusí být přístupné vůbec nebo výrazně složitějšími postupy.

2. Černá skříňka (black-box testování)

Black-box testování [12] – specifikační testování – provádí typicky tester či vývojář, který se nepodílel na vývoji testovaného kódu. Black-box testování svůj název získalo z pohledu na software jako neprůhlednou krabici. Tester vidí jen data zadávaná na vstupu a data získaná z výstupu, vnitřní stav softwaru je mu skryt. Neznalost kódu někdy může dopomoci nalézt chybu ve zpracování vstupních dat díky tomu, že jsou zadána úplně jiná data, než na která je kód připravený, protože tester nebyl ovlivněn vnitřním rozložením a zpracováním očekávaného vstupu.

3. Šedá skříňka (grey-box testování)

Šedá skříňka [12] je spojením černé skříňky a bílé skříňky. Tester má přístup k softwaru pouze jako k monolitickému objektu, není možné se zaměřit na jednotlivé dílčí části. Zároveň má ale k dispozici popisy algoritmů a vnitřního uspořádání (např. zdrojový kód), pomocí kterých se může tester zaměřit na další specifika nedostupná metodami černé nebo bílé skříňky.

2.3 Úrovně testování

1. Jednotkové testy

Někdy také komponentové testy; jednotkové testy [13] se zaměřují na specifické části zdrojového kódu. Obvykle je nejmenší testovanou jednotkou funkce, v objektově orientovaném prostředí je možno přistoupit i k testování na úrovni jednotlivých tříd. Jednotkové testy může vytvářet přímo vývojář během práce na jimi ověřovaném kódu. Samy o sobě nedokazují spolehlivou funkci celku, který součet jednotek vytváří, ale ukazují, že každá základní jednotka dokáže správně fungovat sama o sobě.

2. Integroční testy

Integroční testy [13] ověřují, že postupným spojováním jednotlivých testovaných komponent vznikne řešení, které nadále splňuje všechny požadavky na něj kladené v návrhové fázi. Často se může stát, že vinou nedostatečného pokrytí testy či nevhodné specifikace v celkovém řešení nesprávně spolupracují i komponenty, které samostatně fungují bezchybně.

3. Testy rozhraní komponent

Testy rozhraní komponent [13] jsou nad rámec integračních testů. Ověřují s větší hloubkou, přesností a náročností, zda všechna data procházejí systémem správně.

4. Systémové testy

Systémové testy [13] (či také end-to-end testy) zjišťují, zda kompletně integrovaný systém splňuje všechny na něj kladené požadavky. Ověřují nejen komplexní funkcionalitu systému jako takového, ale i to, že nemá negativní vliv na stroj, na kterém běží, a na ostatní potenciálně běžící procesy.

5. Akceptační testy

U zákazníka musí při předání softwaru proběhnout další testy (akceptační testování [13]), které ověří funkčnost softwaru dle jeho požadavků. Zároveň ale mohou být parametry testů navrženy výrobcem softwaru např. podle toho, jak byl produkt testován během vývoje.

2.4 Typy testování

1. Instalační testování

Instalace je zpravidla první úkon, který bude proveden u zákazníka, a instalační testování [13] ověřuje jeho hladký průběh. Toto je počáteční způsob, jak ověřit kompatibilitu s hardwarem, který bude zákazník používat, a s jeho softwarovou platformou (operační systém, doprovodné nástroje apod.).

2. Upgrade testování

Upgrade testování [13] má podobný cíl jako instalační testování, jen v jiné časové relaci. Lze jím ukázat, že i po nainstalování aktualizace či upgradu softwaru zůstane zachována jeho funkcionalita a nedojde k selhání vinou konfliktu v novém kódu či v přidané konfiguraci. Zároveň se také ověřuje zpětná kompatibilita.

3. Kouřové testování (smoke test)

Kouřové testování [14] (angl. smoke test nebo také sanity test) je způsob, jak ověřit naprosto základní funkcionalitu produktu v krátkém čase. Může se jednat například o jednoduchý pokus o spuštění a zastavení produktu po každé změně v kódu s následným sestavením. Lze tak rychle odhalit nejzákladnější chyby, pro jejichž detekci není třeba použít celou testovací sadu. Pokud selže kouřový test, je zbytečné dále pokračovat v

testování, jelikož to pravděpodobně ani nebude možné. Tyto testy se často používají v automatizovaných prostředích, kde se spouští po každém sestavení produktu. V rámci interního vývoje lze provedení kouřových testů brát jako akceptační testování, po kterém lze přikročit k regresnímu a integračnímu testování.

4. Regresní testování

Regresní testování [13] slouží k ověření stálé funkčnosti stávajícího kódu. Neslouží k ničemu jinému, než opakovanému spouštění testů nad již odladěným kódem, aby bylo zajištěno udržení funkcionality. U složitějších programů lze jen velmi těžko předpovědět dopad libovolných změn v kódu na celkovou stabilitu v oblastech, které se dotyčné funkcionality přímo netýkají. Během regresního testování může náhle selhat některý test – interakcí nového kódu se starým kódem vznikne regrese, která způsobí poškození či selhání již existující funkcionality, případně opětovný návrat již opravené chyby. Tyto testy se často používají v automatizovaných prostředích po úspěšném průchodu kouřových testů.

5. Funkční testování

Funkční testování [13] ověřuje, že součásti softwaru korektně plní svou funkci. Funkční testování testuje všechny funkční požadavky, které jsou na software kladené. Jedná se tedy zvláště o možnost využívat rozličné funkcionality a uživatelsky viditelné úkony.

6. Nefunkční testování

Nefunkční testování [13] se naopak věnuje nefunkčním požadavkům, které nejsou svázané s určitou uživatelskou aktivitou nebo jednou funkcionalitou. Jedná se zvláště o aspekty spojené s výkonem, škálovatelností, spolehlivostí a bezpečností softwaru.

7. Destruktivní testování

Destruktivní testování [10] (či testování robustnosti či zátěžové testování) má za úkol systém poškodit či vyřadit z provozu pomocí běžných prostředků, čímž se otestuje odolnost vnitřních částí a preciznost vstupních modulů. Destruktivní testování (či testování robustnosti) pomocí poškozených vstupů se nazývá fuzzing (mlžení). Na testy robustnosti lze také použít mutační testování, ve kterém se upravuje testovaný systém na úrovni zdrojového kódu – např. změna znaménka v rovnici.

8. Testování výkonu (performance test)

Testování výkonu [10] se dělí na několik částí. Jednak se pomocí něj měří nejvyšší dosažitelný stabilní výkon, jednak škálovatelnost, dále lze ověřovat stabilitu systému při špičkově zvýšených zatíženích či provoz s velkými objemy dat. Další názvy, se kterými se lze setkat, jsou load testing, stability testing, volume testing nebo stress testing. Ačkoliv se každý z vyjmenovaných testů specializuje na mírně odlišný aspekt výkonu, názvy se obvykle volně zaměňují.

2.5 Testování v rámci agilních metodik

Některé z mladších metodik vývoje softwaru, zejména agilní vývoj nebo extrémní programování, přistupují k testování odlišným způsobem. Zatímco ve většině vývojových modelů velká část testování probíhá po dokončení návrhu a vývoje, v agilních metodikách se testování stává součástí celého průběhu od fáze návrhu. Aplikuje se tzv. testy řízený vývoj [15] (TDD – Test-Driven Development) – po dokončení prvního návrhu se dle specifikace vytvoří jednotkové testy a kód se píše tak, aby tyto testy začaly procházet. Pokud se během programování kódu narazí na výjimky nebo případy, které je třeba ošetřit, přidávají se do existující testující sady. Tato metoda zvyšuje zatížení vývojářů a může mít za následek zdržení vytvoření prvního prototypu, ale zvyšuje spolehlivost a v dlouhodobém horizontu urychluje dodávání nových verzí [15].

3 SPECIFIKA VÝVOJE A TESTOVÁNÍ SOFTWARE V ACISIONU

V Acisionu tvorba nové verze produktu začíná komunikací produktového manažera se zákazníkem. Jakmile jsou vytvořeny požadavky na novou funkčnost, jsou formálně zpracovány. Na tomto základě vzniká vysokoúrovňový návrh, detailní návrh a implementační plán. Ve stejnou dobu se sestavuje návrh testovacího plánu, vznikají metadata, podle nichž budou později naimplementovány systémové testy. Během vývoje nové vlastnosti jsou vytvářeny jednotkové testy, které jsou ověřovány při libovolném zásahu do kódu. Poté, co je k dispozici první sestavení produktu či jeho komponenty obsahující novou vlastnost, se začnou vytvářet systémové testy podle dříve navrženého plánu. Na základě výsledku nových systémových testů je možná oprava implementace. Po úspěšném feature testingu se přistoupí k regresním testům, aby bylo zjištěno, zda nedošlo k porušení zpětné kompatibility. V případě úspěchu je množina nových testů zaintegrovaná do regresní sady a nová vlastnost je uvolněna k vydání.

V případě údržby – maintenance – je proces jednodušší. Vzniká méně dokumentů a korektní testy rovnou mohou být jednotlivě začleňovány do regresní sady.

V obou případech je nutné, aby změny v kódu byly schváleny architektem a zkontrolovány dalším vývojářem. Nové testy musí být zkontrolovány dalším testerem, správcem regresní sady a musí je odsouhlasit vývojář.

Softwarový vývoj i testování v Acisionu se v současné době transformují z klasického přístupu do agilního. Postupně jsou do vývojových procesů začleňovány metodiky Scrum, Kanban a jejich varianty, týmy jsou vedeny k lepší komunikaci se zákazníkem a k lepší samoorganizaci. Dosud využívaný, opouštěný koncept spočívá v separovaných částech vývoje, kdy postupně probíhá posun z V-modelu do W-modelu. Testování se tak rozevírá větší prostor a tím se zvyšuje jeho efektivita a přínos pro vývoj.

Postupným přechodem z W-modelu do Scrumu (či Kanbanu a jejich variant) bude ve firemním prostředí docházet k úplnému prolínání jednotlivých fází vývoje. Systémové testování může začít ve stejnou chvíli, ve kterou začíná samotný vývoj dané vlastnosti, což umožňuje nejen dialog developera a testera, ale zároveň i jejich spolupráci při tvorbě nové vlastnosti produktu. To má za následek lepší interdisciplinární znalosti, kdy se tester dovídá více informací o vývoji a naopak, a možnost prevence chyb v implementaci i v testech.

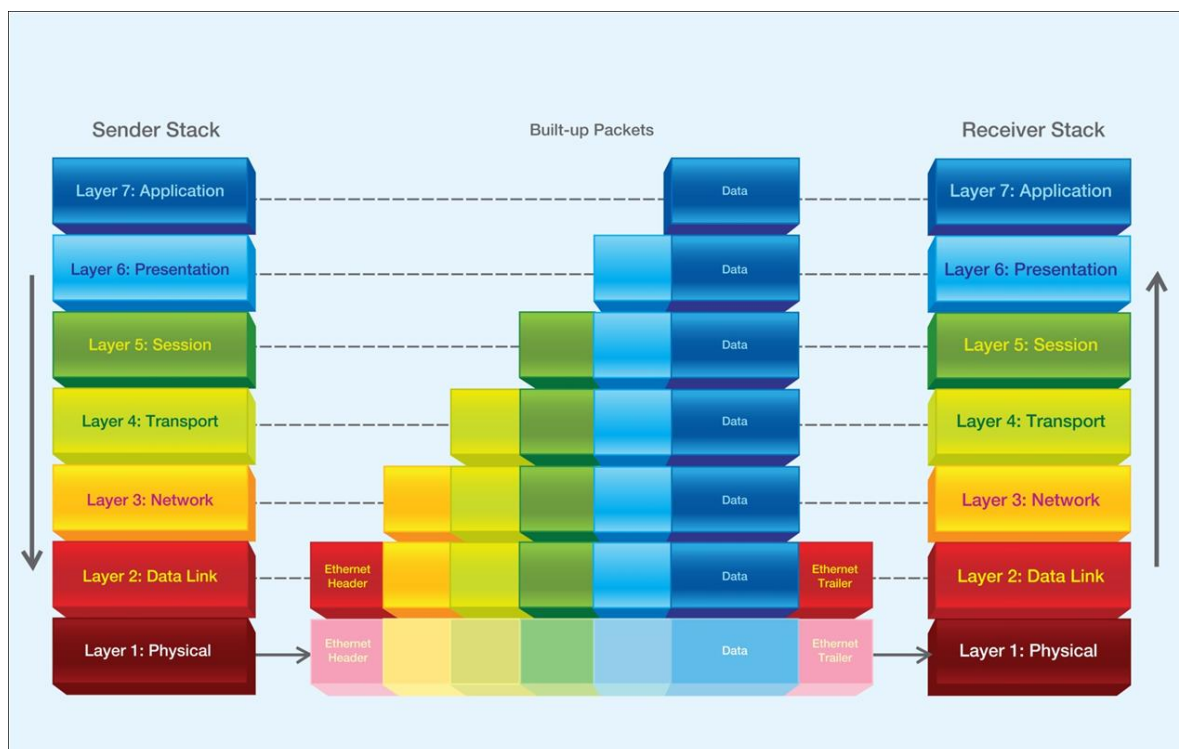
4 SÍŤ

4.1 Síťové vrstvy a protokoly

4.1.1 Síťový model ISO/OSI

Organizace ISO v 80. letech vytvořila model OSI [16], viz obrázek (Obr. 4), aby popsala komunikaci jednotlivých síťových zařízení a protokolů. Model obsahuje celkem sedm vrstev, od nejvyšší:

- aplikační (Application)
- prezentační (Presentation)
- relační (Session)
- transportní (Transport)
- síťová (Network)
- linková (Data Link)
- fyzická (Physical)



Obr. 4. Síťový model ISO/OSI [17].

1) Fyzická vrstva

Nejnižší vrstva ISO/OSI modelu specifikuje elektrické a mechanické vlastnosti vyžadované pro přenos dat skrze fyzické médium, ať už pomocí kabelu nebo bezdrátově. Specifikace zahrnují změny napětí, jejich časování, datové toky, maximální vzdálenosti kabeláže, fyzické médium a konektor, topologii a fyzické rozvržení sítě.

Data zpracovávaná fyzickou vrstvou jsou bity, které jsou reprezentovány světelnými pulzy nebo výkyvy v napětí. Řazení a zpracování sekvencí bitů má na starosti linková vrstva.

2) Linková vrstva

Linková vrstva udržuje spojení mezi dvěma koncovými body, které jsou zpravidla nazývány uzly. Linková vrstva komunikuje pomocí rámců, v nichž jsou jednotlivé bity řazeny a organizovány. Linková vrstva je také zodpovědná za kontrolu toku – časování odesílání a přijímání dat, aby byla ideálně využita kapacita linky – a za ošetření chyb fyzické vrstvy.

Síťová zařízení provozovaná na této vrstvě jsou L2 switch (přepínač) a bridge (most). L2 switch snižuje vytížení sítě díky tomu, že zasílá data jen na port, ke kterému je připojený cílový počítač, namísto toho, aby data rozeslal na všechny porty. Bridge poskytuje možnost rozdělit síť pomocí tabulek MAC adres jejích počítačů do dvou částí a filtrovat provoz mezi těmito dvěma stranami mostu.

3) Síťová vrstva

Tato vrstva komunikuje pomocí paketů, k označení počítačů se používá logické adresování. Logické adresy nejsou trvalé a jsou definovány softwarově (na rozdíl od MAC adresy, kterou má každý síťový adaptér napevno nastavenou). Příkladem logické adresy jsou IP adresy nebo IPX adresy. Protokoly používající logické adresy jsou směrovatelné, jelikož v jejich adresních schématech je identifikována síť nebo podsíť.

Síťová vrstva je také zodpovědná za vytvoření virtuálního okruhu (tj. logického, nikoliv fyzického spojení) mezi body či uzly. Uzel je zařízení, které má přiřazenou MAC adresu – typicky to jsou počítače, tiskárny, chytrá zařízení a routery.

Tato vrstva je také zodpovědná za směrování (routing), L3 přepínání a preposílání paketů. Směrování je preposílání paketů z jedné sítě nebo podsítě do druhé, díky čemuž mohou uzly komunikovat i mezi sítěmi, zatímco bez směrování by mohly komunikovat jen v

rámci své dané sítě či podsítě. Dalo by se tedy říci, že směrování tvoří základní stavební kámen internetu.

Síťová vrstva poskytuje další možnosti kontroly toku a také ošetření chyb linkové vrstvy.

Síťová zařízení provozovaná na této vrstvě jsou L3 přepínače a routery.

Všechny dosavadní vrstvy ISO/OSI modelu zahrnují hardware, zatímco následující vrstvy jsou již implementovány softwarově.

4) Transportní vrstva

Transportní vrstva má za úkol přenášet data z jednoho uzlu na druhý. Mezi uzly je možný transparentní přenos dat, je poskytována end-to-end kontrola toku, detekce a oprava chyb.

Protokoly transportní vrstvy vytváří spojení mezi konkrétními porty na různých počítačích a vytvoří virtuální okruh. Transportní protokoly na obou počítačích ověří, že oba konce jsou připraveny začít datový přenos. Jakmile je tato synchronizace dokončena, jsou data odeslána. Zatímco jsou data přenášena, transportní protokol na obou strojích monitoruje datový tok a detekuje chyby přenosu. Pokud jsou nalezeny chyby přenosu, transportní protokol poskytne zotavení se z chyby.

Dva z protokolů nejčastěji spojovaných s transportní vrstvou jsou spojově orientovaný TCP a na zprávy orientovaný UDP.

Spojově orientovaný protokol se snaží spolehlivě doručit všechny zasláné pakety, k čemuž používá ověřování doručení a vícenásobné pokusy o zaslání. Na zprávy orientovaný protokol pakety odešle bez dalšího ověřování, zda dorazí, a případné zajištění spolehlivosti je záležitostí aplikace.

Transportní vrstva zajišťuje logické adresování portů. Mnoho portů má definované využití, např. port 80 slouží ke komunikaci přes HTTP protokol, port 53 je použit pro DNS, port 25 využívá SMTP protokol.

5) Relační vrstva

Poté, co transportní vrstva ustaví virtuální spojení, mezi dvěma procesy na dvou různých počítačích se vytvoří komunikační relace. Relační vrstva spravuje vytvoření, monitorování a ukončení relace pomocí virtuálních okruhů ustavených transportní vrstvou.

Důležitou funkcí relační vrstvy je zvolit, zda se komunikace v rámci relace posílá jako full-duplex (plný duplex) nebo half-duplex (poloviční duplex) zprávy. Obě varianty umožňují

obousměrnou komunikaci, ale v případě half-duplex probíhá komunikace v libovolnou chvíli vždy jen jedním směrem.

Zatímco transportní vrstva vytváří spojení mezi dvěma počítači, relační vrstva vytváří spojení mezi dvěma procesy. Aplikace může mít spuštěných více procesů najednou, aby mohla vykonávat svou činnost.

6) Prezentační vrstva

Hlavní činností prezentační vrstvy je překlad dat. Když jsou data zaslána od odesilatele k příjemci, jsou přeložena v prezentační vrstvě do společného formátu. Jakmile jsou data přijata u příjemce, prezentační vrstva je transformuje ze společného formátu zpět do formátu podporovaného aplikací. Na této vrstvě probíhá překlad protokolů – tedy proces, díky kterému mohou komunikovat počítače postavené na různých platformách či s různými operačními systémy.

Na této vrstvě také probíhá komprese dat, díky čemuž není nutné přenášet tak velké množství bitů po nižších vrstvách. Rovněž tady probíhá šifrování a dešifrování dat.

7) Aplikační vrstva

Aplikační vrstva je bod, kde dochází k interakci uživatelské aplikace se sítí. V rámci aplikace jsou vytvořeny aplikační procesy a data vytvořená tímto procesem jsou předána aplikační vrstvě.

Na této vrstvě běží mnoho známých protokolů: FTP, Telnet, SMTP, POP, IMAP, HTTP, NNTP či SNMP.

4.1.2 Síťový model TCP/IP (DoD)

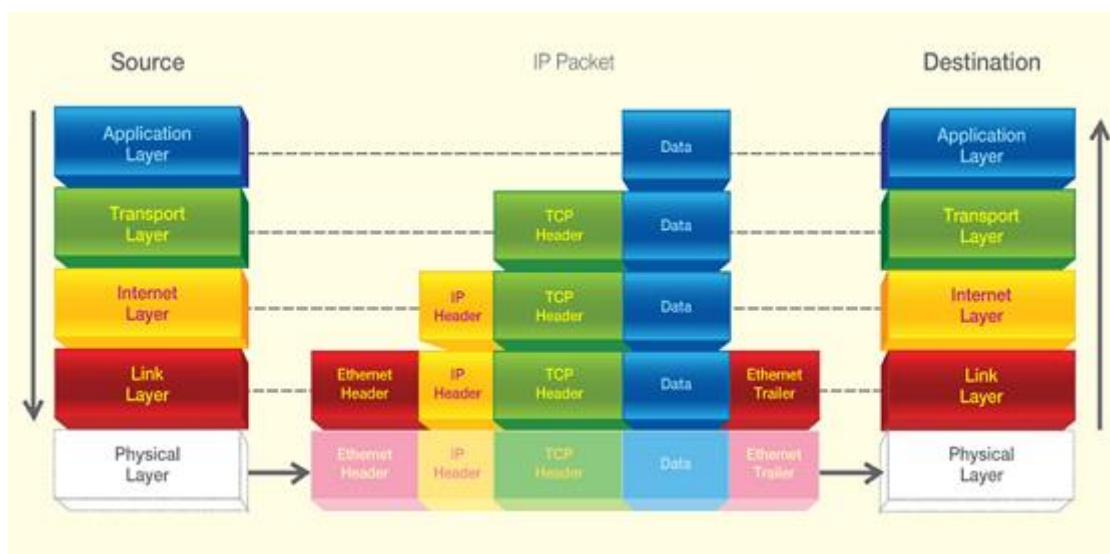
ISO/OSI model je velmi obecný, aby dle něj bylo možné popsat a vysvětlit libovolný síťový protokol. Znalost tohoto modelu umožňuje provádět kvalitní síťovou analýzu, srovnání a vyhledávání chyb. Je ale důležité upozornit, že ne všechny síťové protokoly byly vytvořeny s ohledem na ISO/OSI model [18]. Jedním z hlasů proti rigidní kategorizaci aktivit síťových protokolů dle ISO/OSI modelu je i Linus Torvalds [19]:

„Pořád hovoříme o 7vrstvě modelu, protože je to pohodlný model pro diskuzi, ale to nemá absolutně nic společného s tvorbou softwaru ve skutečném světě. Jinými slovy, je to způsob jak o věcech diskutovat, ale ne jak je implementovat. A to je důležité. [Specifikace]

jsou základ pro povídání si o věcech. Ale nejsou základem pro softwarovou implementaci.“

Základní protokoly, na kterých je postavený internet, byly vyvinuty ještě před vznikem tohoto modelu v rámci amerického Ministerstva obrany (DoD) a řídí se jeho 4vrstevným modelem, viz obrázek (Obr. 5). Model nemá pevně dané jméno, někdy je uváděn jako DoD, někdy jako TCP/IP.

Zde jsou 4 vrstvy; od nejvyšší: aplikační, transportní (host-to-host), internetová a linková (někdy síťová).



Obr. 5. Síťový model TCP/IP (či DoD) [17].

4.2 Síťová analýza

Síťová analýza [20] je proces zachytávání síťového provozu a jeho rozbor za účelem zjištění informací o něm, o použitých protokolech, jednotlivých paketech či za účelem odposlouchávání. Síťový analyzátor dekoduje datové pakety protokolů a zobrazuje síťový provoz v čitelné formě.

Sniffer je program, který monitoruje data pohybující se po síti. Nepovolený sniffer je nebezpečný, jelikož je složité jej odhalit a existuje mnoho míst, na které jej lze v síťové infrastruktuře vložit.

4.2.1 Analýza síťového provozu

Síťový provoz se dříve analyzoval pomocí hardwarových zařízení sestavených výlučně k tomuto účelu, ale postupem času a vývoje bylo možné vytvořit softwarové aplikace, které tuto funkcionalitu suplují. Analýza je dvojsečná; síťový administrátor či auditor síťové bezpečnosti ji mohou použít k odhalování slabín sítě, zjišťování chyb a monitorování provozu, nicméně ke stejnému účelu ji může použít i potenciální útočník.

Možnosti využití analýzy síťového provozu [20]:

- konverze binárních dat v paketech do lidsky čitelného formátu
- analýza výkonu sítě při hledání úzkých hrdel
- detekce průniků do sítě (IDS)
- záznam síťového provozu pro forenzní účely a sběr důkazů
- analýza provozu aplikací, ověřování součinnosti s firemní politikou
- odhalování selhávajících síťových karet a jiných síťových zařízení
- určování zdroje virových nákaz a DoS útoků, detekce spywaru
- odhalování chyb při vývoji síťových aplikací
- detekce kompromitovaného počítače
- zdroj informací při studiu síťových protokolů
- reverzní inženýrství síťových protokolů za účelem tvorby kompatibilních klientů a podpůrných programů

Možnosti zneužití analýzy síťového provozu [20]:

- zachytávání nešifrovaných uživatelských jmen a hesel
- odhalování vzorů chování uživatelů na síti
- kompromitace důvěrných informací
- zachytávání a přehrávání VoIP telefonních konverzací
- mapování rozložení sítě a rozpoznávání operačního systému

Znamé analyzátory síťového provozu jsou Wireshark, tcpdump, WinDump, snoop, Ettercap.

5 WIRESHARK

V softwarovém ekosystému společnosti Acision se na prohlížení, zpracování a diagnostiku síťového provozu využívá software Wireshark. Wireshark je analyzátor síťového provozu zdarma dostupný pod svobodnou licencí GPL. Dříve byl vyvíjen pod názvem Ethereal s částí zdrojových kódů svobodně dostupných, posléze byly všechny kódy uvolněny, ale název Ethereal zůstal pod copyrightem, proto došlo ke změně názvu [21]. K dispozici jsou sestavení pro všechny majoritní desktopové operační systémy, na všech se pro zobrazení využívá knihovny GTK. Wireshark podporuje stovky protokolů (ve verzi 1.10.2 je podporováno přes 1300 protokolů a jejich variant), umožňuje načíst a zpracovat výstup desítek dalších sniffovacích a zachytávacích nástrojů. Dva hlavní formáty používané ve společnosti Acision jsou tcpdump a pcap. Ve Wiresharku je zabudovaná možnost zachytávání síťového provozu z více síťových karet najednou a filtrace zachyceného provozu podle nejrůznějších pravidel.

Spolu s hlavní GUI variantou je dodáván¹ i CLI nástroj tshark, který lze použít v systémech bez grafického výstupu či pro automatizaci činnosti, zvláště pak v unixových systémech. Stejně jako grafická varianta umožňuje zachytávání ostrého provozu či práci se záchytnými soubory.

Dále jsou dodávány nástroje editcap, mergecap a text2pcap [20]. Nástroj editcap umožňuje odstraňovat pakety ze souboru a převádět mezi různými formáty záchytných souborů, díky čemuž je možné původní surová data upravit a převést do formátu, který je vyžadován v dalším kroku zpracování. Mergecap se používá ke spojování více záchytných souborů do jednoho. Text2pcap slouží k překladu ASCII záchytných dat do záchytného souboru ve formátu libpcap/PCAP. Všechny tyto nástroje podporují čtení stejných formátů záchytných souborů jako Wireshark.

Aby měl Wireshark přímý přístup k síťovému provozu a nemusel využívat zprostředkování pomocí síťových vrstev operačního systému ani různé techniky jejich obcházení, využívá abstrakce poskytnuté v ovladači na zachytávání paketů (packet capture driver). Tento ovladač poskytuje pro unixové operační systémy knihovna libpcap, zatímco její alternativou pro OS Windows je WinPcap. Tento ovladač je dodáván jako součást hlavního instalačního balíku a napojuje se přímo na ovladače jednotlivých síťových zařízení.

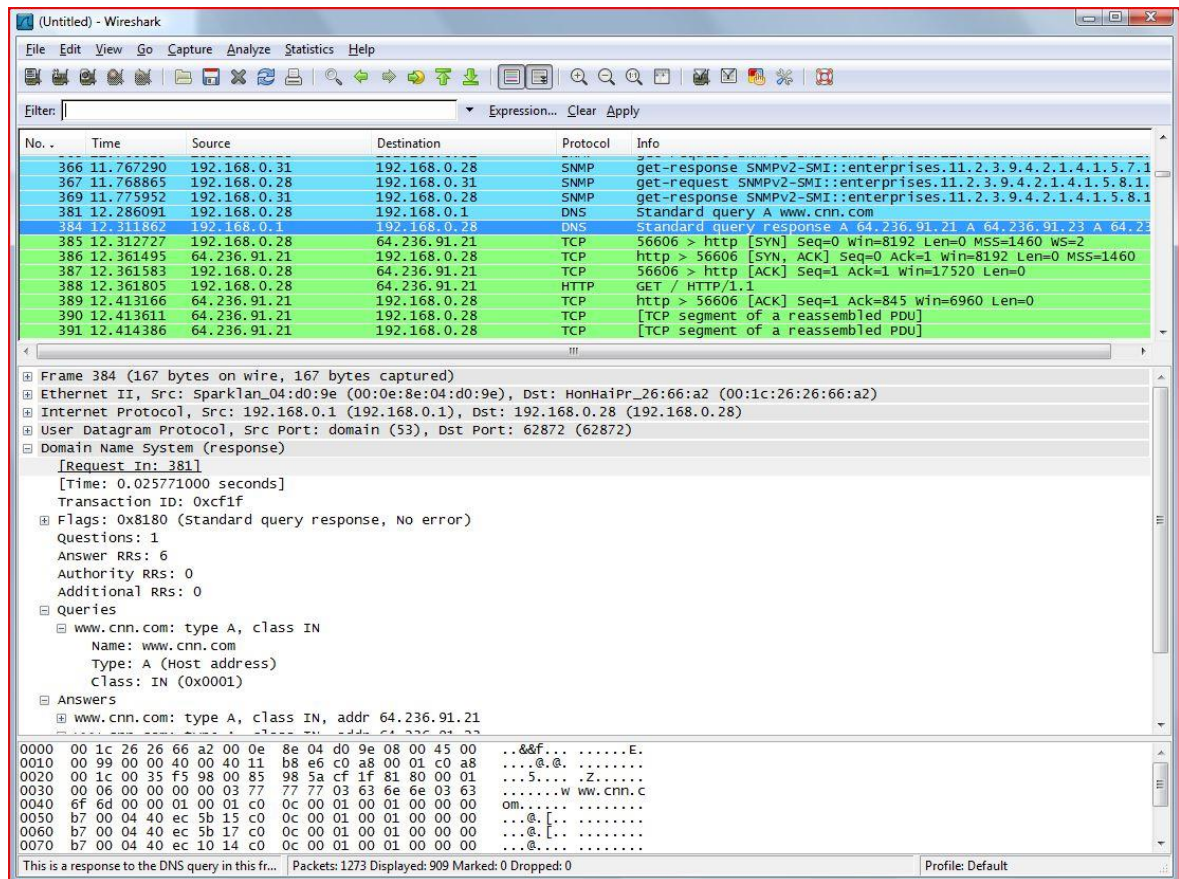
¹ Obě varianty lze stáhnout z oficiálních stránek <http://www.wireshark.org/>

Podpora pro část protokolů je ve Wiresharku zabudovaná v rámci hlavní aplikace a je možné do tohoto místa přidávat i další. Rozšiřující dekodéry pro další protokoly jsou zároveň podporované skrze zásuvné moduly – pluginy – které jsou též zvané disektory. Jeden disektor může být schopný dekódovat více protokolů či variant protokolů. Disektory v pluginech mají výhodu jednodušší správy bez potřeby znovu sestavovat celou aplikaci Wireshark. Pro podporu tvorby disektorů je k dispozici zevrubná dokumentace [22].

Je možné uživatelsky ovlivňovat funkce disektoru pomocí nastavení v GUI.

Rozhraní síťového analyzátoru Wireshark se dá rozdělit na menu, nástrojové lišty a následující tři části:

- přehled – seznam jednořádkových záznamů, každý ze záznamů odpovídá jednomu zachycenému paketu. Každý záznam obsahuje své pořadové číslo, čas jeho zachycení, zdrojovou a cílovou adresu, název a případné další informace o protokolu nejvyšší vrstvy
- podrobnosti – detaily jednoho paketu organizované do stromu, každá vrstva protokolu je v samostatném stromě
- surová zachycená data – v hexadecimální a v textové formě



Obr. 6. Výchozí styl uživatelského rozhraní analyzátoru Wireshark [23].

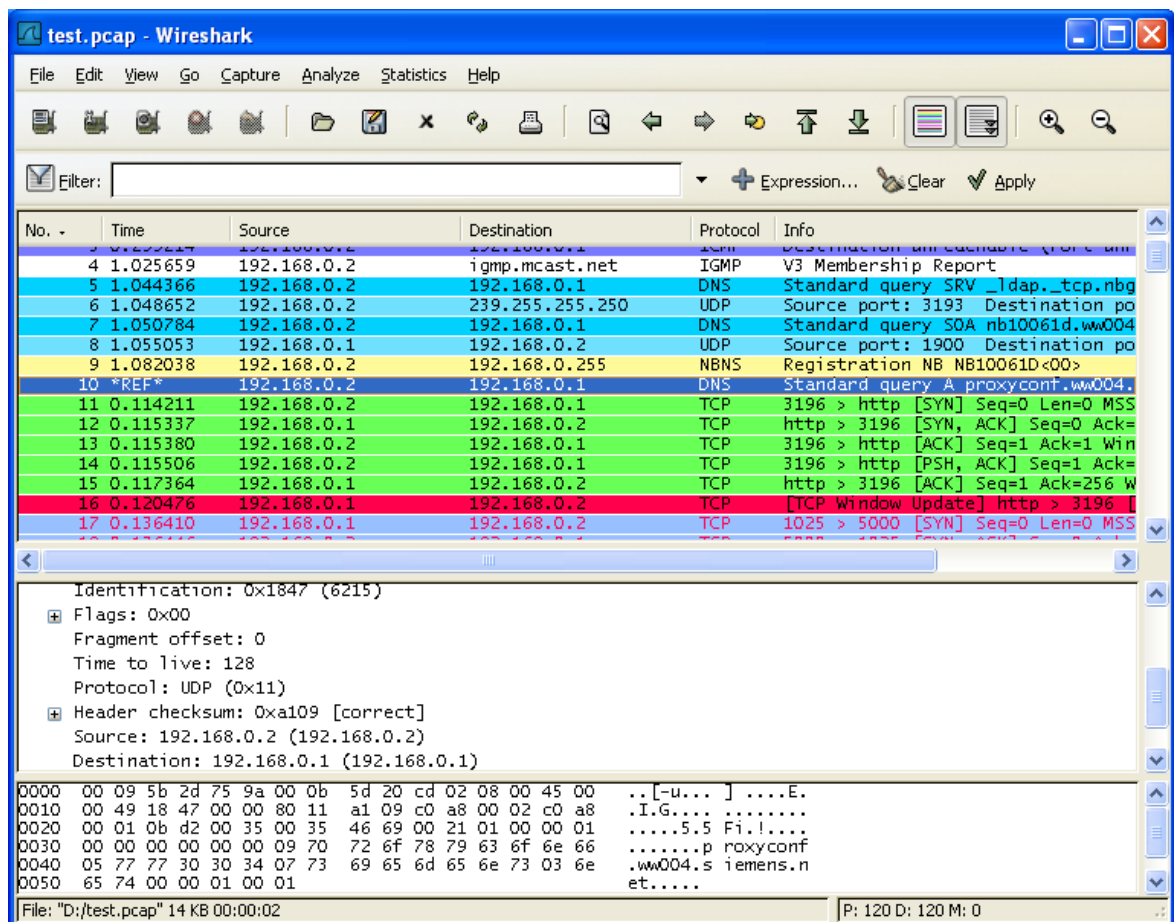
Informace zobrazované v přehledu lze měnit a konfigurovat jak na straně uživatelského rozhraní aplikace, tak i na nižší úrovni ze strany disektoru, který do uživatelského rozhraní informace dodává. Jednotlivé sloupce lze řadit a měnit jejich pořadí, jak se již u většiny aplikací stalo zvykem. Wireshark nabízí šest základních uspořádání tří výše popsaných panelů, které je v rámci uspořádání možné libovolně přesouvat. V rámci této práce bude dále uvažováno rozložení uvedené na obrázku (Obr. 6).

Klikáním myši na jednotlivé údaje v prostředním panelu se zvýrazňují odpovídající části surových zachycených dat. Stejně tak, je-li to možné, bude vybírání údajů ze surových zachycených dat zobrazovat jim odpovídající údaje z podrobností paketu.

Filtrování umožňuje zmenšit oblast prohledávaných paketů ke zjištění požadovaných dat. Wireshark podporuje dvě kategorie filtrů – filtry zachytávání a filtry zobrazení. Filtr zachytávání určuje, které dostupné síťové pakety z celkového provozu budou Wiresharkem zachyceny pro další zpracování a které budou ignorovány. Filtr zobrazení nezávisle na jeho obsahu neodstraní žádné již zachycené pakety. Tyto dva druhy filtrů nepodporují stejnou funkcionalitu. Filtry zobrazení podporují rozmanitější možnosti využití. Tyto tedy není

v grafické části možné použít na místě filtru zachytávání, tuto funkcionalitu podporuje pouze varianta pro příkazovou řádku tshark. Mnohé z podporovaných protokolů umožňují filtrování na základě jejich složek, například v IP protokolu je možné filtrovat pomocí konkrétní IPv4 adresy, specificky podle určité zdrojové nebo cílové adresy, čísla ofsetu, příznaků, délky hlavičky, TTL a mnoha dalších údajů. Jednotlivé zvolené filtry lze ukládat pod zvoleným jménem na nástrojovou lištu a při pozdějších analýzách je lze kdykoliv okamžitě nahrát a použít.

Wireshark umožňuje použít filtry také ve svém nastavení k úpravě barvy paketů zobrazených v přehledovém panelu, viz obrázek (Obr. 7). Je tak možné uživatelsky rozlišit skupiny protokolů, typy odesílatelů, příjemců či jiných specifických aspektů zpráv, za účelem jednodušší práce s větším množstvím paketů.



Obr. 7. Použití barev k odlišení různých druhů paketů [24].

II. PRAKTICKÁ ČÁST

6 TGC A UGC

UGC (Universal Gateway Component) je product Acisionu, který primárně slouží k propojování heterogenních komunikačních rozhraní s rozšiřitelnou funkcionalitou, viz obrázek (Obr. 8). K tomuto účelu původně sloužil produkt GBG (Generic Billing Gateway), na který UGC navazuje a který dále rozšiřuje. UGC se dříve používalo jako základní matrice, ze které se sestavovalo několik dalších produktů firmy Acision, tzv. reinkarnace. Mezi reinkarnace patřilo i TGC (Test Gateway Component), které se používalo k ověřování funkčnosti reinkarnací. V současné době ovšem od konceptu reinkarnací bylo upuštěno a pracuje se pouze s UGC jako takovým.

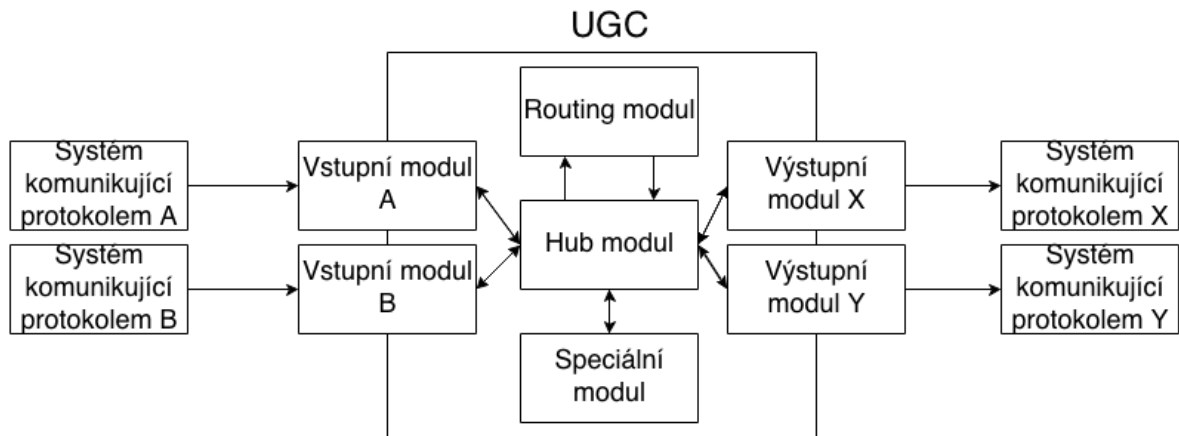
6.1 Obecné informace

UGC je modulární komponenta napsaná v jazyce C++, s jejíž pomocí je možné propojovat systémy, které využívají síťového rozhraní ke komunikaci, ale každý z nich využívá jiný protokol. UGC slouží jako mediátor v této komunikaci, který překládá pakety jednoho protokolu na druhý a tak systémům umožňuje transparentně se dorozumívat bez jakýchkoliv dodatečných úprav. Tato komponenta podporuje proprietární protokoly Acisionu, mezi nimi i protokol LINK. Je dostupná i podpora pro standardní protokoly jako jsou SMPP, Diameter, LDAP, XMPP (Jabber) a mnoho dalších. Při vstupu zprávy dovnitř této komponenty dochází k překladu přijatých dat do interní dynamické struktury (interní zprávy), která se pak používá při komunikaci uvnitř UGC.

UGC běží na platformě Linux. Konfigurace je uložena v INI souborech, základní správa procesu je poskytována init skripty, ke složitějším úkonům slouží spustitelný soubor ugcmgr. Umožňuje ukládat výpis záznamu o svém běhu s možnostmi nastavení různých úrovní podrobnosti záznamu informací. Spolehlivost je zajišťována procesem procman, který sleduje stav této komponenty a v případě neočekávaného ukončení ji opětovně spustí. Je podporováno spouštění více instancí na jednom stroji se separátní správou každé z těchto instancí. Jádro UGC i jednotlivé moduly lze nakonfigurovat, aby poskytovaly informace o svém provozu přes SNMP rozhraní. Jednotlivé čítače a jejich sémantiku lze měnit.

6.2 Konfigurovatelnost

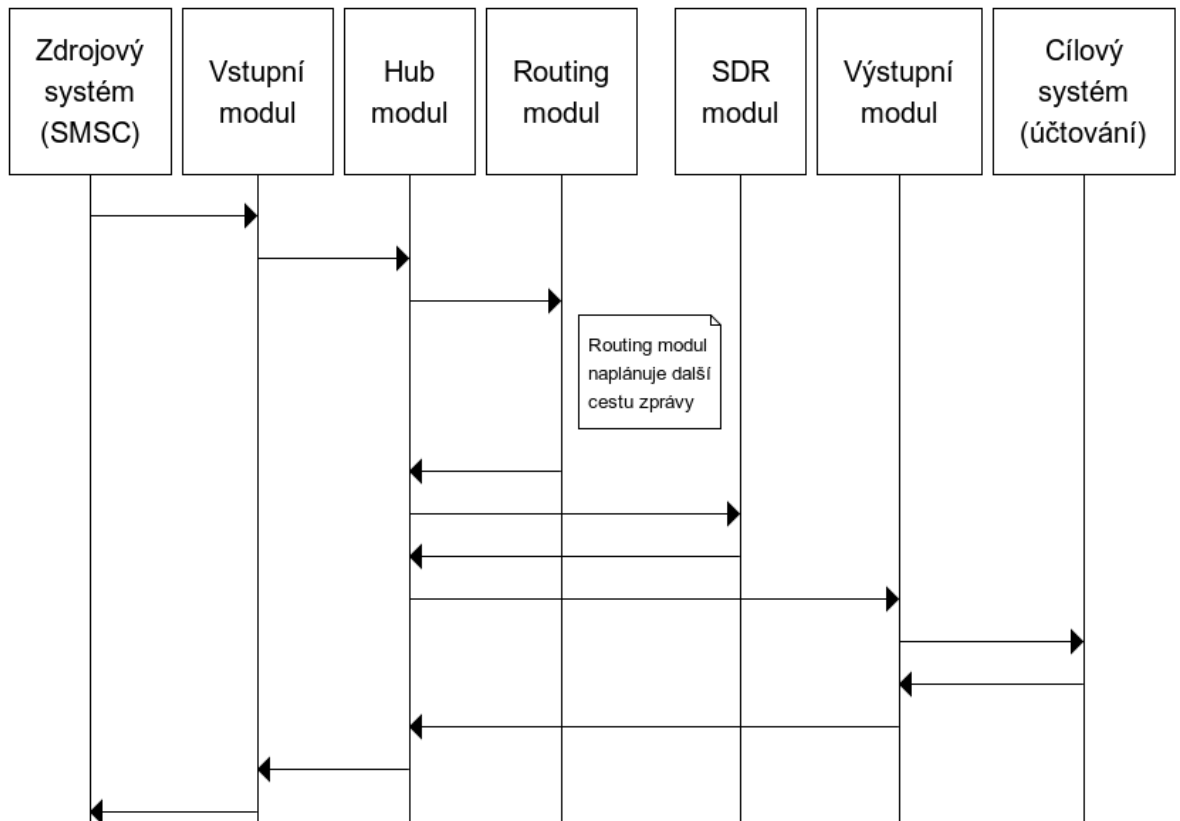
K obslužení různých protokolů slouží moduly, které se dají rozdělit do tří základních kategorií – vstupní, výstupní a speciální moduly.



Obr. 8. Architektura UGC.

Vstupní moduly se zpravidla chovají jako server, naslouchají na konfigurovaných portech, zpracovávají příchozí spojení a předávají data dále dovnitř UGC. Výstupní moduly obvykle pracují jako klient, navazují spojení na server a data přijatá zevnitř UGC zasílají serveru. Do kategorie speciálních modulů lze zařadit moduly vnitřní logiky UGC a moduly, které nelze zařadit jako vstupní či výstupní. Některé z nich mohou být schopné zároveň odesílat a přijímat zprávy. Mezi tyto moduly patří například:

- hub modul (rozbočovač) – propojuje vstupní a výstupní moduly a poté předává zpět původním odesílatelům výsledek zpracování zprávy, viz obrázek (Obr. 9).
- routing modul (směrovač) – na základě definovaných pravidel volí další cestu každé přijaté interní zprávy.
- SDR modul – provádí účtování všech definovaných operací do textových souborů, zvaných SDR (někdy CDR).
- významným typem modulu je modul plugin, jež umožňuje do modulární konstrukce UGC připojit libovolné rozšíření, které musí poskytovat definované rozhraní. Tyto pluginy lze psát pouze v jazyce C++.



Obr. 9. Ukázka průběhu zprávy přes UGC.

UGC lze upravovat pro specifické účely mnoha způsoby, např. je možné jej nakonfigurovat tak, že samo sebe testuje generovanými zprávami, nebo je možné vůbec nepoužít routing modul. Překlad parametrů zpráv vstupních protokolů do interní zprávy a stejně tak překlad interní zprávy do výstupních protokolů je parametrizovatelný, data je možné volně upravovat volně přesouvat mezi parametry dle požadované specifikace – například je možné oříznout text zprávy, vyměnit telefonní čísla odesilatele a příjemce. Složitější úpravy se dají dělat s pomocí vestavěného skriptovacího jazyka. Za předpokladu poskytnutí jednoduchého rozhraní, shodného s tím pro modul plugin, je možné upravit či rozšířit chování všech modulů. Každé řešení má své výhody. Rozšíření ve skriptovacím jazyce lze průběžně upravovat a kritická chyba v jejich těle způsobí pouze výjimku v prováděcím vlákne, zatímco moduly plugin napsané v jazyce C++ poskytují vyšší výkon, ale po každé změně je nutné je překompilovat a jejich selhání může způsobit pád celého UGC.

Velké možnosti úprav se používají ve formě solution packů – balíčků konfiguračních, skriptových souborů a sdílených knihoven, které je možné aplikovat na UGC komponentu

ve výchozím stavu a u každého zákazníka tak dosáhnout jeho požadované specifické funkcionality.

7 LINK

7.1 Obecné informace

Protokol LINK [25] je odlehčený protokol, který slouží k propojení více instancí UGC. Umožňuje efektivní komunikaci mezi jednotlivými instancemi pomocí interní GBG zprávy (univerzální struktura, kterou se uvnitř UGC reprezentuje zpráva, nezávisle na vstupním či výstupním protokolu). Komunikace probíhá přes TCP/IP protokol, je asynchronní a probíhá dle schématu klient-server. Klient zahajuje spojení se serverem a zasílá mu požadavky, server téměř vždy jen odpovídá. Asynchronnost komunikace je na úrovni požadavků, tj. odpověď na určitý požadavek nemusí přijít hned, ale i po zaslání několika dalších požadavků.

Protokol LINK je zamýšlený pro použití na zabezpečené vnitřní síti v rámci důvěryhodného prostředí. Vzhledem k tomu není využíváno šifrování, autentizace ani autorizace. Veškeré ošetření chyb přenosu využívá mechanismů poskytovaných TCP/IP protokolem. Výchozí komunikační port není stanoven, doporučeno je využívat porty 13000 a následující.

Spojení je navazováno dvoucestným handshakem, ukončováno trojcestným. Udržováno je pomocí keep-alive paketů. V případě zahlcení se může server dočasně označit jako zaneprázdněný, aby měl prostor zpracovat nahromaděné požadavky a uvolnit zdroje pro přijímání dalších nových požadavků. Spojení může být ze strany serveru ukončeno na základě vypršení několika různých časových prodlev:

- neaktivita klienta – klient neposlal Keep-alive paket
- pozdní navázání spojení – klient neposlal Bind paket
- pozdní odpověď na uvolňovací požadavek – klient neodpověděl na Release paket

Popisy významu jednotlivých paketů jsou uvedeny dále v kapitole 7.3. Spojení může být ukončeno i ze strany klienta:

- neaktivita serveru – server včas neodpověděl na Keep-alive paket
- pozdní navázání spojení – server včas neodpověděl na Bind paket
- pozdní odpověď na datový požadavek – server včas neodpověděl na Data paket

- pozdní odpověď na uvolňovací požadavek – server včas neodpověděl na Release paket

Požadavky i odpovědi na ně mohou nést parametry zprávy, reprezentované číselným ID, typem a hodnotou. Množina číselných ID se bijektivně zobrazuje do množiny textových řetězců, každé ID má tedy svůj vlastní uživatelsky čitelný název či popis.

7.2 Struktura

Protokol je binární, základní element je bajt. Data jsou přenášena se síťovou endiánitou – ve vícebajtové hodnotě je bajt s nejvýznamnějším bitem odesíláný jako první. Textové informace jsou kódovány v UTF-8.

Každé PDU (Protocol Data Unit) protokolu LINK má strukturu popsanou v tabulce (Tab. 1) – níže uvedená čísla značí bity.

31	24 23	16 15	8 7	0
MAGIC	OPCODE	DATA-SIZE		
...data...				

Tab. 1. Základní struktura paketu protokolu LINK.

Prvních 32 bitů tvoří hlavičku. MAGIC je signatura – identifikační hodnota, podle které se dá rychle rozlišit paket protokolu LINK od jiného. Její hodnota je definovaná jako 0xAB. OPCODE je kód operace, který určuje zaměření paketu. Jednotlivé kódy operace a jejich význam jsou vysvětleny níže. DATA-SIZE určuje velikost datové části paketu.

Přesahuje-li velikost datové části paketu 65 535 bajtů, je použita rozšířená hlavička. DATA-SIZE je nastavená na hodnotu 0xFFFF a po ní následuje 32bitová hodnota, která uvádí skutečnou velikost datové části. Teprve po ní jsou zasílána data paketu.

7.3 Operační kódy

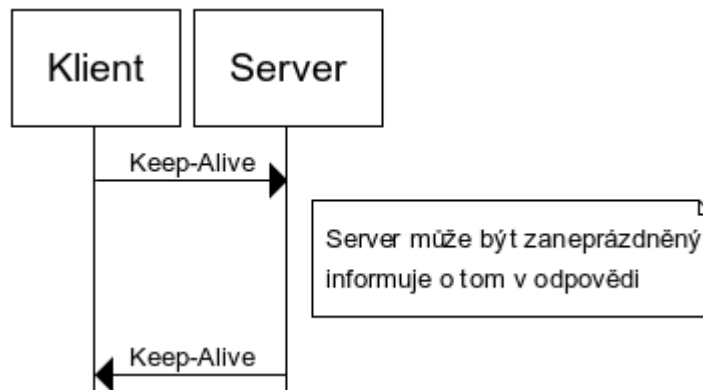
Protokol LINK podporuje kódy operace popsané v tabulce (Tab. 2) – opkód je zkratka pro „operační kód“.

Opkód	Název	Zasílá	
		Klient	Server
0x00	Keep-Alive	X	X
0x01	Bind Request	X	
0x02	Bind Response		X
0x03	Release Request	X	X
0x04	Release Response	X	X
0x05	Close Request	X	X
0x06	Names Update	X	X
0x07	Data Request	X	
0x08	Data Response		X

Tab. 2. Seznam operačních kódů protokolu LINK.

7.3.1 Keep-Alive

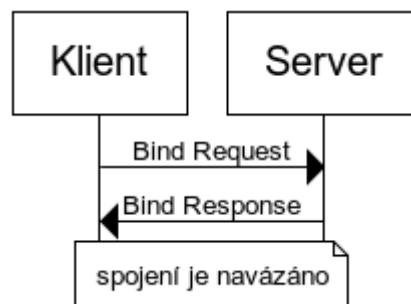
Tento paket můžou zasílat obě strany, viz obrázek (Obr. 10). Server jej zasílá jako odpověď na klientův paket, naopak klient na Keep-Alive pakety ze serveru neodpovídá – jinak by došlo k zacyklení. V rámci odpovědi na Keep-Alive paket server zasílá informaci o tom, zda je zaneprázdněný, přičemž klient by v takovém případě měl přestat zasílat další Data Request pakety, dokud server v některé z dalších odpovědí na Keep-Alive paket neuvede jinak.



Obr. 10. Průběh komunikace paketem Keep-Alive.

7.3.2 Bind Request/Response

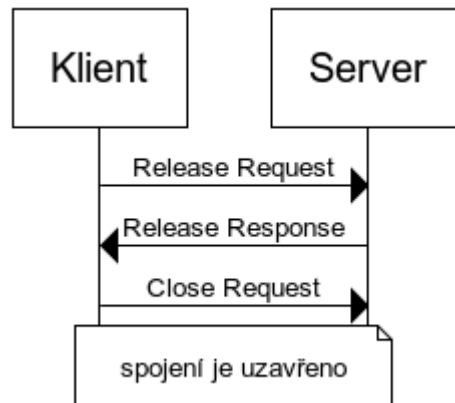
Klient zasílá paket Bind Request, na který pak server odpovídá paketem Bind Response, viz obrázek (Obr. 11). Jakmile je tato výměna dokončena, je ustaveno spojení. V požadavku i v odpovědi jeho zasilatel uvádí svou verzi protokolu LINK a svou identifikaci. Server navíc ve své odpovědi zasílá i výsledek operace. Pokud je výsledek negativní, LINK spojení není ustaveno a TCP/IP spojení je uzavřeno.



Obr. 11. Průběh navázání spojení.

7.3.3 Release Request/Response, Close Request

Klient i server mohou zahájit ukončovací proceduru pomocí paketu Release Request, viz obrázek (Obr. 12). Druhá strana poté odpovídá paketem Release Response, jakmile zpracuje všechny nedokončené požadavky. Iniciátor ukončovací procedury po obdržení odpovědi zasílá paket Close Request, spojení je poté ukončeno.



Obr. 12. Průběh ukončení spojení.

7.3.4 Names Update

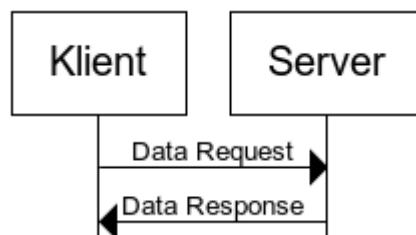
Tento paket můžou zasílat obě strany kdykoliv od navázání spojení, neposílá se na něj žádná odpověď. V paketu jsou zaslány dvojice ID-název, které pomáhají v uživatelské identifikaci parametrů.

7.3.5 Data Request/Response

Tento paket tvoří hlavní náplň práce LINK protokolu, jelikož si přes něj jednotlivé instance UGC zasílají data, viz obrázek (Obr. 13). Každý požadavek symbolizuje jednu interní GBG zprávu. Základem Data paketů je 64bitové transakční ID, podle kterého se párují požadavky s odpověďmi. Toto ID musí být v rámci spojení pro každý požadavek unikátní. Následuje posloupnost parametrů, každý z nich je ve formátu ID – typ – hodnota.

V odpovědi se zasílá totéž transakční ID, výsledek operace a v případně úspěšného výsledku lze zaslat i posloupnost parametrů.

Jsou podporovány 8-, 16-, 32- a 64bitové znaménkové a bezznaménkové číselné datové typy, 8bitový logický booleovský datový typ, ASCII textový řetězec, binární řetězec a UTF-8 textový řetězec. Dále je možné pracovat se strukturami a sekvencemi, které jsou kontejnery pro předchozí datové typy. Struktura je kontejner pojmenovaných parametrů, zatímco sekvence obsahuje nepojmenované parametry (jejich ID jsou 0).



Obr. 13. Průběh zasílání dat.

7.4 Výsledky operací

Protokol zahrnuje výsledkové kódy s jednotným číslováním pro všechny operace:

0	Neznámá chyba
1	Navázáno spojení
2	Chyba navazování spojení – nesedí verze
3	Chyba navazování spojení – neplatná data
4	Úspěšně zpracováno
5	Zpracovávání selhalo – neplatná data
6	Zpracovávání selhalo – nastala interní chyba
7	Zpracovávání selhalo – zahlceno
8	Zpracovávání selhalo – server není připravený

Tab. 3. Seznam dostupných výsledkových kódů.

8 TTT

Společnost Acision pro testování produktů vyvinula a využívá nástroj TTT (Tel Test Tool). Jak již napovídá název, TTT jsou napsána v jazyce Tcl [26].

8.1 Simulátory

Jednou ze zásadních schopností TTT je schopnost vytvořit kompletní simulaci zákaznického systému kolem testovaného produktu. Díky tomuto lze pro produkt obstarat simulaci všech síťových spojení a testovat všechny případy užití. TTT zahrnují simulátory následujících entit a protokolů:

- GSM

GSM [27] je protokol vytvořený pro hlasovou a textovou komunikaci. GSM protokol je spolu s CDMA celosvětově podporovaný mobilními telefony.

- SMPP aplikace

Aplikace v SMPP světě vykonávají podobnou činnost jako mobilní telefony, avšak jsou používány se záměrem automatizace a potenciálně mnohem větším pohybem zpráv. SMPP aplikace se používají například pro odesílání autorizačních zpráv z bank, reklamních sdělení obchodů či pro hlasování v televizních pořadech. SMPP protokol [28] byl vyvinut ve firmě Aldiscon, předchůdci firmy Acision.

- SMPP brány

Tyto brány sdílí mnoho společných prvků s SMPP aplikacemi, zpravidla je brána jen jeden bod, který může abstrahovat větší množství aplikací za ním ukrytých.

- HLR (Home Location Register)

HLR [27] je centrální databáze u mobilního operátora, která obsahuje informace o všech uživateli mobilního telefonu. Ukládá se především jeho IMSI (unikátní identifikátor SIM karty), ale také telefonní číslo MSISDN, aktuální poloha či odebírané služby.

- Diameter

Diameter je „AAA protokol (autentizace, autorizace a účtování) používaný pro přístup k síti nebo pro IP mobilitu“ [29]. Komunikuje se v něm pomocí AVP (dvojice parametr-hodnota). Součástí protokolu je schéma vyjednávání schopností (capabilities negotiation), hlášení o chybách, rozšiřitelnost vlastními AVP.

- LDAP

LDAP [30] je otevřený aplikační protokol pro ukládání a správu dat na adresářovém serveru. Používá se pro práci se standardizovanými daty, která nevyžadují příliš časté aktualizace, například pro informace o uživateli.

- účetní aplikace

O každé zprávě, která projde zákaznickým systémem, je třeba vytvořit záznam – zaúčtovat ji. Na konci zúčtovacího období se na základě těchto záznamů vytváří faktury pro uživatele zákaznickova systému. Tyto záznamy jsou obvykle nazývány CDR nebo SDR [31] (Call/Service Data Record). Zákaznické systémy v ICT (informační a telekomunikační technologie) světě jsou na těchto záznamech tedy pochopitelně do velké míry závislé, díky nim mohou navíc analyzovat tok zpráv ve své síti.

- předplacená brána

Tak jako účetní aplikace vedou záznamy o paušálních zákaznících, tyto brány udržují stav předplacených zákazníků. Podstatným rozdílem v jejich funkci je, že předplacená brána rovnou vykoná pokus o odečtení kreditu předplaceného zákazníka. Ať již pokus skončí jakkoliv, záznam o výsledku je předán účetní aplikaci.

V rámci této práce je seznam rozšiřován o protokol LINK. Více podrobností o jeho simulátoru je popsáno v kapitole 9.

8.2 Testovací scénáře

Test je v pojetí TTT vnímán obdobně jako v jiných testovacích nástrojích. V obecném pohledu je to ověření jednoho aspektu testovaného produktu. Ať už jde o poslání SMS zprávy z jednoho mobilního telefonu na druhý nebo ověření vytvoření účetního záznamu při neúspěšném zaslání zprávy, vždy je třeba do produktu zadat kontrolovaný vstup. Poté je nutné ověřit následný výstup dat, SNMP údajů a ladicích informací co nejvíce dostupnými způsoby. Každý scénář je samostatná entita, která není přímo závislá na průběhu jiného scénáře.

TTT dovolují logické závislosti (viz dále), v případě množiny více testovacích scénářů se stejnými závislostmi ale na pořadí spouštěných scénářů nezáleží. Každý scénář na svém počátku automaticky ověří, že všechny procesy produktu, které jsou nutné, jsou stále

funkční. Stejně tak po skončení scénáře se ověří, zda produkt zůstal funkční, a zkontroluje se přítomnost případných chyb v ladicích informacích.

8.3 Konfigurační scénáře

Testované produkty společnosti Acision mají množství různých funkcí, které lze nezávisle na sobě konfigurovat. Zákazník nemusí použít všechny, ale zároveň jsou připraveny podmínky pro to, aby se v případě potřeby dalo využít maximum nabízené funkcionality. Produkt UGC podporuje různé protokoly a funkcionality pomocí svých vstupních, výstupních a speciálních modulů, díky čemuž vzniká poměrně velký prostor možných testovacích scénářů. Aby bylo možné udržovat scénáře ve strukturované formě, současně se dala udržet jejich nezávislost a testy nebyly příliš složité svou náplní ani údržbou, existují specifické scénáře, které přímo neověřují žádnou funkcionality – testovaný produkt pouze konfiguruje. Každý takový specifický konfigurační scénář po proběhnutí nastavení zkontroluje, že po jeho aplikaci produkt zůstává ve funkčním stavu – mimo záměrně vadné konfigurace, které ověřují pravý opak. Ke každému konfiguračnímu scénáři existuje jeho protiklad, který konfiguraci produktu vrátí do původního stavu na základě automaticky sledované množiny provedených změn.

8.4 Plánování

TTT podporují rozmanité možnosti výběru spouštěných testovacích scénářů. Při každém spuštění nástroje jsou načteny informace o všech dostupných scénářích a poté je vybíráno z celkové množiny scénářů na základě uživatelské volby. TTT plánování dovoluje zařazovat či vynechávat specifické funkcionality produktu, které jsou přirovnatelné ke štítkům (skupinám scénářů zabývajících se například účtováním, SMPP zprávami či užitím určitého protokolu), ale zároveň je možné vybírat v rámci jednotlivých testovacích scénářů, ať už jejich výčtem či obecněji regulárním výrazem. Každý testovací scénář může mít závislost na jednom či více jiných. Toho se využívá pro urychlení regresního testování, kdy komplikovanější scénáře mají logickou závislost na jednodušším scénáři a v případě jeho selhání se vůbec nevykonají a rovnou se přeskočí.

Každý z testovacích scénářů dále může mít libovolné množství závislostí na výše popsaných konfiguračních scénářích. Rovněž konfigurační scénáře mohou mít závislosti na jiných konfiguračních scénářích, ale nemohou mít závislosti na testovacích scénářích.

Plánování je ošetřeno proti selhání způsobenému cyklickou závislostí. Díky rozdělení na sekvence testovacích scénářů a konfigurační scénáře lze vytvořit přehlednou testovací hierarchii. Plán průběhu testovací sady lze rozdělit dle počtu testovaných scénářů či dle doby jejich běhu na rovnoměrné části, které lze spustit na více strojích zároveň.

8.5 Výstup

Stejně jako je podstatná spolehlivá funkčnost testovaného produktu, je důležité věnovat pozornost i výstupům, ať už z produktu či samotné testovací sady. Pro tento účel jsou TTT vybavena několika různými úrovněmi ladicích výpisů, které si tester může při spuštění nastavit. Každý testovací i konfigurační scénář je rozdělen na jednotlivé kroky, každý z těchto kroků je testerem dokumentován. V případě selhání testu je usnadněno vyhledání místa, ve kterém došlo k chybě, a pro potřeby dalších informací je možné zvýšit úroveň výpisu ladicích informací. V TTT je zavedena i podpora pro ladění testovaných produktů – probíhající testovací scénáře lze pozastavit, zvýšit úroveň výpisu ladicích informací produktu a poté pokračovat ve scénářích. Poté, co je získáno dostatečné množství informací, lze úroveň výpisu opět snížit, aniž by bylo třeba ukončit a znovu spustit celou testovací sadu.

Tester může nastavit, aby se při průběhu každého testovacího scénáře zachytával síťový provoz pomocí aplikace tcpdump. Lze si také vyžádat konfiguraci produktu při selhání testu, která se spolu s identifikací právě selhavšího testovacího scénáře uloží pro pozdější studium. Po proběhnutí testovací sady je vypsáno shrnutí, ve kterém se testy rozdělí na úspěšné, neúspěšné a přeskočené. Na stroji, který spouštěl testovací sadu, zůstanou uložená veškerá data z jejího průběhu. Z nich je možné vygenerovat dynamický report, ve kterém jsou přehledně dostupná účetní data, uložené konfigurace produktu, zachycený tcpdump a kompletní plán testovacích scénářů spolu s možností prohlížet záznamy o jejich průběhu. V případě, že testovací sada byla naplánována na více oddělených částí, všechny dílčí záznamy se na konci testovacího běhu spojí do jednoho.

8.6 Automatizace

Aby bylo možné vyvíjet testovací a konfigurační scénáře v požadované kvalitě s co nejnižší režií, byly vytvořeny společné pomocné funkce. Mezi tyto funkce patří například odeslání příkazu do vzdálené příkazové řádky na testovaném stroji, zastavení a spuštění

skupiny procesů, automatické vrácení konfigurace do dříve uloženého stavu či ověření hodnoty SNMP čítačů. Nejčastěji použitelné funkce jsou popsány na zvláštní webové stránce.

Všechny scénáře jsou pečlivě dokumentované. Jednotlivé kroky jejich činnosti jsou dostatečně popsány a každý scénář obsahuje hlavičku se jménem autora, shrnutím náplně scénáře a případnými poznámkami. Z těchto hlaviček se pomocí nástroje Doxygen² vygeneruje dynamická online dokumentace.

Každý scénář je psán tak, aby byl co nejvíce robustní. V případě kritického selhání, například hrozícího zaplnění disku, jsou všechny zbývající testovací a konfigurační scénáře přeskočeny a testovací sada se nejrychlejším korektním způsobem ukončí. Díky tomuto opatření je možné rychle ukončit testování bez ztráty stability prostředí testovaného produktu, společně s možností vygenerovat závěrečný report a zároveň stav testovacího stroje dále nezhoršit.

² K dispozici na oficiálních stránkách <http://www.stack.nl/~dimitri/doxygen/>

9 LINK SIMULÁTOR PRO TTT

Pro TTT bylo potřeba vytvořit simulátor protokolu LINK. Vytvořil jsem tedy funkce pro každý krok průběhu protokolu (pro více informací o operacích protokolu viz kapitolu 7).

Nejprve jsem vytvořil klientskou část, kterou jsem odladil přímo v komunikaci s implementací protokolu v produktu UGC. Poté jsem vytvořil serverovou část, kterou jsem odladil v komunikaci se simulovaným klientem. Jakmile byl simulátor dostatečně stabilní, začal jsem vyhledávat duplicitu v kódu a od těch největších vytvářel pomocné funkce, díky kterým je kód přehlednější a obsahuje méně chyb.

Simulátor je přiložen v datovém archivu, resp. na CD. Ve zdrojovém souboru CL-LINK.tcl jsou funkce potřebné pro klientskou i serverovou stranu. Následuje seznam a stručný popis vytvořeného API. Pokud název funkce obsahuje samostatné písmenko *i* (např. `gbglink_i_setup`), znamená to, že je funkce navržena jako interní; není ale žádným způsobem zakázána. Uživatel ji samozřejmě může používat, pokud to vyžaduje specifický scénář či pokud chce testovat všechny kroky ručně.

9.1 Obecné API

Toto API je nezávislé na klientské či serverové části, mohou jej používat obě strany.

9.1.1 `gbglink_i_setup`

Nastaví globální proměnné využívané simulátorem. Funkce nemá vstupní parametr, v případě úspěchu vrací 0, v případě selhání jinou celočíselnou hodnotu. Je nastavováno počáteční transakční ID, textový popis jednotlivých operačních kódů a signatura paketu. Kromě těchto hodnot jsou navíc nastavovány i časové prodlevy v situacích stanovených protokolem.

9.1.2 `gbglink_release`

Provede korektní ukončení spojení LINK. Přijímá jediný parametr `gbglinkid` – handle na dané LINK spojení. V případě úspěchu vrací 0, v případě selhání jinou celočíselnou hodnotu.

Odešle Release Request paket, počká na paket Release Response, poté pošle paket Close Request a ukončí spojení. Funkci může zavolat klient i server.

9.1.3 `gbglink_i_send`

Odešle LINK paket druhé straně spojení. Přijímá dva parametry – *gbglinkid* a *hexdata* – tj. handle na dané LINK spojení, resp. hexadecimální řetězec s obsahem daného paketu. Vrací vždy 0.

9.1.4 `gbglink_rcv`

Přijme LINK paket od druhé strany spojení. Přijímá tři parametry – *gbglinkid*, *outcode* a *outdata* – tj. handle na dané LINK spojení, výstupní proměnnou s hodnotou operačního kódu a výstupní proměnnou s hodnotou velikosti datové části. V případě úspěchu vrací 0, v případě selhání jinou celočíselnou hodnotu. *Outcode* a *outdata* jsou nastavené na správnou hodnotu v případě korektního paketu, v opačném případě jsou nastaveny na -1.

Funkce abstrahuje další interní volání. Uvnitř se používají pomocné funkce `gbglink_i_decode_header_opcode` a `gbglink_i_decode_header_datasize`.

9.1.5 `gbglink_send_namesupdate`

Odešle paket Names Update druhé straně. Přijímá dva parametry – *gbglinkid* a *names* – tj. handle na dané LINK spojení a seznam odeslaných dvojic ID-název. V případě úspěchu vrací 0, v případě selhání jinou celočíselnou hodnotu.

9.1.6 `gbglink_receive_namesupdate`

Přijme paket Names Update od druhé strany. Přijímá dva parametry – *gbglinkid* a *names* – tj. handle na dané LINK spojení a volitelnou výstupní proměnnou, do které se uloží přijaté dvojice ID-název. V případě úspěchu vrací 0, v případě selhání jinou celočíselnou hodnotu.

9.1.7 `gbglink_dump_packet`

Dekóduje celý LINK paket a vypíše o něm dostupné informace. Přijímá jediný parametr *hexdata* – hexadecimální řetězec s LINK paketem. V případě úspěchu vrací 0, v případě selhání jinou celočíselnou hodnotu.

Funkce je vhodná pro ladící účely, ať už v rámci testování či vývoje dalších API.

9.2 Klientské API

Toto API dává smysl jen pro klientskou stranu protokolu LINK.

9.2.1 `gbglink_open`

Vytváří TCP spojení k serveru LINK. Přijímá volitelné parametry *port*, *server* a *lport*. Parametry *port* a *server* jsou adresa a port vzdáleného serveru. Parametr *lport* lze využít pro volbu, ze kterého portu se Tcl bude ke vzdálenému serveru připojovat, v opačném případě si Tcl zvolí port automaticky. Funkce vrací handle na vytvořené LINK spojení.

Funkce zavolá `gbglink_i_setup`, aby pro dané připojení nastavil časové prodlevy. Poté se na základě zadaných parametrů utvoří TCP spojení a pomocí Tcl funkce `fconfigure` se vypne buffering, nastaví se binární kódování a spojení se nastaví jako neblokující.

9.2.2 `gbglink_close`

Uzavře klientské spojení k serveru LINK. Přijímá jediný parametr *gbglinkid* – handle na dané LINK spojení. Vrací vždy 0.

9.2.3 `gbglink_client_bind`

V rámci vytvořeného TCP spojení ustaví spojení nad LINK protokolem – zašle paket Bind Request, počká na paket Bind Response. Přijímá jediný parametr *gbglinkid* – handle na dané LINK spojení. V případě úspěchu vrací 0, v případě selhání jinou celočíselnou hodnotu.

Funkce abstrahuje další interní volání, kde se kontroluje, že od serveru přišel korektní paket, který obsahuje správný výsledkový kód a shodnou verzi protokolu.

9.2.4 `gbglink_client_keealive`

Pošle serveru paket Keep-alive, počká na odpověď stejným paketem. Přijímá jediný parametr *gbglinkid* – handle na dané LINK spojení. V případě úspěchu vrací 0, v případě selhání jinou celočíselnou hodnotu.

9.2.5 `gbglink_client_data`

Pošle serveru paket Data Request, počká na paket Data Response. Přijímá dva parametry – *gbglinkid* a *params_list* – tj. handle na dané LINK spojení, resp. seznam trojic ID-typ-parametr. V případě úspěchu vrací 0, v případě selhání jinou celočíselnou hodnotu.

Funkce abstrahuje další interní volání. V nich se kontroluje platnost zadaných trojic ID-*typ-parametr*, sestavuje výsledný paket pro odeslání v požadavku a zároveň se ověřuje transakční ID, výsledkový kód a případný seznam trojic ID-*typ-parametr* v odpovědi.

9.3 Serverové API

Toto API dává smysl jen pro serverovou stranu protokolu LINK.

9.3.1 *gbglink_listen*

Serverový ekvivalent klientského *gbglink_open*, vytváří server LINK a přijímá TCP spojení klienta. Přijímá volitelné parametry *port* a *tout*. *Port* je lokální port, který bude očekávat příchozí spojení, *tout* je časová prodleva, během které se očekává připojení klienta. Funkce vrací handle na vytvořené LINK spojení.

Funkce zavolá *gbglink_i_setup*, aby pro dané připojení nastavil časové prodlevy. Poté se na základě zadaných parametrů otevře naslouchající TCP spojení. Jakmile se klient připojí, pomocí Tcl funkce *fconfigure* se vypne využití vyrovnávací paměti, nastaví se binární kódování a spojení se nastaví jako neblokující. Pokud se během časové prodlevy nepřipojí žádný klient, naslouchající spojení se uzavře.

9.3.2 *gbglink_unlisten*

Serverový ekvivalent klientského *gbglink_close*, server LINK ukončuje TCP spojení ke klientovi. Funkce přijímá jediný parametr *gbglinkid* – handle na dané LINK spojení.

9.3.3 *gbglink_server_bind*

V rámci vytvořeného TCP spojení ustaví spojení nad LINK protokolem – počká na paket Bind Request, poté zašle paket Bind Response. Přijímá jediný parametr *gbglinkid* – handle na dané LINK spojení. V případě úspěchu vrací 0, v případě selhání jinou celočíselnou hodnotu.

Funkce abstrahuje další interní volání, kde se kontroluje, zda od klienta přišel korektní paket, který obsahuje shodnou verzi protokolu.

9.3.4 `gbglink_server_keepalive`

Počká na paket Keep-alive, pošle klientovi stejný paket. Přijímá dva parametry – *gbglinkid* a *busy* – tj. handle na dané LINK spojení a přepínač, zda klientovi hlásit zaneprázdnění. V případě úspěchu vrátí 0, v případě selhání jinou celočíselnou hodnotu.

Ověřuje, že klient v rámci datové části paketu posílá pouze nulové bajty.

9.3.5 `gbglink_server_data`

Počká na Data Request, pošle klientovi paket Data Response. Přijímá tři parametry – *gbglinkid*, *rescode* a *params_list* – tj. handle na dané LINK spojení, výsledkový kód, který má být zaslán klientovi, a vstupně-výstupní proměnnou přijatých trojic ID-typ-parametr. V případě úspěchu vrátí 0, v případě selhání jinou celočíselnou hodnotu.

Funkce abstrahuje další interní volání. V nich se kontroluje platnost přijatých trojic ID-typ-parametr a zároveň se sestavuje výsledný paket s výsledkovým kódem a případným seznamem trojic ID-typ-parametr v odpovědi.

9.4 Interní a pomocné API

Toto API je navrženo spíše pro interní využití než pro přímé použití uživatelem. Ve zdrojovém kódu existují i další interní funkce, které jsou vždy součástí některé z těch popsaných v této kapitole.

9.4.1 `gbglink_i_verify_opcode`

Ověřuje správnou hodnotu operačního kódu. Vždy proti rozsahu danému protokolem a volitelně navíc i proti jiné referenční hodnotě. Přijímá dva parametry – *opcode* a *expect* – tj. ověřovaný operační kód a volitelně referenční hodnotu, se kterou je porovnán. Vrací 0 v případě úspěchu, jinou celočíselnou hodnotu v případě selhání.

9.4.2 `gbglink_i_encode_header`

Obaluje specifikovaná data LINK hlavičkou. Přijímá dva parametry – *hexdata* a *op* – tj. hexadecimální řetězec s datovou částí budoucího paketu a zamýšlený operační kód. Vrací výsledný hexadecimální řetězec vytvořené LINK hlavičky.

9.4.3 `gbglink_i_decode_header_opcode`

Dekóduje hlavičku LINK paketu a vyčte z ní operační kód. Přijímá jeden parametr *hexhdr* – hexadecimální řetězec s hlavičkou paketu. Je-li paket korektní, vrací hodnotu operačního kódu, jinak vrací -1.

Funkce abstrahuje interní volání `gbglink_i_verify_opcode`.

9.4.4 `gbglink_i_decode_header_datasize`

Dekóduje hlavičku LINK paketu a vyčte z ní velikost datové části. Přijímá jeden parametr *hexhdr* – hexadecimální řetězec s hlavičkou paketu. Je-li paket korektní, vrací velikost datové části, jinak vrací -1.

9.4.5 `gbglink_i_decode_header`

Dekóduje hlavičku z LINK paketu, vypíše signaturu paketu, operační kód a velikost datové části. Přijímá jeden parametr *hexhdr* – hexadecimální řetězec s hlavičkou paketu, případně s celým LINK paketem. Vrací hexadecimální řetězec hlavičky v případě úspěchu, prázdný řetězec v případě selhání.

Funkce interně využívá volání `gbglink_i_verify_opcode`.

9.5 Testovací API

Pro testování API simulátoru jsem vytvořil několik jednotkových testů. Jejich současný počet je kolem 100.

9.5.1 `cl_gbglink_apitest`

Spouští dostupné jednotkové testy pro LINK protocol. Uvnitř jsou spouštěny tři funkce:

- `cl_gbglink_apitest_001` – testuje funkce `gbglink_i_verify_opcode` a `gbglink_i_verify_rescode`
- `cl_gbglink_apitest_002` – testuje funkce `gbglink_i_decode_packet_header`, `gbglink_i_decode_header_opcode` a `gbglink_i_decode_header_datasize`
- `cl_gbglink_apitest_003` – testuje funkci `gbglink_i_encode_header`

9.6 Ukázka použití v praxi

Pro použití simulátoru stačí terminálové okno a tclsh – interpret jazyka Tcl. Následuje ukázka scénáře, ve kterém jsou využity všechny dostupné operační kódy protokolu LINK. Jednotlivé kroky posloupnosti jsou rozděleny mezi klienta a server, každá logická množina kroků je zvlášť okomentována. Některé funkce přijímají volitelné parametry, které nemusejí být v ukázce využity.

Klient	Server
<ul style="list-style-type: none"> Na obou stranách je třeba spustit interpret Tcl a načíst do něj všechny funkce dostupné v souboru CL-LINK.tcl. 	
tclsh	tclsh
source CL-LINK.tcl	source CL-LINK.tcl
<ul style="list-style-type: none"> Server začne pomocí gbglink_listen naslouchat na portu 13000, naslouchá v tomto případě dle druhého parametru 100 vteřin, během očekává připojení klienta pomocí gbglink_open. Poté probíhá připojovací handshake. V této ukázce běží klient i server na stejném stroji – adresa je localhost. 	
	set gbglinkid [gbglink_listen 13000 100]
set gbglinkid [gbglink_open 13000 localhost]	
gbglink_i_client_send_bind_req \$gbglinkid	
	gbglink_i_server_get_bind_req \$gbglinkid
	gbglink_i_server_send_bind_rsp \$gbglinkid
gbglink_i_client_get_bind_rsp \$gbglinkid	
<ul style="list-style-type: none"> Klient zašle paket Names Update, ve kterém zasílá seznam dvojic ID-název. Server může odpovědět podobným paketem, ale nemusí. 	
gbglink_i_both_send_namesupdate \$gbglinkid [list 0x1 Cislo 0x2 Struktura 0x3 VelkeCislo 0x4 MaleCislo]	

	gbglink_i_both_get_namesupdate \$gbglinkid
<ul style="list-style-type: none"> Klient zasílá paket Data Request, v něm celočíselnou hodnotu a strukturu obsahující další dvě celočíselné hodnoty. Protokol umožňuje poslat v odpovědi ze serveru další hodnoty nebo prázdnou odpověď. 	
gbglink_i_client_send_data_req \$gbglinkid [list 0x01 int8 64 0x02 struct [list 0x03 int64 123456789 0x04 int16 -256]]	
	gbglink_i_server_get_data_req \$gbglinkid
	gbglink_i_server_send_data_rsp \$gbglinkid
gbglink_i_client_get_data_rsp \$gbglinkid	
<ul style="list-style-type: none"> Ukázka Keep-Alive paketu. Server v tomto případě nevyužívá volitelného parametru pro příznak zaneprázdněnosti – implicitně nastaveno na hodnotu false. 	
gbglink_i_client_send_keepalive_req \$gbglinkid	
	gbglink_i_server_get_keepalive_req \$gbglinkid
	gbglink_i_server_send_keepalive_rsp \$gbglinkid
gbglink_i_client_get_keepalive_rsp \$gbglinkid	
<ul style="list-style-type: none"> Klient zahajuje odpojovací proceduru. Poté jsou uzavřena TCP spojení obou stran. 	
gbglink_i_both_send_release_req \$gbglinkid	
	gbglink_i_both_get_release_req \$gbglinkid
	gbglink_i_both_send_release_rsp \$gbglinkid
gbglink_i_both_get_release_rsp \$gbglinkid	

gbglink_i_both_send_close_req \$gbglinkid	
	gbglink_i_both_get_close_req \$gbglinkid
gbglink_close \$gbglinkid	
	gbglink_unlisten \$gbglinkid
Klient	Server

Tab. 4. Ukázka použití simulátoru protokolu LINK v základním scénáři.

10 DISEKTOR PROTOKOLU LINK PRO WIRESHARK

Součástí cílů práce je vytvoření disektoru protokolu LINK. Nástroj Wireshark dovoluje bez větších nároků vytvoření dodatečné funkcionality ve formě zásuvných modulů, tzv. pluginů. Plugin je v pojetí Wiresharku oddělená a uzavřená jednotka, která komunikuje s hlavní aplikací pomocí jednotného API a v případě vnitřního selhání nezpůsobí pád celé aplikace.

Pro využití možností nabízených nástrojem Wireshark jsem implementoval disektor pro protokol LINK. V implementaci jsem vyšel ze struktury API základních funkcí využívané ostatními disektory ve firmě Acision. V dalších částech této kapitoly je poskytnuta dokumentace.

Zdrojový kód disektoru je přiložen v datovém archivu, resp. na CD. Pro úspěšné sestavení na systémech Linux jsou třeba následující balíky:

- wireshark-dev, libglib2.0, libglib2.0-dev (rodina Ubuntu)
- wireshark-dev, libglib-2.0, libglib-2.0-dev (rodina Debian)
- wireshark-devel, libglib-2.0, libglib-2.0-devel (rodiny Red Hat, Fedora)

Sestavení probíhá pomocí zavolání příkazu make, úspěšně ověřeno bylo na systémech Ubuntu 12.04, Ubuntu 13.10, Debian 7 a Fedora 20.

Celé řešení disektoru sestává ze 4 souborů – packet-ugclink.c, config.c, ugclink.ini a Makefile.

Soubor packet-ugclink.c obsahuje základní a pomocné funkce disektoru popsané dále, které jsou využívány během zpracovávání síťového provozu. INI soubor ugclink.ini v sobě zahrnuje definice podstatných protokolových informací – překlad číselných hodnot operačních kódů, výsledkových kódů a datových typů na smysluplné řetězce, které lze touto cestou snadno upravovat bez nutnosti rekompilace. V souboru config.c se nacházejí funkce nutné ke zpracování informací ze souboru ugclink.ini při načítání pluginu do Wiresharku a mechanismy načítání pluginu z cest, které se mění podle použitého operačního systému. Soubor Makefile v sobě má informace nutné pro úspěšné sestavení projektu.

10.1 Základní funkce

Jádro funkcionality disektoru se dá rozdělit do tří zásadních funkcí – proto_register_UGCLINK, proto_reg_handoff_UGCLINK a dissect_UGCLINK.

10.1.1 proto_register_UGCLINK

Tato funkce slouží k přípravě všech informací potřebných pro korektní pozdější disekci. Jsou načteny informace nutné k dekódování hlavičkových polí a zpracovány struktury s nimi pracující, ze souboru ugclink.ini jsou načteny hodnoty výčtových typů. Funkce dále zaregistruje metadata protokolu, což jsou jeho plný i zkrácený název a zkratka. Disektor je přidán do seznamu dostupných zachytávacích a zobrazovacích filtrů Wiresharku. Nakonec je pomocí funkce proto_reg_handoff_UGCLINK v GUI aplikace vytvořena stránka s uživatelskými nastaveními.

10.1.2 proto_reg_handoff_UGCLINK

Zde se zpracovávají zásahy do uživatelské konfigurace dostupné z Wiresharku. Aktuálně jediná podporovaná dostupná konfigurace je rozsah detekčních portů – tedy portů, ze kterých odeslané nebo přijaté pakety budou považovány za kandidáty na protokol LINK a takto zpracovávány. Porty lze zadat výčtově (1, 2, 3) či rozsahově (1-3), tyto přístupy lze i kombinovat. Výchozí nastavení zpracovávaných portů, které je zakódováno v souboru a importováno do uživatelsky upravitelného nastavení, je „13000, 13001“. Tato funkce je volána při inicializaci pluginu, ale s její pomocí lze měnit rozsahy detekčních portů i později za běhu.

10.1.3 dissect_UGCLINK

Disektor je v tomto bodě nakonfigurován, jsou nastavené porty, jejichž pakety budou disektovány jako náležící protokolu LINK. Uživatel aplikoval filtr „ugclink“, případně vynutil ruční disektování nekandidátského paketu jakožto paketu protokolu LINK. Tím je spuštěna tato funkce nejvyššího významu, která dekóduje co nejvíce informací z daného paketu či paketů a umožní jejich zobrazení uživateli.

Nejprve je ověřená délka paketu – nejmenší validní paket protokolu LINK musí obsahovat alespoň 4 bajty hlavičky, v opačném případě se jedná o poškozený paket nebo paket náležející jinému protokolu. Poté se ověří „magic“ hodnota na začátku paketu – předem

daná konstanta, pomocí které můžeme dostatečně dlouhý paket prohlásit za pravděpodobně náležející očekávanému protokolu. Následně je dekodován kód operace a případná navazující data. V případě některých operačních kódů jsou využity pomocné funkce, které jsou popsány dále. Více podrobností o protokolu LINK lze nalézt v kapitole 8.

10.2 Pomocné funkce

Pro zpřehlednění kódu a snížení jeho duplikace byly vytvořeny pomocné funkce `ugclink_parse_names`, `ugclink_parse_params` a `ugclink_parse_endpoints`.

Všechny tyto funkce přebírají ve svých parametrech odkaz na protokolový strom, na blok paměti obsahující aktuální přijatý paket a na offset, který ukazuje dovnitř tohoto paketu.

10.2.1 `ugclink_parse_endpoints`

Používá se při zpracování požadavků na spojení a jejich odpovědí (kódy operace 0x01 a 0x02). Aktuální offset paketu ukazuje na místo, odkud lze vyčíst informace o přípojném bodu (endpointu). Přípojný bod o sobě sděluje podporovanou verzi protokolu a identifikační řetězec. V řetězci lze zaslat název, IP adresu a port či jiné informace.

10.2.2 `ugclink_parse_names`

Používá se při zpracování zpráv s názvy parametrů (kód operace 0x06). Z paketu lze vyčíst seznam párů 32bitových ID a řetězců s názvem parametru.

10.2.3 `ugclink_parse_params`

Funkce slouží ke zpracování parametrů, které jsou zaslány v datových požadavcích a odpovědích (kódy operace 0x07 a 0x08). Z paketu lze zjistit ID parametru, jeho typ a hodnotu. Je podporováno rekurzivní zpracování struktur a sekvencí, které samy o sobě obsahují další parametry. Hloubka zanoření není v implementaci nijak omezena.

10.3 Ukázka použití v praxi

Pro vyzkoušení disektoru v praxi je třeba provést několik kroků. Nejprve je nutné nainstalovat Wireshark a zkompileovat disektor.

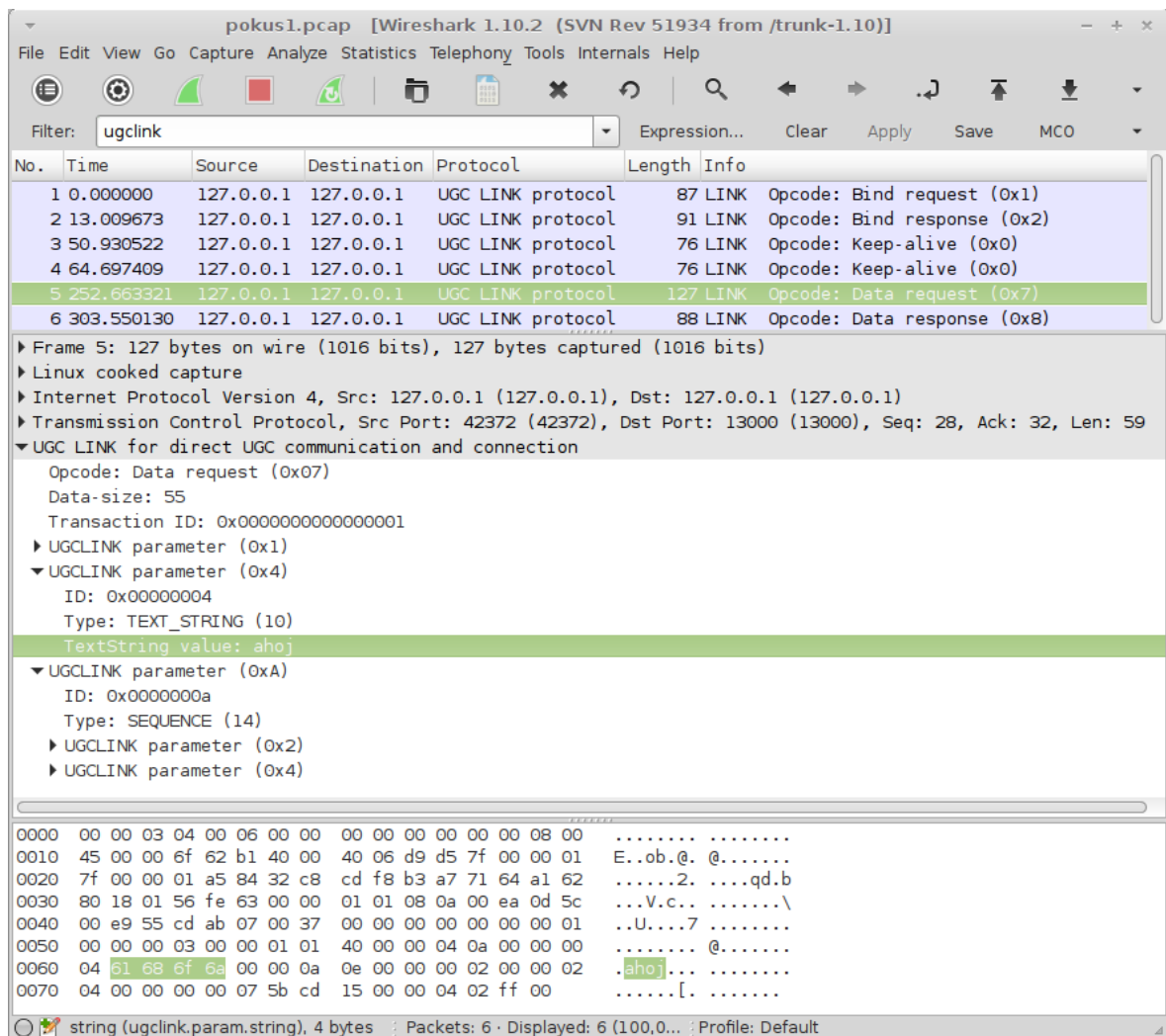
```
cd wireshark-ugclink
```

make

Poté je třeba zachytit několik paketů protokolu LINK v rámci síťového provozu.

tcpdump -i any -w provoz.cap

Vytvořením dvou instancí simulátoru v Tcl interpretu tclsh a využíváním jejich API pak lze provést síťovou komunikaci, která bude zachycena do souboru provoz.cap. Po otevření souboru ve Wiresharku a zadáním filtru „ugclink“ pak bude dekódován a zobrazen tento provoz, viz obrázek (Obr. 14).



Obr. 14. Disektor protokolu LINK zpracoval několik paketů tohoto protokolu.

ZÁVĚR

V rámci práce jsem prozkoumal prostředí vývoje ve firmě Acision, zpracoval základní informace o produktech UGC, TTT a protokolu LINK. Dále jsem vytvořil nástroje na poskytnutí dostatečně vhodného prostředí pro testování komponenty UGC za pomoci protokolu LINK.

Vytvořil jsem simulátor protokolu LINK do testovacího nástroje TTT. Vypracoval jsem klientskou i serverovou část. Obě části jsem samostatně ověřil vůči implementaci protokolu LINK v komponentě UGC, zároveň jsem jejich interoperabilitu ověřil ve vzájemném spojení. Simulátor jsem zběžně výkonově otestoval – byl schopen zpracovat stovky jednoduchých zpráv za sekundu. Simulátor pokrývá téměř celou specifikaci protokolu, chybí zejména kontrola stavu spojení ve vztahu k určitým zprávám – např. Data Request paket je nepřijatelný ve chvíli, kdy ještě neproběhl spojovací handshake. Testování jsem prováděl na platformě Linux, všechny chyby odhalené za pomoci automatických i manuálních testů jsem opravil, všechny automatické testy jsou součástí zdrojového kódu. Dokumentace je uložena v rámci zdrojového kódu a pomocí nástroje Doxygen z ní lze vygenerovat výstup ve formátu HTML.

Dále jsem vytvořil disektor protokolu LINK pro síťový analyzátor Wireshark. Disektor dodávám ve formě samostatného pluginu. Naimplementoval jsem kompletní specifikaci protokolu, otestoval všechny součásti protokolu a veškeré odhalené chyby opravil. Během implementace jsem nenarazil na žádné zvláštní obtíže; všechny problémy se podařilo vyřešit pomocí konzultace volně dostupné Wireshark dokumentace v souboru README.dissector. Disektor je odolný vůči poškozeným paketům, při přijetí takového paketu dochází k přerušení jeho disekce bez pádu aplikace. Kód disektoru je přenosný mezi platformami Windows a Linux, s ohledem na povahu Wiresharku se ovšem mezi různými operačními systémy liší požadavky na vývojové prostředí. Testy sestavení jsem prováděl na platformě Linux. Dokumentace je dodávána ve formě komentářů v kódu a v rámci práce.

Pro budoucí rozvoj projektu je vhodné dále vylepšovat TTT simulátor protokolu LINK, také jeho API. Simulátor v současném stavu neumožňuje automatickou komunikaci, např. nejsou odesílány Keep-Alive pakety podle nastavené časové prodlevy. Veškerá komunikace simulátoru je pouze explicitní, což s ním sice může v části scénářů usnadnit práci, ale toto řešení postrádá flexibilitu. Zároveň s rozšiřováním možností tohoto nástroje

pro TTT je nutné implementovat další jednotkové testy a testovací scénáře, aby se dalo předejít regresím a zvýšilo se pokrytí kódu testy. Pokud by měl být simulátor využíván nejen pro funkční, ale i pro zátěžové testování, je na zvážení optimalizace všech API pro rychlejší zpracování. Další možností je reimplementace do kompilovaného jazyka pro zvýšení výkonu, pojící se ovšem se ztížením údržby.

Výsledkem práce je funkční základ prostředí pro testování UGC pomocí protokolu LINK, které lze dále rozvíjet ve směru udávaném dalšími potřebami firmy Acision.

ZÁVĚR V ANGLIČTINĚ

In the thesis I explored the development environment in the Acision company, grasping the basic information about UGC and TTT products and about the LINK protocol. I have created tools to provide a proper environment that allows to test the UGC component using the LINK protocol.

During the project I have created a LINK protocol simulator to be used within the TTT tool. I have created both client and server parts. I have verified these parts against the LINK protocol implementation within the UGC component; I also verified their interoperability in mutual connection. I have been carried out rudimentary performance tests on the simulator; its ability has been measured to hundreds of messages per second. The simulator covers almost the entire protocol specification, one of the features missing is connection state tracking – e.g. the Data Request packet is unacceptable before the binding handshake. I have done the testing on the Linux platform and fixed all code errors detected using both manual and automatic tests. All automatic tests are part of the source code. Documentation is provided as part of the source code, HTML output can be generated from it using Doxygen tool.

Furthermore, I have created a LINK protocol dissector for network analyser. I provide the dissector as a standalone plugin. I have implemented a complete protocol specification, all parts have been tested and all detected code errors have been fixed. I have not run into any particular issues during the implementation; all problems have been solved consulting the freely available Wireshark development documentation in README.dissector file. The dissector is resistant to corrupted packets; parsing such a packet will interrupt the dissection without causing application crash. Dissector code is portable between Windows and Linux platforms; due to the Wireshark upstream, however, build environment requirements may differ between platforms. I have carried out build tests on the Linux platform. Documentation is provided within code comments and within the thesis.

For future project evolution it is viable to further improve the TTT simulator of LINK protocol, also its API. The simulator currently does not provide fully automatic communication, such as sending Keep-Alive packets at pre-set timeouts. All simulator communication is explicit, which may help to make its usage easier, but this approach lacks flexibility. At the same time with the extension it is necessary to implement more unit tests and test scenarios to prevent regressions and improve code coverage. If the

simulator was to be used not only for functional, but also for stress testing, performance optimisation of all APIs is to be considered. Another option is reimplementation into compiled language, making its maintenance more complex, however.

The thesis results in a working basic environment for UGC component testing using LINK protocol; further development is possible, based on future Acision requirements.

SEZNAM POUŽITÉ LITERATURY

- [1] What are the Software Development Life Cycle phases? *ISTQB EXAM CERTIFICATION* [online]. 2012-01-13 [cit. 2014-05-14]. Dostupné z: <http://istqbexamcertification.com/what-are-the-software-development-life-cycle-phases/>
- [2] BOOKS LLC. *Software Development Process: Waterfall Model, Computer Programming, Extreme Programming, Capability Maturity Model, Software Testing, Software Architecture, Code and Fix, Revision Control, Spiral Model, Iterative and Incremental Development*. Memphis (USA): Books LLC, 2011, 168 s. Wiki Series. ISBN 1156607671.
- [3] V Model to W Model. *Software Testing Times* [online]. 2010-04 [cit. 2014-04-20]. Dostupné z: <http://www.softwaretestingtimes.com/2010/04/v-model-to-w-model-w-model-in-sdlc.html>
- [4] Manifest agilního vývoje software. *Agile Manifesto* [online]. © 2001 [cit. 2014-04-21]. Dostupné z: <http://agilemanifesto.org/iso/cs/>
- [5] TAKEUCHI, Hirotaka a Ikujiro NONAKA. *The New New Product Development Game*. Boston: Harvard Business Review, p.137-146, 1986.
- [6] Kanban. *Dictionary.com* [online]. 2011 [cit. 2014-05-14]. Dostupné z: <http://dictionary.reference.com/browse/Kanban>
- [7] OHNO, Taiichi. *Toyota Production System: Beyond Large-Scale Production*. Portland, Oregon: Productivity Press. 1988, p. 75-76. ISBN 0-915299-14-3.
- [8] SKARIN, Mattias a Henrik KNIBERG. *Kanban and Scrum: making the most of both*. 2nd ed. S.l.: C4Media, Inc., 2010, 122 s. ISBN 978-0-557-13832-6.
- [9] HUCHENSON, Marnie L. *Software Testing Fundamentals: Methods and Metrics*. Indianapolis (USA): Wiley, 2003, 432 s. ISBN 047143020X.
- [10] GELPERIN, David a Bill HETZEL. *The growth of software testing*. Communications of the ACM. vol. 31, issue 6. DOI: 10.1145/62959.62965. Dostupné z: <http://portal.acm.org/citation.cfm?doid=62959.62965>
- [11] ČSN EN ISO 9000:2006. *Systémy managementu kvality: Základní principy a slovník*. Praha: Český normalizační institut, 2006.
- [12] PATTON, Ron. *Software Testing*. 2nd ed. Indianapolis: Sams, 2005, 408 s. ISBN 0-672-32798-8.
- [13] IEEE Computer Society. *SWEBOK: Guide to the Software Engineering Body of Knowledge*. 3rd ed. Washington D.C.: IEEE Computer Society, 2014, 335 s. ISBN 978-0-7695-5166-1.
- [14] KANER, Cem, James BACH a Bret PETTICHORD. *Lessons Learned In Software Testing: A Context-Driven Approach*. Indianapolis (USA): Wiley, 2001, 352 s. ISBN 0471081124.
- [15] CRISPIN, Lisa a Janet GREGORY. *Agile testing: A Practical Guide for Testers and Agile Teams*. Boston: Addison-Wesley, 2009, 576 s. ISBN 978-032-1534-460.
- [16] ISO/IEC 7498-1:1994. *Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model*. Geneva: International Organization for Standardization, 1994.

- [17] The DoD Four Layer Model. *Sixscape Communications* [online]. 2014 [cit. 2014-02-23]. Dostupné z: <http://www.sixscape.com/joomla/sixscape/index.php/technical-backgrounders/tcp-ip/the-dod-four-layer-model>
- [18] RFC 1122. *Requirements for Internet Hosts - Communication Layers* [online]. Robert Braden. October 1989 [cit. 2014-02-23]. 116 s. Dostupné z: <http://tools.ietf.org/html/rfc1122>
- [19] TORVALDS, Linus. Specs. YARVIN, Norman. *Yarchive: Usenet archives* [online]. 2005 [cit. 2014-04-20]. Dostupné z: <http://yarchive.net/comp/linux/specs.html>
- [20] OREBAUGH, Angela et al. *Wireshark & Ethereal: Network Protocol Analyzer Toolkit*. Gilbert Ramirez. Boston: Syngress, 2007, 448 s. ISBN 15-974-9073-3.
- [21] Frequently Asked Questions. *Wireshark* [online]. [2014] [cit. 2014-04-20]. Dostupné z: <http://www.wireshark.org/faq.html#q1.2>
- [22] HOWTO for Wireshark developers interested in writing or working on Wireshark protocol dissectors. *Wireshark* [online]. [2014] [cit. 2014-01-20]. Dostupné z: anonsvn.wireshark.org/wireshark/trunk/doc/README.dissector
- [23] Wireshark: Network Analyzer. *Havy Software* [online]. [2012] [cit. 2014-02-15]. Dostupné z: <http://havyssoftware.com/windows-software/networking-software/network-tools/wireshark.html>
- [24] Wireshark User's Guide. *Wireshark* [online]. 2013 [cit. 2014-04-20]. Dostupné z: http://www.wireshark.org/docs/wsug_html/
- [25] HRUŠKA, Jiří. ACISION. *LINK: Light-Weight GBG Internal Protocol*. Version 2. Praha, 2012.
- [26] FLYNT, Clif. *Tcl/Tk: A Developer's Guide*. 3rd ed. Waltham (USA): Morgan Kaufmann, 2012, 816 s. ISBN 0123847176.
- [27] HEINE, Gunnar. *GSM Networks: Protocols, Terminology and Implementation*. Boston: Artech House, 1999, 417 s. ISBN 0-89006-471-7.
- [28] SMPP Developers Forum. *Short Message Peer to Peer: Protocol Specification v3.4*. Version 1.2. Dublin, 1999.
- [29] RFC 6733. *Diameter Base Protocol* [online]. Victor Fajardo, Glen Zorn et al. October 2012 [cit. 2014-02-23]. 152 s. ISSN 2070-1721. Dostupné z: <http://tools.ietf.org/html/rfc6733>
- [30] RFC 4511. *Lightweight Directory Access Protocol (LDAP): The Protocol* [online]. Jim Sermersheim, Novell, Inc. June 2006 [cit. 2014-02-23]. 68 s. Dostupné z: <http://tools.ietf.org/html/rfc4511>
- [31] HORAK, Ray. *Telecommunications and data communications handbook*. Hoboken, N.J.: Wiley-Interscience, 2007. 111 s. ISBN 0470127228.
- [32] Don't know what I want, but I know how to get it. PATTON, Jeff. *AgileProductDesign.com* [online]. 2008-01-21 [cit. 2014-04-21]. Dostupné z: http://www.agileproductdesign.com/blog/dont_know_what_i_want.html

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

AFG	Acision Flexible Gateway
API	Application Programming Interface
AVP	Attribute-Value Pair
CDMA	Code Division Multiple Access
CDR	Call Data Record
CLI	Command-line Interface
ČSN	Česká státní norma
DNS	Domain Name Service
DoD	Department of Defense
DoS	Denial of Service
FTP	File Transfer Protocol
GBG	Generic Billing Gateway
GIMP	GNU Image Manipulation Program
GNU	GNU is Not Unix
GPL	GNU Public Licence
GSM	Global System for Mobile Communications (původně Group Spécial Mobile)
GTK	GIMP Tool Kit
GUI	Graphical User Interface
HLR	Home Location Register
HTTP	Hypertext Transfer Protocol
ICPM	Internet Control Message Protocol
ICT	Information and Communications Technology
ID	Identification number
IDS	Intrusion Detection System

IMSI	International Mobile Subscriber Identity
IP	Internet Protocol
IPX	Internet Package Exchange
ISO	International Organization for Standardization
JIT	Just-in-time
LDAP	Lightweight Directory Access Protocol
MITM	Man-in-the-middle
MSISDN	Mobile Subscriber Integrated Services Digital Network-Number
OS	Operační systém
OSI	Open Systems Interconnection
PCAP	Promiscuous Capture Library
PDU	Protocol Data Unit
RDP	Remote Desktop Protocol
SDR	Service Detail Record
SIM	Subscriber Identity Module
SMPP	Short Message Peer-to-Peer
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
SSH	Secure Shell
Tcl	Tool Command Language
TCP	Transmission Control Protocol
TDD	Test-Driven Development
TGC	Test Gateway Component
TTL	Time-to-live
TTT	Tcl Testing Tool
UDP	User Datagram Protocol

UGC	Universal Gateway Component
VoIP	Voice over IP
WIP	Work in Progress
XMPP	Extensible Messaging and Presence Protocol
XP	Extreme Programming

SEZNAM OBRÁZKŮ

Obr. 1. Schéma W-modelu [3].	13
Obr. 2. Znázornění inkrementálního vývoje [32].	13
Obr. 3. Znázornění iterativního vývoje [32].	14
Obr. 4. Síťový model ISO/OSI [17].	23
Obr. 5. Síťový model TCP/IP (či DoD) [17].	27
Obr. 6. Výchozí styl uživatelského rozhraní analyzátoru Wireshark [23].	31
Obr. 7. Použití barev k odlišení různých druhů paketů [24].	32
Obr. 8. Architektura UGC.	35
Obr. 9. Ukázka průběhu zprávy přes UGC.	36
Obr. 10. Průběh komunikace paketem Keep-Alive.	41
Obr. 11. Průběh navázání spojení.	41
Obr. 12. Průběh ukončení spojení.	42
Obr. 13. Průběh zasílání dat.	43
Obr. 14. Disektor protokolu LINK zpracoval několik paketů tohoto protokolu.	61

SEZNAM TABULEK

Tab. 1. Základní struktura paketu protokolu LINK.	39
Tab. 2. Seznam operačních kódů protokolu LINK.	40
Tab. 3. Seznam dostupných výsledkových kódů.	43
Tab. 4. Ukázka použití simulátoru protokolu LINK v základním scénáři.	57

SEZNAM PŘÍLOH

P I Obsah CD

PŘÍLOHA P I: OBSAH CD

Součástí diplomové práce je CD se zdrojovými kódy projektu.

Text diplomové práce je uložen v souboru fulltext.pdf.

Obsah CD:

- fulltext.pdf: text diplomové práce
- README: tento soubor
- scenario.pcap: ukázkový tcpdump s pakety modulu LINK
- ttt-ugclink: obsahuje simulátor LINK protokolu pro Tcl interpret
- wireshark-ugclink: obsahuje Wireshark disektor pro LINK protokol

- CD/ttt-ugclink:
 - CL-LINK.tcl: kompletní simulátor LINK protokolu pro Tcl/TTT

- CD/wireshark-ugclink:
 - config.c: doplňkový zdrojový kód s obsluhou ugclink.ini
 - Makefile: předpis pro úspěšné sestavení
 - packet-ugclink.c: hlavní zdrojový kód
 - ugclink.ini: slovník pro překlad číselných hodnot do člověku srozumitelných názvů

Návod k použití:

- Jak sestavit a nainstalovat Wireshark disektor:

Zavoláním příkazu make; soubor s pluginem bude automaticky vytvořen v adresáři Wiresharku a definiční INI soubor ugclink.ini nakopírován k němu.

- cd wireshark-ugclink
- make

Objektové soubory zůstávající po kompilaci lze odstranit pomocí příkazu make clean.

- Jak odinstalovat Wireshark disektor:

Zavoláním příkazu `make uninstall`; soubor s pluginem a definiční INI soubor budou odstraněny z adresáře Wiresharku.

- `cd wireshark-ugclink`
- `make uninstall`

- Jak používat Wireshark disektor:

Po instalaci pomocí příkazu `make` stačí otevřít libovolný soubor se zachyceným síťovým provozem, do řádku filtrů zadat "ugclink" a potvrdit klávesou ENTER. Pro vyzkoušení lze například použít soubor `scenario.pcap`, který je přiložen.

- Jak instalovat a odinstalovat LINK simulátor pro TTT:

Simulátor není třeba instalovat ani odinstalovat. Stačí nakopírovat soubor simulátoru `CL-LINK.tcl` do libovolného adresáře.

- Jak používat LINK simulátor pro TTT:

Pro spuštění je třeba interpret jazyka Tcl (například `tclsh`). Do něj je nutné nahrát zdrojový soubor `CL-LINK.tcl` pomocí Tcl příkazu `source`.

- `cd ttt-ugclink`
- `tclsh`
- `source CL-LINK.tcl`

Další podrobnosti, dokumentace API a ukázky použití jsou k dispozici v kapitolách 9 a 10 a přímo ve zdrojovém kódu.