VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

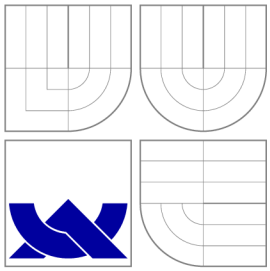
IMPLEMENTACE KOMUNIKAČNÍHO MIDDLEWARE V PROSTŘEDÍ JAVA ME

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

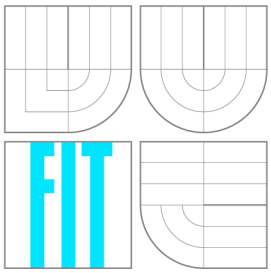
AUTOR PRÁCE
AUTHOR

Bc. JAN MARTINÁK

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

IMPLEMENTACE KOMUNIKAČNÍHO MIDDLEWARE V PROSTŘEDÍ JAVA ME

IMPLEMENTATION OF COMMUNICATION MIDDLEWARE IN JAVA ME

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAN MARTINÁK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN KOŘENEK

BRNO 2009

Zadání práce

1. Nastudujte problematiku tvorby komunikačního middleware v prostředí JAVA ME. Soustřeďte se na SOA infrastrukturu.
2. Navrhněte aktivní komponentu včetně způsobu konfigurace a nástrojů pro generování zpráv.
3. Dále navrhněte XML schéma pro komunikaci s komponentou.
4. Proveďte implementaci navrženého řešení v jazyku Java ME s ohledem na využití ve vestavěném multifunkčním zařízení.
5. Vytvořte demonstrační aplikaci, která ukáže možnosti navržené komponenty. Snažte se o pokrytí všech funkcí poskytovaných komponentou a vestavěným zařízením.
6. Zhodnoťte dosažené výsledky a možnosti využití navržené komponenty v různých aplikacích.

Licenční smlouva

Abstrakt

V prostředí téměř každého podniku existuje softwarová podpora pro podporu obchodních procesů. S narůstajícím počtem těchto aplikací roste i potřeba integrovat je tak, aby vzniklo komplexní prostředí, které funguje efektivně a generuje zisk. V oblasti integrace podnikových aplikací existuje několik principů, každý se svými výhodami i nedostatky, technika zasílání zpráv se však pro mnoho integračních problému ukázala jako nejvýhodnější. Tato práce se zabývá podnikovým prostředím s aplikacemi vestavěných systémů multifunkčních kancelářských zařízení na platformě Java ME a prezentuje vlastní řešení uniformní komunikační vrstvy pro integraci těchto aplikací do podnikové infrastruktury. Výsledná softwarová komponenta aplikuje principy techniky zasílání zpráv v SOA pro správu firemního tisku, kde je spolu s navrženým schématem zpráv použita v komunikaci s tiskovým serverem.

Klíčová slova

Integrace aplikací, komunikační vrstva, zasílání zpráv, vestavěný systém, multifunkční zařízení

Abstract

In almost each enterprise there is a software support for business processes. With a growing number of the applications there is an increasing demand to integrate those applications in order to have an effectively working environment which generates profit. In enterprise applications integration there are few principles, each with their own advantages and disadvantages. However, the message-oriented middleware layer proves to be the best solution to many integration scenarios. This work deals with an enterprise of applications running on multifunction embedded office devices based on Java ME platform, and introduces an in-house developed communication middleware layer to integrate those applications. The resulting software component applies principles of messaging in a printing management SOA environment to communicate with a print server using a designed set of messages.

Keywords

Enterprise integration, middleware, messaging, embedded system, multifunction device

Citace

Jan Martinák: Implementace komunikačního middleware v prostředí Java ME, diplomová práce, Brno, FIT VUT v Brně, 2009

Implementace komunikačního middleware v prostředí Java ME

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Jana Kořenka. Další informace a rady mi poskytli odborní pracovníci z firmy Y Soft s.r.o. Dále prohlašuji, že jsem uvedl všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jan Martinák
25. května 2009

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Janu Kořenkovi za konzultace formální stránky technické zprávy a pracovníkům z firmy Y Soft s.r.o. za odbornou pomoc.

Obsah

Kapitola 1

Úvod

Každý podnik má svůj informační systém. Tento systém může být buď manuální, nebo určitým způsobem automatizovaný. Zahrnuje lidi a stroje (počítače či jiné technické prostředky), včetně metod organizovaných pro sběr, zpracování, přenos a šíření dat, která reprezentují uživatelskou informaci [21]. Podnikové informační systémy velkých firem se však v dnešní době bez rozsáhlé softwarové podpory neobejdou. Aby fungovaly rychle, spolehlivě a úsporně, a tím generovaly zisk, jsou nuceny nahradit lidský faktor softwarovými aplikacemi. S rostoucím počtem takových různorodých aplikací však podnik opět ztrácí na efektivitě. Vzniká redundance dat, kdy každá aplikace vlastní svou kopii těch samých dat, což znesnadňuje správu celého systému. Podobný problém platí i pro funkcionalitu — proč mít více způsobů realizace jedné funkce, když můžeme mít jednu implementaci, která dělá přesně to, co požadujeme, a poskytovat ji ostatním aplikacím. Řešením je sjednotit tyto aplikace, tedy *integrovat*.

Tato práce si klade za cíl seznámit čtenáře s obecnými způsoby integrace aplikací a prostředky, které lze k integraci použít. Předmětem samotné práce je potom komponenta, realizující jeden z prostředků integrace — zasílání zpráv (*messaging*), tzv. *komunikátor*. Tato komponenta má za úkol poskytnout jednotné rozhraní pro vzájemnou komunikaci aplikací pomocí funkčního řešení softwarové komunikační vrstvy, založené na principu zasílání zpráv (*message-oriented middleware*).

Komunikátor je primárně vyvíjen pro platformu Java ME, určenou pro běh aplikací na vestavěných zařízeních, kde bude také nasazen a testován. Přesto je navržen tak, aby byly jeho implementace použitelné v širším spektru softwarových platforem. Plánované využití této komponenty je jako součást nové generace vestavěných softwarových terminálů multifunkčních kancelářských zařízení od firmy Ricoh. Tato zařízení umožňují běh aplikací díky platformě Embedded Software Architecture založené právě na Javě ME. Zmíněný vestavěný terminál se stane součástí distribuované infrastruktury řešení pro správu firemního tisku SafeQ vybudované v duchu SOA a bude využívat komunikátor pro komunikaci s jednotlivými entitami SafeQ.

Kapitola 2 poskytuje obecnější pohled na problematiku propojování systémů. Zabývá se obecnými otázkami integrace aplikací a věnuje se mj. prostředí SOA. V následující kapitole 3 jsou probrány softwarové prostředky integrace aplikací se zvláštním důrazem na techniku zasílání zpráv (kap. 3.4). Kapitola 4 seznamuje čtenáře s prostředím SafeQ a s principy a architekturou multifunkčních zařízení od firmy Ricoh, pro kterou je komunikátor primárně vyvíjen. Následující kapitola 5 již rozvíjí motivaci pro vývoj komunikátoru, definuje problémy, které je nutné řešit, a předkládá konkrétní požadavky a omezení výsledného produktu. V kapitole 6 je vytvořena formální analýza těchto požadavků, která dále v kapi-

tole 7 slouží jako podklad pro návrh architektury komunikátoru. V téže kapitole je rovněž definován základ architektury vestavěného terminálu s grafickým uživatelským rozhraním pro multifunkční zařízení Ricoh (kap. 7.5) a navržena XML struktura zpráv a některé možné typy zpráv použitelné v SafeQ (kap. 7.3). Poslední kapitola 8 se zabývá nasazením komunikátoru v prostředí s vestavěným terminálem a demonstruje jeho použitelnost na klasické úloze zabezpečeného odloženého tisku, kdy si uživatel tiskne úlohu až po potvrzení na terminálu.

Tato diplomová práce plně navazuje na stejnojmenný semestrální projekt vytvořený v zimním semestru, který se zabýval především teoretickými základy problematiky integrace aplikací (kap. 2) a principu zasílání zpráv (kap 3.4). Jeho výsledkem byla analýza požadavků a omezení pro vyvíjenou komponentu a koncept architektury (viz kap. 6). Zde je práce dále rozvinuta krátkou studií o cílovém prostředí (viz kap 4), čímž zakončuje teoretickou část práce, a doplněna detailním návrhem a implementací všech požadovaných částí.

Kapitola 2

Integrace aplikací

Integrace aplikací (*enterprise integration*) je široký pojem. Neznamená pouze propojení počítačových aplikací, ale v širším kontextu zahrnuje propojení lidí či celých společností v podnikovém prostředí *byznys procesů*. Byznys proces je posloupnost aktivit, jejichž výsledkem je produkt nebo služba poskytnutá zákazníkovi na základě jeho požadavků [20]. Podnikové aplikace redukují lidský chybový faktor a zajišťují, aby tyto procesy fungovaly rychle, spolehlivě a efektivně.

Prostředí podnikových aplikací bývá silně heterogenní. Figurují v něm aplikace vybudované na různých platformách, za použití různých technologií a programovacích jazyků. Typickým příkladem je firma s několika odděleními (marketingové oddělení, finanční oddělení, oddělení výroby, správy informačních technologií — IT a lidských zdrojů — HR). Jak se firma vyvíjí, přibývají softwarové aplikace, které slouží jednotlivým oddělením a postupně nahrazují předchozí „papírové“ informační systémy.

Tyto aplikace však zároveň zapouzdřují informace uvnitř oddělení a často neberou v potaz existenci ostatních aplikací (*information silo* [7]). Jakmile chce tedy firma automatizovat nějakou činnost, která zahrnuje spolupráci více oddělení, je nutné tyto úkony provést v každém systému odděleně a výměnu dat realizovat obskurními způsoby (*sneakernet*, *swivel-chair integration*[7]). Jakkoliv jsou tyto způsoby nenáročné na implementaci, jsou značně neefektivní, náchylné k chybám a v konečném důsledku vedou k nespokojenosti zákazníka.

2.1 Typy integrací

Integrace je proces, který v různém kontextu nabývá různých podob. V zásadě lze identifikovat několik integračních scénářů, které opakovaně vyvstávají na povrch při řešení problémů tohoto typu [6]. Tyto scénáře představují různé varianty řešení podnikové integrace a lokalizují problémy, které vyvstávají při jejich použití. V praxi se často vyskytují v různých kombinacích, v závislosti na požadavcích byznys procesů:

- Informační portál
- Sdílená funkcionalita
- Architektura orientovaná na služby
- Sdílená data

Uvedený výčet slouží jako kostra pro identifikaci problémů integrace. Zvláštní pozornost je věnována konceptu architektury orientované na služby, neboť představuje prostředí námi vyvíjeného komunikačního rozhraní.

2.1.1 Informační portál

Informační portály sdružují informace z různých částí podnikového systému a prezentují je koncovému uživateli skrze modulární uživatelské rozhraní. Jde o integraci na prezentační úrovni (*portal integration* [14]. Uživatel nemusí přistupovat ke každému systému zvlášť, informační portál tato data transparentně získá a zobrazí v jednotlivých sekcích jednotného uživatelského rozhraní.

Uživateli jsou prezentována data a zpřístupněny funkce z různých systémů a je čistě na něm, jak s nimi bude operovat. Takto je zabezpečena flexibilita v rozhodování o jednotlivých krocích byznys procesu, na druhou stranu operátoři manipulující s daty dělají často při jeho plnění chyby. Tento způsob integrace je nejméně efektivní, z hlediska implementace je však zároveň nejjednodušším řešením problému integrace aplikací.

Informační portály nabízejí různé stupně interakce mezi výstupy z jednotlivých systémů, obecně je však tento typ interakce značně neefektivní. Příkladem je tzv. *screen scraping*, což je technika automatického získání užitečných dat z jejich vizuální reprezentace v jednom systému a jejich vložení do druhého systému pomocí emulace terminálu [14]. Tato technika byla obzvlášť v dřívějších dobách jedinou možností integrace kvůli absenci aplikačních rozhraní na úrovni byznys logiky v jednotlivých aplikacích.

Nástup webových rozhraní výrazně zjednodušil rozvoj tohoto typu integrace. Příkladem jsou elektronické obchody, které prezentují informace o stavu zásob zboží a zároveň stavu doručení objednávky.

2.1.2 Sdílená funkcionalita

Hlavním problémem, který sdílení funkcionality řeší, je realizace distribuovaných byznys procesů. Takové procesy se rozprostírají přes více podnikových aplikací, přičemž každá aplikace poskytuje specifickou funkcionalitu. Byznys proces tak má charakter zpracování požadavku v postupných sousledných krocích (např. po přijetí objednávky postupně následuje její validace, ověření skladových zásob, výpočet daně atd.) [6].

Základní podmínkou pro sdílení funkcionality je poskytnutí vhodného rozhraní [12]. Nejintuitivnější přístup spočívá v exportu aplikačního programového rozhraní (API), které umožní klientským aplikacím pracovat přímo s objekty aplikační logiky. Z hlediska technické realizace je základním řešením použití vzdáleného volání procedur nebo distribuovaných objektů, např. standardu CORBA[1], potažmo aplikačních rozhraní, jako Java RMI[2], nebo .NET Remoting[3]. Jinou možností je využití technologií na bázi zasílání zpráv, např. standardů SOAP[4], JMS API[5], potažmo komerčních řešení (IBM WebSphere MQ[6], Microsoft MSMQ[7]). Každý přístup má však své pro a proti a záleží na požadavcích prostředí, ve kterém integrační řešení vyvíjíme (více v kap. 3).

[1] The official CORBA standard from the OMG group - http://www.omg.org/docs/formal/04-03-12.pdf
[2] Remote Method Invocation Home - http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp
[3] .NET Remoting Overview - http://msdn.microsoft.com/en-us/library/kwdt6w2k(VS.71).aspx
[4] SOAP Version 1.2 Part 1: Messaging Framework - http://www.w3.org/TR/soap12-part1/
[5] Java Message Service - http://java.sun.com/products/jms/
[6] WebSphere MQ Product Page - http://www-01.ibm.com/software/integration/wmq/
[7] Microsoft Message Queueing - http://www.microsoft.com/windowsserver2003/technologies/msmq/default.mspx

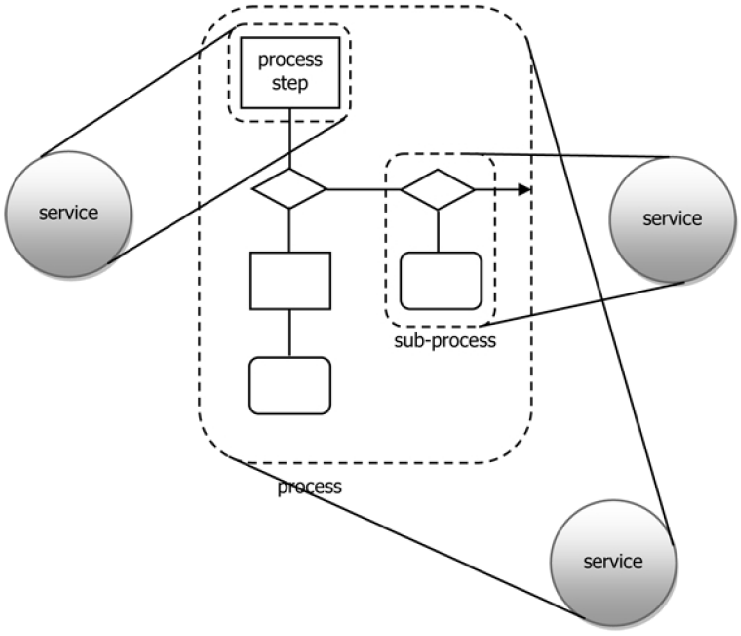
Sdílení funkcionality souvisí se sdílením a replikací dat [6]. Například můžeme požadovat implementaci sdílené funkce pro získání adresy zákazníka, která dovoluje klientským aplikacím přístup k těmto datům, až když je skutečně potřebují, než aby je uchovávaly ve svém lokálním úložišti. Data jsou navíc zapouzdřena, což odstraňuje nevýhody přímého sdílení (viz kap. 2.1.4). Rozhodnout, zda data replikovat, nebo poskytnout sdílenou funkcí je otázkou mnoha kritérií jako např. frekvence změny (adresa může být žádána často, ale k její změně dochází zřídka).

2.1.3 Architektura orientovaná na služby

SOA (*Service Oriented Architecture*) je způsob vývoje systémů, které zakládají svoji funkcionalitu na sémantice byznys procesů a zapouzdřují ji v diskrétních jednotkách nazývaných služby [3]. Tato architektura je speciálním příkladem aplikace sdílení funkcionality v podnikové integraci a vyznačuje se jistými charakteristickými rysy.

Služba je v podstatě sdílená funkce s danými vstupy a výstupy (*contract*). Prostředí SOA však navíc definuje mechanismy a infrastrukturu pro přístup a vyhledávání těchto služeb, čímž zajišťuje jejich opakovatelnou použitelnost. Koncept služeb vychází z objektově-orientovaného návrhu. Zapouzdřují logiku byznys procesů, případně jejich částí, rozsahem tak mohou pokrývat mnohem větší funkcionalitu než klasické objekty tříd. Princip zapouzdření s jasně daným rozhraním však zůstává.

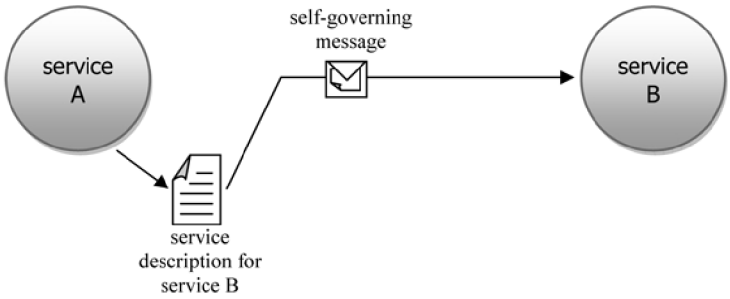
Aplikace se v prostředí SOA vyvíjejí skládáním dostupných služeb, což je velmi efektivní a přínosné z hlediska flexibility správy byznys procesů. Množství logiky byznys procesu, kterou může služba obsáhnout, není striktně ohraničeno. Jak ukazuje obr. 2.1, služba zapouzdřuje určitý krok nebo podproces složený ze sekvence kroků. Může však také zapouzdřit kompletní byznys proces.



Obrázek 2.1: Struktura služeb (převzato z [3])

SOA je založena na paradigmatu distribuovaného programování, klade ovšem specifické požadavky na způsob návrhu služeb, zpráv a vztahů mezi nimi. Na rozdíl od distribuované aplikace, kde jsou její jednotlivé výpočetní uzly svázány těsnými vazbami, danými pevně daným rozhraním objektů, aplikace v servisně-orientovaném prostředí spoléhají na volné vazby mezi službami (*loose coupling* [7]) v čistě heterogenním prostředí a volání vzdálené služby je umožněno díky vhodnému komunikačnímu rámci.

Oním vhodným komunikačním rámcem může být např. zasílání zpráv (*messaging*). Způsob, jakým spolu služby komunikují, je popsán tzv. deskriptorem služby. Tento deskriptor v základě obsahuje název služby, její vstupy a výstupy. Pokud chce služba A komunikovat se službou B, musí mít k dispozici deskriptor služby B, teprve poté může probíhat užitečná komunikace. Deskriptory a užitečná data jsou přenášena právě pomocí autonomních zpráv (viz obr. 2.2).



Obrázek 2.2: Komunikace služeb (převzato z [3])

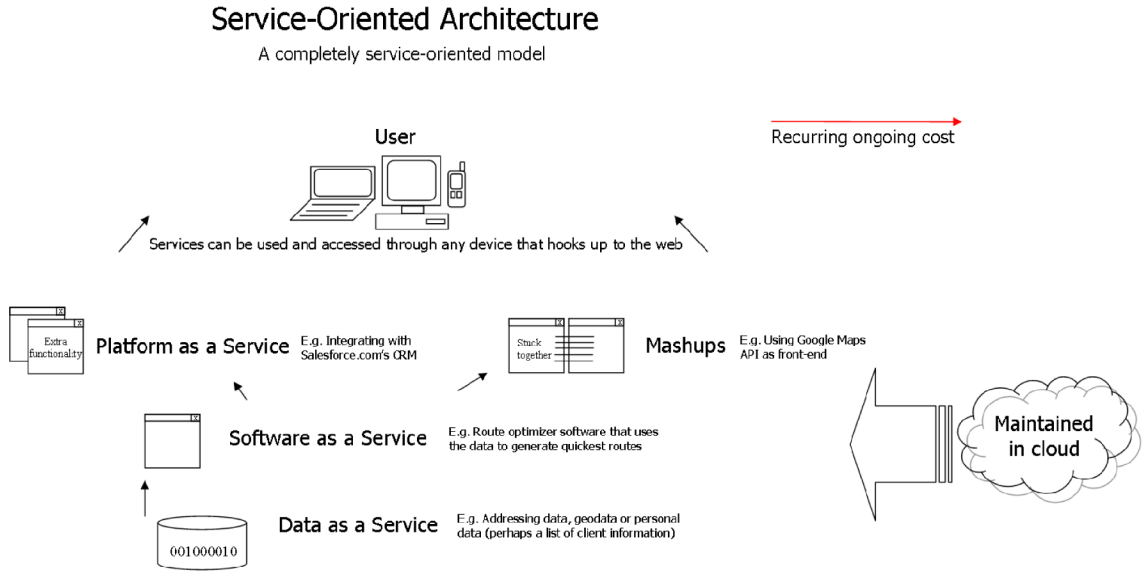
SOA není definovaný standard, ale spíše paradigma pro efektivní tvorbu aplikací, které delegují určité klíčové činnosti na aplikace jiné. Koncept SOA není svázán s konkrétní technologií, na internetu a v literatuře se však často hovoří o SOA jako o prostředí webových služeb (standard W3C Web Services[8]). Důvod je zřejmě ten, že je pro implementaci tohoto konceptu obzvlášť vhodný, protože maximalizuje míru uvolňování vazeb. Formát zpráv a protokol pro komunikaci s webovou službou je dán standardem SOAP, který definuje strukturu zprávy jako XML dokument, přenášený přes webový protokol HTTP. Pro vybudování SOA je však také možné využít komplexních komerčních řešení, opět však nejčastěji založené na principu zasílání zpráv. Rozbor výše zmíněných standardů přesahuje rámec této práce, detaily jsou diskutovány v literatuře [3] a [7].

Obr. 2.3 ilustruje příklad aplikace v prostředí SOA. Na nejnižší úrovni leží služba pro získání dat z datového úložiště (zde zeměpisná data, adresy). Tato data jsou dále zpracována jinou službou (zde vyhledání nejkratší cesty za použití zeměpisných dat). Výsledná data mohou být pak využita podnikovými platformami (CRM systém) nebo prezentována koncovému uživateli pomocí služby Google Maps API [23].

2.1.4 Sdílení dat

Sdílení dat představuje formu integrace aplikací na nejnižší úrovni. Smyslem je poskytnout jednotný pohled a přístup k určitému typu informace. Problém sdílení dat má dvě roviny: poskytnutí jednotného pohledu na určitý typ dat (datovou entitu, viz [11]) a udržení konzistence jednotlivých instancí těchto dat mezi integrovanými aplikacemi.

[8] http://www.w3.org/2002/ws/



Obrázek 2.3: Příklad SOA (převzato z [23])

Představme si např. datovou entitu s názvem Zákazník. Tato entita vystupuje v různých systémech: CRM systém, účetní systém a doručovatelský systém. Každý z těchto systémů udržuje vlastní pohled na tuto datovou entitu ve vlastním datovém úložišti. Bylo by ovšem žádoucí, aby se s informacemi o zákazníkovi manipulovalo jednotným způsobem ve všech aplikacích, které informace o zákazníkovi potřebují sdílet, neboli disponovat vrstvou, která zajistí agregaci potřebných informací. Pokud by potom uživatel jedné aplikace změnil údaje určitého zákazníka, změna by byla viditelná i v ostatních aplikacích.

Zatímco zajištění jednotného pohledu na data je záležitostí definice jednotné entity Zákazník, která je schopna obsáhnout informace potřebné ve všech aplikacích (např. jednotný formát adresy), existuje více způsobů pro zajištění konzistence dat v těchto aplikacích. Buď agregační vrstva disponuje přímým spojením s úložišti ostatních aplikací a veškeré změny jsou prováděny v reálném čase, nebo jsou data replikována a agregační vrstva pracuje s jednotnými entitami ve svém úložišti.

Přímé sdílení dat může být přínosné. Není to invazivní technika, kdy zasahujeme do struktury aplikací, protože aplikace bývají navrženy s oddělenou datovou vrstvou (např. databáze, která z podstaty umožňuje transakční přístup pro více uživatelů). Aplikace však mívají přístup k datům v nezapouzdřené podobě, což ale může být nevýhoda, neboť data nemusí být čitelná pro ostatní aplikace a případně změny mohou narušit individuální integritu dat.

Kapitola 3

Prostředky integrace

Jakmile je rozhodnuto o použití integrace, je na čase zvolit technologii, která umožní propojení systémů. V této kapitole jsou popsány čtyři hlavní přístupy k integraci: sdílení souborů, sdílení databáze, vzdálené volání procedur a zasílání zpráv. Žádný z těchto přístupů není ten nejlepší a každý má své výhody i nevýhody, které zde uvedeme. Jde o to, vybrat v daném případě ten nejvhodnější, případně zvolit kombinaci, která zajistí vyvážené řešení integračního problému.

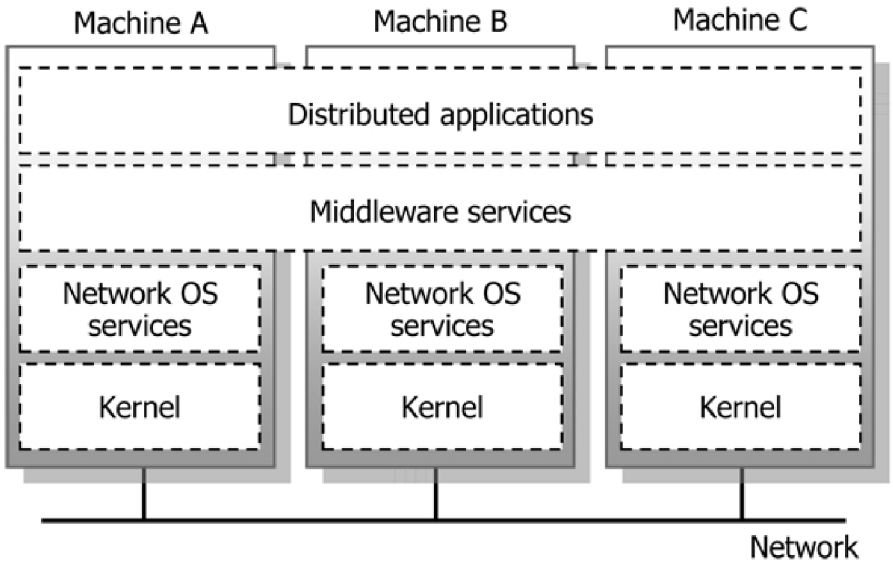
Jednotlivé přístupy mají specifické vlastnosti, které vymezují jejich použitelnost při řešení daného problému. Jednou z hlavních vlastností je již zmiňovaná míra uvolňování vazeb. Aplikace vystavěné na tomto principu radikálně usnadňují podnikovou integraci, protože jsou navrženy tak, aby skrz vnitřní změny neovlivňovaly funkčnost aplikací druhých — používají vhodné komunikační prostředky. Takový návrh obětuje optimalizaci výkonnosti ve prospěch zajištění interoperability mezi systémy různých technologií, výkonností, dostupností a umístnění [7]. Systém takových aplikací se pak snadněji adaptuje na nečekané změny v jednotlivých jeho částech.

Právě uvolňování vazeb je důvod proč věnovat návrhu integrace tolik pozornosti a nespokojit se s použitím proprietárního textového protokolu [6]. Přestože se to na první pohled nezdá, tento přístup není pro účely integrace ideální: obě komunikující strany uvažují o té druhé z hlediska technologie platformy (odlišná reprezentace datových typů), lokality (problém změny IP adresy), času (problém synchronní komunikace — obě strany se musejí podílet na výměně dat zároveň) a formátu zasílaných dat (modifikace protokolu vede na změny na obou stranách). Všechny tyto požadavky kladou velké nároky na znalosti aplikace o svém okolí. Pokud však dokážeme tyto nedostatky eliminovat, jsme schopni zajistit, aby se aplikace vyvíjela ve svém prostředí nezávisle na něm, což v konečném důsledku ušetří peníze i čas.

Dalším jevem, který je patrný především v datové integraci, je neshoda dat [13]. V zásadě existují dva typy neshody dat. Syntaktická neshoda dat (*structural dissonance*) znamená problém rozdílné reprezentace dat, sémantická neshoda dat (*semantic dissonance*) představuje rozpor dvou aplikací v chápání smyslu (sémantiky) dat, kdy je obtížné určit, co přesně data znamenají (např. zda je udávaná cena s daní nebo bez daně). Oba problémy vyvstávají při transformaci dat, kdy je nutné převádět data z formátu, který používá jedna aplikace, do formátu vlastního aplikaci druhé. Zatímco syntaktická neshoda se automaticky řeší snadno podle jasně daných pravidel, sémantická neshoda představuje problém, který je nutno zohlednit při návrhu transformace.

Výše zmíněná omezení jsou rozhodující pro výběr vhodné komunikační vrstvy, tzv. *middleware*. Tento termín označuje softwarovou vrstvu tvořící komunikační rozhraní mezi

klientskými aplikacemi v distribuovaných systémech [22]. Jeho primární účel je zapouzdřit komunikaci dvou aplikací a snížit tak vazby v systému, často však poskytuje další užitečné služby, jako např. transakce a monitorování komunikace. Middleware také redukuje závislost klientských aplikací na operačním systému (např. abstrahuje od používání síťového rozhraní), čímž zajišťuje interoperabilitu heterogenních systémů.



Obrázek 3.1: Schéma middleware (převzato z [17])

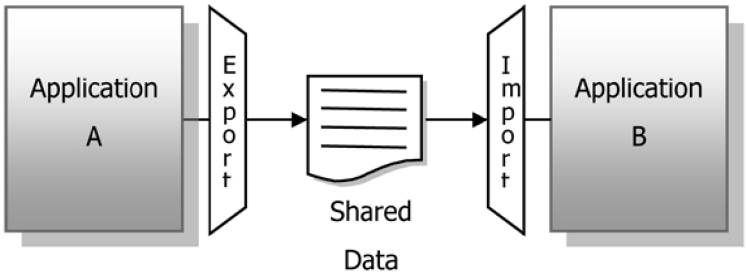
Middleware může být obecně vybudován nad různými technologiemi, počínaje sdílením souborů a zasíláním zpráv konče. Jednotlivé typy mají specifické vlastnosti, výhody i nevýhody, a to jaký typ použít často závisí na konkrétní situaci.

3.1 Sdílení souborů

Sdílení souborů je nejzákladnější technika integrace aplikací, která slouží k předávání dat. Je velmi jednoduchá, nevyžaduje speciální hardware nebo software, a proto je široce podporovaná aplikacemi na rozličných platformách. Jedna z aplikací exportuje soubor v určitém formátu aplikačních dat na určité místo ve sdíleném souborovém systému a ostatní aplikace tento soubor čtou.

Výhoda sdílení souborů spočívá v redukci vazeb mezi aplikacemi, protože exportovaný soubor může být libovolně transformován nezávisle na zdrojové aplikaci tak, aby vyhověl požadavkům druhých aplikací na formát dat. Zdrojová aplikace se může libovolně vyvíjet, ovšem za předpokladu zachování exportovaných dat v dohodnutém tvaru. Požadavek na formát dat v souboru je důležitým rozhodnutím při sdílení souborů. V dnešní době je časté využití formátování podle standardu XML, kolem kterého byla vybudována rozsáhlá softwarová podpora pro jeho zpracování (viz kapitola 3.5).

Nevýhoda sdílení souborů spočívá v načasování výměny dat mezi systémy. Z důvodů zvýšené zátěže není vhodné provádět export příliš často, na druhou stranu nízká frekvence může vést ke ztrátě synchronizace mezi systémy a používání zastaralých dat. Další nevýhodou je již zmiňovaná sémantická neshoda dat: Pokud transformujeme soubor s daty

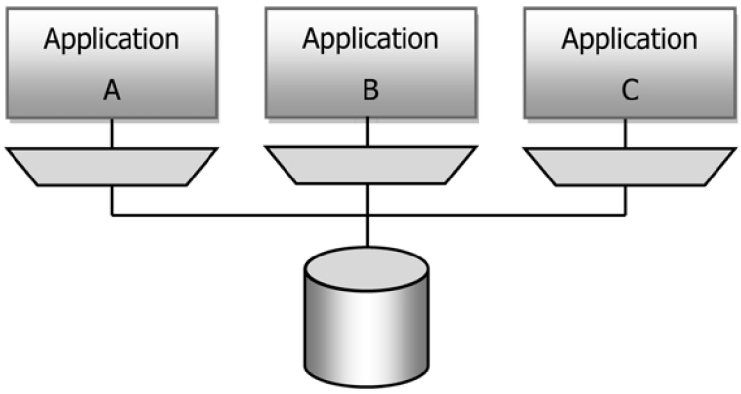


Obrázek 3.2: Sdílení dat (převzato z [6])

exportovanými z jedné aplikace do formátu pro import aplikace druhé, nestačí pouze automaticky provést syntaktickou konverzi (např. z XML do CSV[1]), ale zároveň je nutné být obeznámen s přesným významem jednotlivých datových položek, aby importovaný soubor obsahovat přesně tu informaci, kterou cílová aplikace požaduje.

3.2 Sdílená databáze

Sdílená databáze představuje centrální úložiště dat, dostupné všem aplikacím. Návrh schématu databáze musí podléhat požadavkům všech integrovaných aplikací a systém řízení báze dat zajišťuje konzistenci dat při jejich získávání a modifikaci. Tím je odstraněna nevýhoda zastaralosti dat, která byla patrná při použití sdílení souborů. Problém nekonzistence v reprezentaci dat je z větší části odstraněn použitím standardizovaného jazyka SQL, který je značně rozšířen v oblasti relačních databází. Nejednoznačnosti v interpretaci sémantiky jsou řešeny již ve fázi návrhu integrace, neboť aplikace se musí shodnout na společném schématu dat. Tím je možné se vyvarovat problémům, které mohu nastat až během provozu.



Obrázek 3.3: Sdílení databáze (převzato z [6])

Návrh vhodného databázového schématu, který splňuje všechny požadavky klientských aplikací, může na druhou stranu vést k rigidním schématům, se kterými se obtížně pracuje.

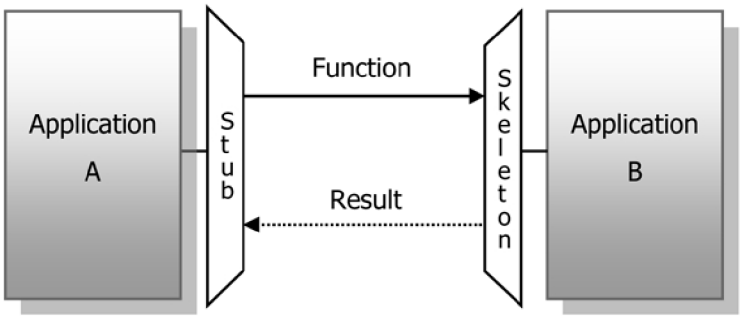
[1] Comma-separated values — hodnoty oddělené čárkami (viz RFC 4180)

Aplikace jsou odstíněny jedna od druhé za cenu svázání s jednotnou strukturou dat. Tento problém se stává o to větším při integraci systémů třetích stran, které spoléhají na určité databázové schéma, které se obvykle mění s novou verzí, a programátor má jen velmi omezené možnosti s tímto požadavkem něco dělat. Další problém představuje výkonnost. Sdílená databáze se snadno může stát úzkým hrdlem systému. Tento problém lze sice řešit replikací dat, potom však musíme počítat s nutností synchronizovat nová data s ostatními uzly datového pole.

3.3 Vzdálené volání procedur

Na rozdíl od předchozích dvou integračních technik je vzdálené volání procedur přednostně prostředek pro sdílení funkcionality. Změna v datech jedné aplikace si často vyžaduje vyvolání okamžitých akcí v ostatních aplikacích, což je u sdílených souborů problematické z hlediska zpoždění dané frekvencí exportu souboru a u sdílené databáze obtížné, protože poskytuje data, která nejsou nijak zapouzdřena, čímž může být, byť nechtěně, narušena integrita dat mezi aplikacemi (aplikace se neshodují v pohledu na správnost dat).

Vzdálené volání procedur aplikuje princip zapouzdření. Operace s daty druhých aplikací jsou realizovány přímým voláním přes definované rozhraní (*stubs*, *skeletons*), což umožňuje včas kontrolovat integritu dat. Obě aplikace navíc mohou vnitřně formátovat data nezávisle na sobě, sémantická neshoda dat je ošetřena poskytnutím patřičně definovaného rozhraní, které reprezentuje požadovanou sémantiku.



Obrázek 3.4: Vzdálené volání procedur (převzato z [6])

Problém vzdáleného volání procedur vyplývá ze sémantiky lokálního volání procedur a její odlišnosti od vzdáleného volání procedur. U lokálních volání musí volající i volaná metoda ležet ve stejném adresovém prostoru, obě musejí být napsány ve stejném jazyce, volající metoda musí předat daný počet parametrů daných datových typů, atd. U komunikace aplikací, které neleží v jednom adresovém prostoru, se však setkáváme s jinými jevy:

- Problém interoperability různých platforem (např. Java versus C++)
- Problém rychlosti — lokální volání je mnohem rychlejší než vzdálené ⇒ Má se čekat?
- Problém dostupnosti v případě výpadku sítě
- Problém bezpečnosti volání — naslouchání, podvržený příjemce volání

Pokud nejsou respektovány tyto uvedené odlišnosti vzdálené a lokální komunikace, můžeme vytvořit systém, který pracuje velmi pomalu a nespolehlivě.

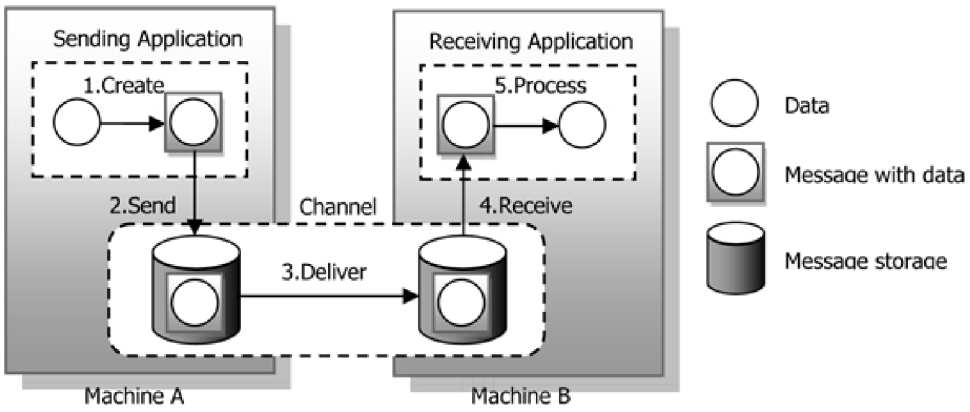
Zapouzdřením sémantiky vzdáleného volání procedur do sémantiky lokálního volání procedur navíc nijak neřeší problém vazeb v systému. Tak jako v jakékoliv jiné objektově-orientované aplikaci, objekty jsou vzájemně svázány prostřednictvím rozhraní, která poskytují. Při návrhu systému využívajícího vzdáleného volání procedur tedy stálé musíme dbát na minimalizaci vazeb vhodným návrhem rozhraní, v opačném případě je výsledkem těžko udržovatelný a málo škálovatelný systém.

3.4 Zasílání zpráv

Technika zasílání zpráv je přístup k implementaci middleware a prostředek integrace, kterým se budeme zabývat přednostně, neboť je předmětem návrhu systému v tomto projektu.

Aplikace komunikují vkládáním jednotek dat – zpráv – do kanálů, což jsou logické cesty propojující aplikace. Z hlediska komunikace hraje aplikace roli producenta nebo konzumenta. Producenti vkládají zprávy do kanálů a konzumenti zprávy z kanálů vybírají. Zprávy jsou datové struktury, které mohou být interpretovány jako data, příkaz nebo událost.

Komunikační vrstva založená na zasílání zpráv (*message-oriented middleware*) působí jako neutrální zóna mezi heterogenními systémy, která zajišťuje přenos aplikačních dat [2]. Komunikace je z podstaty asynchronní, což znamená, že producenti zpráv nečekají, až vzdálení konzumenti přijmou a zpracují zprávu. Jakmile producent předá zprávu vrstvě middleware, pokračuje ve své činnosti, a vrstva zajistí doručení zprávy na pozadí (*send-and-forget*). Systém je zodpovědný za doručení zprávy, i když vzdálená strana není dostupná. Zprávy jsou ukládány na straně producenta a v případě chybovosti přenosu opakovaně zasílány straně konzumenta, dokud nedojde k úspěšnému přenosu (*store-and-forward*). To zaručuje maximální možnou spolehlivost přenosu. Asynchronnost mj. umožňuje, aby konzumenti nemuseli aktivně čekat na příchozí zprávy, ale byli uvědomeni systémem pomocí mechanismu zpětného volání (*callback*).



Obrázek 3.5: Schéma komunikace zasláním zprávy (převzato z [6])

Asynchronní zasílání zpráv je v mnoha směrech výhodná technologie. Na rozdíl od sdílení souborů umožňuje vedle dat sdílet i funkcionalitu a odstraňuje problémy se stárnutím

informace, sdílená databáze zase představuje úzké hrdlo a vzdálené volání procedur způsobuje těsné vazby mezi aplikacemi a nereflektuje problémy vzdálené komunikace.

Podobně jako sdílení souborů uvolňuje vazby mezi aplikacemi — zprávy mohou být libovolně transformovány, směrovány i monitorovány; producentská aplikace neví, co se stane se zprávou v momentě, kdy ji umístní do kanálu, neví tedy ani, kterými cestami zpráva projde, jak bude transformována a dokonce ani která aplikace ji zkonzumuje. V případě transformací zpráv pak sice nastává nejednoznačnost v sémantice dat, přístup principu zasílání zpráv je však takový, že se nesnaží navrhnout systém tak, aby se tomuto problému vyvaroval do budoucna (jako v případě sdílené databáze), ale snaží se přímo na tento problém poukázat a donutit vývojáře jej včas řešit.

Komunikace prostřednictvím zasílání zpráv s sebou přináší i některá omezení. Jedná se především o složitost celého modelu vývoje a testování integrovaného prostředí, která je zapříčiněna asynchronností komunikace. Programátor musí počítat s velkým počtem paralelních operací. Ty mohou být v reakci na příjem zprávy vyvolávány v těžko určitelných okamžicích i během vykonávání jiných operací a aplikace musejí s těmito jevy počítat. Paralelismus v takovém prostředí tedy znesnadňuje testování a odstraňování chyb v systému.

V knize [6] jsou prezentovány vzory, které jsou používány pro návrh řešení integrace aplikací pomocí zasílání zpráv. Tato kapitola nastíní význam některých těchto vzorů a jejich vlastností (podvzory), uvedeny však budou pouze ty nejzákladnější, které použijeme při vývoji komponenty, která je předmětem tohoto projektu.

3.4.1 Zpráva

Zpráva je atomická jednotka, zapouzdřující data putující kanálem. Obsahuje informaci producenta, která musí být přenesena z jeho adresového prostoru do adresového prostoru konzumenta. Proces zabalení informace do zprávy se nazývá *marshaling*, opačný proces rozbalení zase *unmarshaling*.

Zprávy jsou specifické datové struktury. Obvykle obsahují hlavičku s metainformacemi o zprávě, užitečnými pro komunikační vrstvu, a tělo zprávy s aplikačními daty. Tento koncept je analogický paketové komunikaci např. v sítích TCP/IP.

Z pohledu programátora aplikací existují rozdílné typy zpráv:

- Příkazová zpráva (*command message*) slouží k vyvolání určité akce (služby) ve vzdálené aplikaci.
- Dokumentová zpráva (*document message*) je pouhým nositelem dat
- Událostní zpráva (*event message*) slouží k oznámení změny v aplikaci.

Často bývá nutné, aby konzument zprávy potvrdil její příjem a zpracování, případně vrátil výsledek operace. Obvykle dokumentová zpráva potvrzuje zprávu příkazovou a obsahuje identifikátor, který určuje kontext potvrzovaného příkazu.

3.4.2 Kanál zpráv

Kanály zpráv řeší otázku, jak jsou zprávy doručovány vzdáleným konzumentům. Kanál je virtuální cesta mezi producentskou a konzumentskou aplikací, která slouží jako prostředek pro komunikaci určitého typu informace (potažmo zprávy). Producent sice neví, která aplikace na druhém konci kanálu zprávu zkonzumuje, může si být ale jista, že je doručena

aplikaci, která má o tuto zprávu zájem. To platí za předpokladu, že zpráva obsahuje informaci kompatibilní s kanálem, do kterého je vložena.

Implementace kanálů se liší systém od systému. Mohou být realizovány přímými síťovými spojeními mezi koncovými body nebo sdružovány přes centrální systém. Jsou dány logickou adresou, jejíž formát je rovněž závislý na implementaci middleware.

Ustavení infrastruktury kanálů je hlavní náplní v projektu integrace. Vždy musí být někdo, kdo kanály staticky definuje tak, aby byly přístupné aplikacím produkující zprávy. Aplikace zažádají o přístup ke kanálu na základě jeho logické adresy a poté mohou vkládat zprávy.

Existují dva základní typy kanálů. *Point-to-point* kanál doručuje zprávu jednoho producenta právě jednomu konzumentovi, kdežto *publish-subscribe* kanál doručí zprávu jednoho producenta více než jednomu konzumentovi, každý konzument pak obdrží totožnou kopii zprávy. Komponenta, která je předmětem tohoto projektu, pracuje výhradně s point-to-point kanály.

3.4.3 Koncový bod

Aplikace a middleware jsou v podstatě dvě oddělené softwarové entity. Aby mohla aplikace posílat a přijímat zprávy, musí mít dispozici rozhraní pro přístup k middleware, obecněji řečenou jakousi vrstvu, která zajistí propojení aplikační domény a domény vrstvy middleware.

Koncový bod je komponenta klientské aplikace, která jedním směrem zajišťuje tvorbu zpráv z aplikačních dat a jejich vkládání do kanálů a opačným směrem extrakci dat ze zpráv a jejich předávání aplikaci. Obvykle k tomu používá aplikační rozhraní middleware. Aplikace často používají více koncových bodů pro přístup k více kanálům.

Posílání zpráv má jednoduchou sémantiku, příjem zpráv se však často liší. Existuje více přístupů k doručení zprávy konzumentovi: konzument může žádat middleware vrstvu v pravidelných intervalech (*polling consumer*) o zprávy nebo naopak samotný systém předává příchozí zprávy automaticky pomocí mechanismu zpětného volání (*event-driven consumer*).

3.4.4 Sběrnice zpráv

Vzor sběrnice zpráv (*message bus*) představuje styl propojení aplikačních systémů v podniku pomocí zasílání zpráv, podobně jako v hardwarové sběrnici (např. PCI), propojující jednotlivé hardwarové komponenty. Sběrnice přenáší data určité aplikační domény, komunikace je však na této úrovni určitým způsob jednotná. Jsou dány:

- Datové entity, přenášené prostřednictvím dokumentových zpráv
- Množina příkazů, přenášené prostřednictvím příkazových zpráv
- Infrastruktura kanálů, které přenášejí vždy pouze jeden určitý typ informace nebo příkazů

Pokud chce aplikace komunikovat, připojí se k této sběrnici prostřednictvím koncového bodu. Poté má k dispozici sadu dokumentových a příkazových zpráv a sadu kanálů, které může využívat.

Jako příklad aplikace sběrnice zpráv lze uvést prostředí aplikace SafeQ (viz kap. 4). Jednotlivá multifunkční kancelářská zařízení od různých výrobců se napojují na jednotnou sběrnici zpráv, prostřednictvím které si vyměňují zprávy s tiskovými servery. Příkladem

může být např. příkazová zpráva `Print` s údaji o tisku úlohy, nebo dokumentová zpráva `PrintJob` nesoucí data tiskové úlohy. Takto lze pomocí sběrnice zpráv vybudovat prostředí s architekturou orientovanou na služby (viz kapitola 2.1.3).

3.5 Úvod do XML

XML (*Extensible Markup Language*) je standardní předpis pro vytváření tzv. *značkovacích jazyků*. Definuje generickou syntaxi pro vkládání jednoduchých značek do uživatelských dat v textové podobě, čímž umožňuje tato data účelově strukturovat. Tato vlastnost činí takto značkované dokumenty ideálním nástrojem pro přenos a uchování informace vhodným pro automatizované zpracování.

Uplatnění jazyka XML dnes leží ve dvou hlavních oblastech. Za prvé slouží jako prostředek pro uchování informace, potažmo její prezentací koncovému uživateli (webové dokumenty, články, knihy atd.). Úspěch v této oblasti vyplývá z úspěchu jazyka HTML pro popis webových dokumentů. Druhá oblast, pro použití v této práci důležitější, těží se schopnosti XML zakódovat informaci pro přenos z jedné aplikace do druhé. Proto XML nalezl široké použití právě v oblasti integrací aplikací ať už se jedná o prostý export a import XML souboru, XML databázi (Apache Xindice[2]), technologie pro vzdálené volání metod (XML-RPC[3]) či právě zasílání zpráv (již zmíněné SOAP a Web Services).

Základní stavební jednotkou XML dokumentu je *prvek*. Prvky se skládají z uživatelské informace (*obsahu prvku*) obklopené startovací a ukončovací značkou (*tagem*) nesoucí název prvku. Prvky mohou být dále doplněny přídavnou informací, zasazenou do počáteční značky, což je hodnota tzv. *atributu*. Specifikace striktně určuje syntaxi značek a atributů, je však na vývojáři aplikace, aby určil množinu prvků a atributů, která bude systémem využívána, tedy vytvořil XML aplikaci [5]. Příkladem aplikací XML je nástupce HTML XHTML[4], RSS[5] — rodina jazyků pro čtení novinek na webových stránkách, nebo jazyk SVG[6] pro popis dvourozměrné vektorové grafiky.

3.5.1 Struktura XML dokumentu

XML formátuje dokument do stromové datové struktury. Prvky lze do sebe libovolně vnořovat při zachování přípustného stylu strukturování: nesmí dojít k jejich vzájemnému překrytí (*well-formedness*) a musí existovat právě jeden kořenový prvek. Vzhled prvků a atributů lze nejlépe ilustrovat na příkladě následujícího XML dokumentu:

```
<adresa typ="panelák" rokNastěhování="1990">
   <jméno>Jan</jméno>
   <příjmení>Martinák</příjmení>
   <ulice>Na Valtické 36</ulice>
   <město>Břeclav</město>
   <psč>69141</psč>
</adresa>
```

[2] http://xml.apache.org/xindice
[3] http://www.xmlrpc.com
[4] Extensible HyperText Markup Language - http://www.w3.org/TR/2001/REC-xhtml11-20010531
[5] Really Simple Syndication - http://www.rssboard.org/rss-specification
[6] Scalable Vector Graphics - http://www.w3.org/Graphics/SVG

Zde má kořenový prvek `adresa` právě dva atributy `typ` a `rokNastěhování`, jejichž hodnoty musí být ohraničeny uvozovkami. Prvek `adresa` obsahuje prvky `jméno`, `příjmení`, `ulice`, `město` a `psč`, jejichž obsah vyjadřuje uživatelskou informaci o adrese autora této práce.

Aby bylo možné podpořit rozvoj určité XML aplikace, je vhodné mít nástroj, který určí, zda daný XML dokument vyhovuje požadavkům systému, který tuto aplikaci využívá. Takovým nástrojem jsou *schémata*. Schémata formalizují omezení na strukturu dokumentu, přípustné prvky a atributy, jejich datové typy, aj. Dokument, který splňuje tato omezení, se nazývá *validní*. V dnešní době existuje vícero způsobů zápisu schémat, z nichž nejvýznamnější jsou jazyky DTD[7] a XSD[8]. První z uvedených sloužil už před nástupem XML k popisu dokumentů jazyka SGML[9]. DTD však má jen omezené možnosti — popisuje, kde v dokumentu se mohou které prvky, atributy a entity vyskytovat. XSD naproti tomu rozšířilo možnosti DTD tím, že umožňuje specifikovat datové typy prvků (např. číslo, řetězec) a další omezení na hodnoty obsahu prvku.

Nutno dodat, že pro XML aplikaci není nutné definovat patřičné schéma. Primární význam leží v jejich použití pro formální definici. Výhoda formalizované definice spočívá v tom, že je jednoznačná a znemožňuje různé interpretace, na rozdíl od definice popsané v přirozeném jazyce. Všichni, kdo chtějí dokumenty XML odpovídající danému schématu zpracovávat, proto vědí, jak mohou dokumenty vypadat [8].

XML disponuje ještě jedním mocným nástrojem, a tím jsou jmenné prostory. Je to mechanismus, který umožňuje v jednom dokumentu XML kombinovat více sad značek, které mohou jinak působit problém kvůli shodným názvům [8]. Jelikož mají ve vztahu k této práci spíše okrajový význam, odkazujeme na literaturu [5], která se jimi zaobírá podrobněji.

3.5.2 Přenos informace pomocí XML

V úvodu této kapitoly byly nastíněny možnosti použití jazyka XML pro přenos dat mezi aplikacemi. Důvodů, proč je tento přístup výhodný, je několik [2]:

- XML z principu umožňuje jednoduše modelovat komplexní hierarchické datové struktury.
- XML elementy a atributy svým názvem popisují data, která obsahují. Těmito informacemi se řídí *parsery* při zpracování dokumentu.
- XML prezentuje data jako řetězec znaků, čímž usnadňuje výměnu dat mezi heterogenními platformami.
- XML zavádí striktní syntaxi dokumentů. Algoritmy, které je zpracovávají, jsou relativně jednoduché a rychlé.
- XML aplikace mohou být inkrementálně doplňovány o nové prvky a atributy při zachování kompatibility se staršími verzemi. Parsery zpracují pouze tu část dokumentu, které rozumí.

[7] Document Type Definition - součást XML původem z SGML - http://www.w3.org/TR/REC-xml

[8] XML Schema - http://www.w3.org/TR/xmlschema-1

[9] Standard Generalized Markup Language - předchůdce jazyka XML, jednou z úspěšných aplikací SGML je jazyk HTML pro tvorbu webových dokumentů. SGML je popsán standardem ISO 8879:1986

Termín *parser* uvedený výše označuje algoritmus využívaný aplikací pracující s XML dokumenty, který zpracovává vstupní dokument a rozděluje jej na prvky, atributy a samotná aplikační data kousek po kousku. V závislosti na názvech extrahovaných prvků a atributů manipuluje s objekty resp. procedurami aplikační logiky (např. vytvoří kolekci a naplní jej položkami z dokumentu). Většina parserů zároveň kontroluje správnost syntaxe dokumentu (well-formedness) a pokud narazí na chybu, přeruší zpracovávání s chybou. Některé parsery jsou také schopny kontrolovat validitu dokumentu podle zadaného schématu, ovšem za cenu snížení výkonu zpracovávání. Více o různých typech parserů, jejich možnostech, výkonnosti a aplikačním rozhraní v jazyce Java v literatuře [9].

Dle specifikace mohou XML dokumenty obsahovat jakýkoliv znak ze znakové sady Unicode, která dokáže obsáhnout naprostou většinu světových abeced. Tyto znaky je možné zakódovat pomocí některého z mnoha vícebajtových kódování (např. UTF-8, UCS-2). Stejně tak je však dovoleno používat i starší znakové sady, jako např. četné nadmnožiny ASCII (ISO 8859-2, Windows-1250 atd.) zakódované na 8 bitech. Parsery se proto snaží zjistit kódování podle několika prvních bytů v dokumentu [5], jeho snaha však končí neúspěchem, pokud je dokument kódován pomocí některé z nadmnožiny ASCII mimo UTF-8. V takovém případě je nutné přidat do dokumentu tzv. XML deklaraci, která dá parseru informaci o použitém kódování:

```
<?xml version="1.0" encoding="iso-8859-2"?>
```

Výše popsaný postup zajišťuje, že bude XML dokument čitelný na všech platformách, které zvládají práci se standardními znakovými sadami a kódováními, proto je XML obzvlášť vhodným nositelem informace mezi heterogenními platformami.

V rámci této práce je XML aplikováno na definici formátu zprávy, která je přenášena mezi systémy, které middleware propojuje. Více o implementaci této XML aplikace v kapitole 7.3

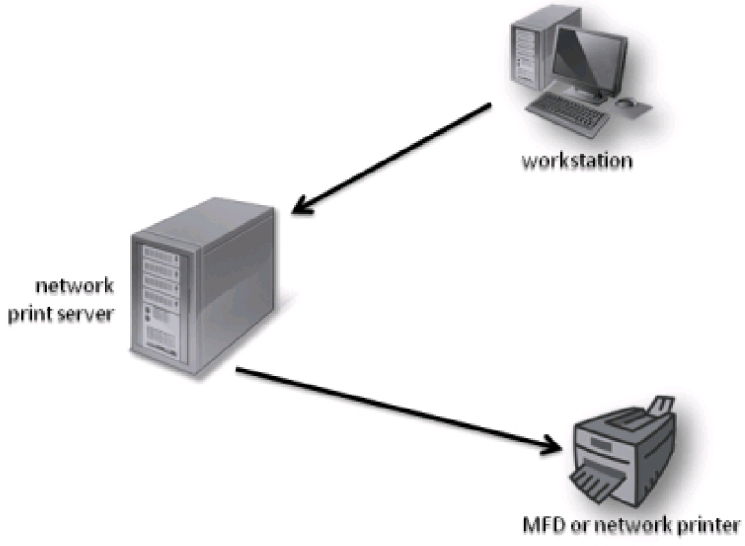
Kapitola 4

Studie cílového prostředí

Tato kapitola má za úkol uvést čtenáře do kontextu řešeného problému a seznámit ho s prostředím, pro které je vyvíjená softwarová komponenta určena. Bude zde pojednáno nejen o použitých softwarových i hardwarových entitách a o roli uživatele v tomto prostředí, ale především o příčinách a okolnostech, které podnítily vývoj této komponenty, a cílech, které má za úkol splnit.

4.1 SafeQ

Distribuovaná servisně-orientovaná aplikace SafeQ slouží ke konsolidaci správy tisku převážně ve větších a středních organizacích (firmy, školy, vládní instituce), jejichž cílem je usnadnit přístup zaměstnanců ke službám multifunkčních kancelářských zařízení (tisk, skenování, kopírování, faxování) a zároveň kontrolovat a optimalizovat spotřebu fyzických prostředku jako papír či toner.



Obrázek 4.1: Prostředí síťového tisku prostředníctvím protokolu LPR

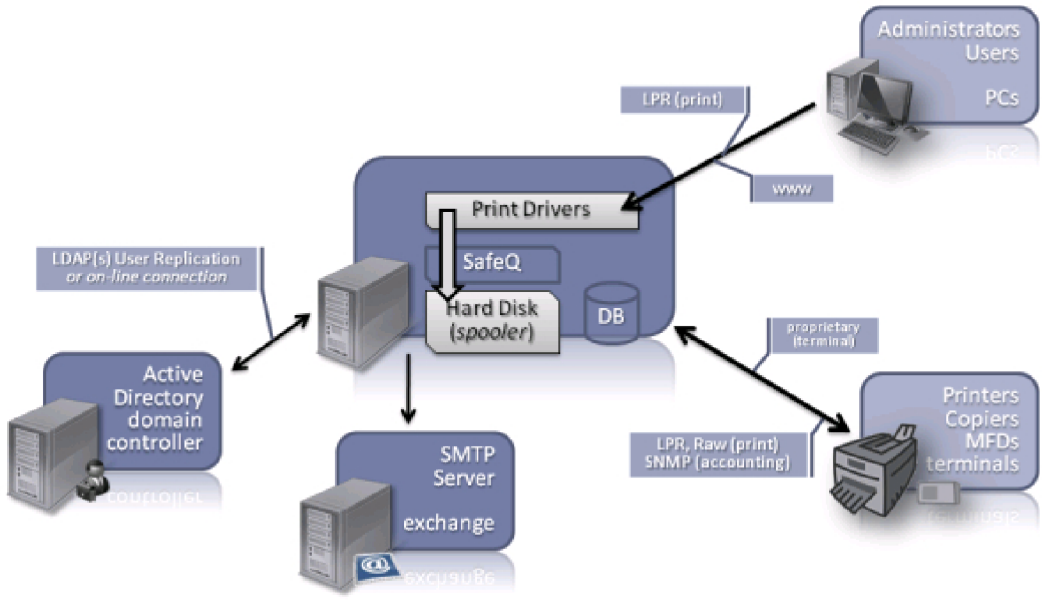
Služby v SOA SafeQ jsou poskytovány převážně multifunkčními kancelářskými zařízeními (MFD). V rámci této architektury jsou implementovány byznys procesy týkající

se užívání MFD a správy tiskové agendy v podniku. SafeQ výrazně zjednodušuje práci s MFD, neboť pro všechny podporované značky zajišťuje jednotné uživatelské rozhraní ovládané dotykovým panelem fyzicky připojeného externího terminálu (viz obr. 5.1), popř. vestavěného terminálu běžícího v MFD.

Hodnota SafeQ spočívá v rozšíření možností klasického síťového tisku prostřednictvím LPR protokolu[1] v unixových systémech (viz obr. 4.1), kdy uživatelské stanice (*workstations*) odesílají tiskové úlohy na LPR server (*network print server*), kde jsou následně řazeny do front pro tisk na jednotlivých připojených síťových tiskárnách (*MFD or network printer*).

SafeQ rozšiřuje funkcionalitu tohoto systému v několika hlavních bodech:

- Velmi přesné účtování spotřebovaného tiskového materiálu na základě výpočtu množství spotřebovaného papíru a jeho procentuálního pokrytí tonerem.
- Evidence nákladů s ohledem na zadanou cenu materiálu vůči jednotlivým uživatelům, skupinám uživatelů či projektům.
- Možnost účtování kopií, faxů i skenů
- Zabezpečený odložený tisk s využitím externích terminálů
- Uživatelský a administrátorský přístup přes webové rozhraní



Obrázek 4.2: Prostředí SafeQ

Schéma architektury SafeQ, znázorněné na obrázku 4.2, vyznačuje komunikační infrastrukturu jednotlivých prvků v systému. Součástí SafeQ serveru jsou sdílené ovladače pro komunikaci s MFD a interní databáze uživatelů (PostgreSQL), kterou lze synchronizovat s Active Directory.

Jako příklad si uvedeme odložený tisk úlohy, který dovoluje zabezpečit proces tisku důvěrných dokumentů tím, že uživatel požádá o tisk až u tiskárny prostřednictvím externě

[1] RFC 1179

napojeného terminálu. Uživatelé používají specializovaný tiskový ovladač k odeslání tiskové úlohy ze své pracovní stanice ve formátu jazyka tiskárny (PCL, Postscript, ...) na SafeQ server prostřednictvím LPR protokolu. Zde jsou data tiskové úlohy ukládána do integrovaného řadiče tiskových úloh (*spooler*) ve formě souborů. Úloha čeká ve spooleru do doby, než se uživatel autentizuje na externím terminálu (např. PINem nebo čipovou kartou) a zažádá o její vytištění. Uživatel může rovněž zažádat o vytištění přes webové rozhraní SafeQ.

4.2 Multifunkční zařízení

Multifunkční zařízení (MFD) jsou kancelářské stroje, které v sobě integrují funkce tiskárny, skeneru, kopírky a faxu. Různé typy pokrývají různé segmenty trhu v závislosti na funkcích, které poskytují, vesměs jsou to však nákladná zařízení s objemnými zásobníky na papír, vybavená mechanismy, které umožňují plně automatizovat proces zpracování dokumentů různých formátů. Tím jsou určena spíše pro organizace, firmy a školy, než pro domácí použití (viz obr. 4.3).

Obrázek 4.3: HP Color LaserJet CM4730mfp (převzato z www.hp.com)

Moderní MFD posunula možnosti automatizovaného zpracování a správy dokumentové agendy uživatele ještě o krok dál. Stala se komplexními vestavěnými systémy obsahující klasický počítačový hardware jako procesor, paměť a pevný disk, na kterém běží síťový operační systém (např. Linux). To umožňuje výrobcům vyvíjet aplikační rozhraní k funkcím MFD, která jsou dále podporována vývojovými nástroji (SDK) pro implementaci aplikací, které kompletně řídí celý proces zpracování dokumentu, případně jeho část (např. rozpoznání znaků v naskenovaném dokumentu, vlastní způsob příjmu tiskové úlohy, předzpracování a kompletace atd.).

Způsoby realizace vrstvy MFD API se liší od výrobce, prakticky se však dělí na dvě kategorie:

Serverové aplikace zpracovávají požadavky na provedení funkce MFD od vzdálené aplikace. Pomocí vývojových nástrojů je možné implementovat aplikaci, která komunikuje s MFD, např. prostřednictvím SOAP zpráv (Konica Minolta OpenAPI).

Interní aplikace jsou plnohodnotné aplikace, které běží rovněž přímo na MFD, např. v rámci virtuálního stroje Javy. Standardní rozhraní Java ME je pak výrobcem zařízení doplněno knihovnami pro ovládání funkcí přístroje popř. tvorbu grafického uživatelského rozhraní s použitím integrovaného LCD panelu. Tohle je případ architektury, kterou zvolila pro svá MFD právě firma Ricoh, která zakládá knihovnu pro tisk na standardním Java rozhraní tisku[2].

4.2.1 Ricoh Embedded Software Architecture

Přestože komunikátor navrhujeme obecně pro jakýkoliv počítačový systém, primárně je plánován provoz na aplikační platformě *Embedded Software Architecture* multifunkčních zařízení Ricoh Aficio MP. Tato platforma je vybudována nad virtuálním strojem jazyka Java ME v konfiguraci *Connected Device Configuration* (CDC) verze 1.1.2 [15].

Zmíněný typ konfigurace je obecně zaměřen na zařízení s vyšším výpočetním výkonem a síťovým rozhraním, jakými jsou právě MFD. Cílem je poskytnout aplikační rozhraní a nástroje podobné platformě Java SE, aby vývojáři znalí této platformy dokázali aplikovat své znalosti i v prostředí vestavěných systémů. CDC bývá rozšířeno třemi typy profilů, které staví jeden na druhém (*Personal Basis Profile* staví na *Foundation Profile* a *Personal Profile* staví na Personal Basis Profile). Ricoh Embedded Software Architecture splňuje profil Foundation Profile, což spolu s charakteristikou CDC dává optimalizovanou implementaci aplikačního rozhraní odpovídající Java SE 1.4.2, mimo podporu grafických uživatelských rozhraní AWT a Swing[3].

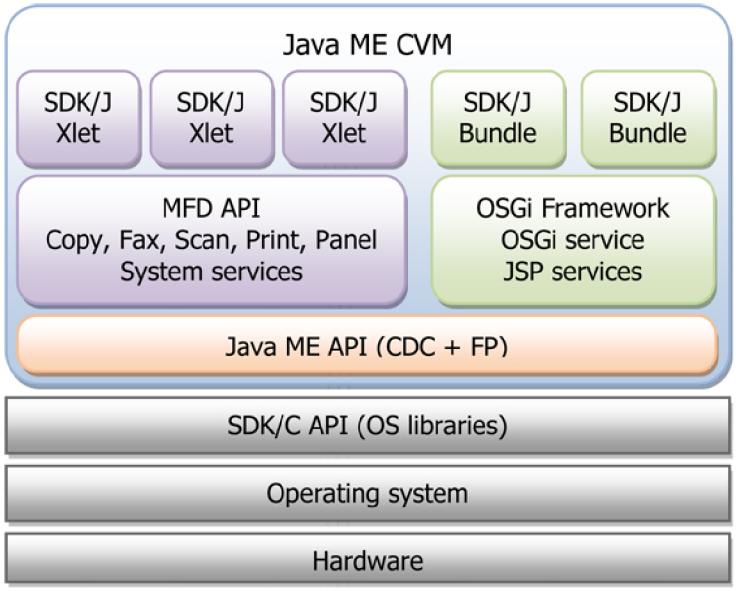
Aplikace, běžící na této platformě (tzv. SDK/J aplikace), využívají standardních funkcí kancelářského zařízení (kopírování, skenování, tisk, fax) a aplikačního rozhraní virtuálního stroje Javy.

Jak je vidět na obrázku 4.4, SDK/J aplikace mohou být dvojího typu. *Xlety* jsou aplikace, jejichž model životního cyklu se významně liší od klasických aplikací v Javě SE. Přestože jsou specifikovány v rámci Personal Basic Profile, firma Ricoh adoptovala tento model pro svá zařízení a vyvinula vlastní implementaci grafického uživatelského rozhraní, založenou na AWT, která poskytuje prvky (např. tlačítka, modální okna) a obsluhu událostí nad těmito prvky pro integrovaný dotykový displej. Xlety jsou podobné webovým appletům, mají definovaný životní cyklus, který spravuje manažer Xletů. V rámci virtuálního stroje může běžet více xletů, které mohou být operátorem zařízení libovolně startovány, pozastavovány a ukončovány. Stavový diagram znázorňující životní cyklus xletů je na obrázku 4.5.

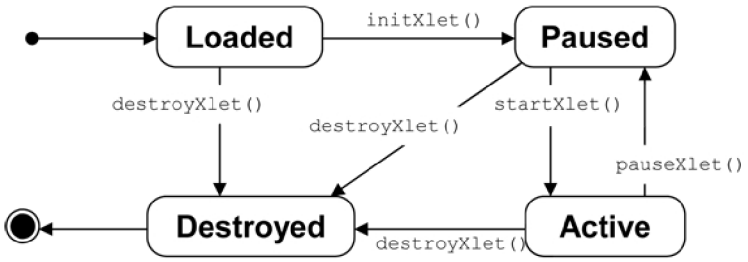
Kromě grafického uživatelského rozhraní mohou xlety pracovat s funkcionalitou samotného zařízení, provádět tisk, skenování, fax atd. Toto rozhraní je vyznačeno na obrázku 4.4 jako MFD API.

[2] Java Print Service - http://java.sun.com/j2se/1.4.2/docs/guide/jps/index.html

[3] Knihovna prvků GUI Swing je vyspělejší náhrada za dřívější Abstract Window Toolkit. Více viz http://java.sun.com/products/jfc/tsc/articles/architecture



Obrázek 4.4: Ricoh Embedded Software Architecture



Obrázek 4.5: Životní cyklus xletu

Druhým typem aplikací, běžících na MFD Ricoh, jsou *serverové aplikace*. Tyto aplikace běží v rámci frameworku OSGi jako tzv. *bundles*. Tento framework spravuje životní cykly bundle aplikací podobně jako manažer xletů spravuje xlety[4]. Účel bundle aplikací je poskytovat funkcionalitu serverových webových aplikací, komunikujících se vzdálenými klienty prostřednictvím webového protokolu HTTP (tzv. *servletů*). OSGi dále nabízí podporu rozhraní JSP (Java Server Pages) pro snadnější tvorbu dynamických webových aplikací. Bundly mohou v rámci virtuálního stroje komunikovat s xlety přes speciální rozhraní (`ComManager`), jejich přístup k MFD API a síťovému rozhraní je proto mnohem omezenější.

[4]Více o architektuře OSGi na stránkách OSGi Alliance http://www.osgi.org/About/FAQ

Kapitola 5

Definice problému

V kapitole 4.1, zabývající se aplikací SafeQ, bylo zmíněno použití externích terminálů, fyzicky napojených na multifunkční zařízení (viz obr. 5.1). Uživatelé ovládají MFD skrze terminál, který komunikuje se SafeQ serverem pomocí proprietárního protokolu. Úkolem terminálů je především:

- Poskytnout jednotné vícejazyčné uživatelské rozhraní pro přístup k MFD
- Autorizovat uživatele pro přístup k funkcím MFD. Uživatelé se mohou identifikovat PINem nebo čipovou kartou
- Umožnit zabezpečený odložený tisk (volba tisku určité úlohy až po autorizaci)
- Komunikovat se SafeQ serverem za účelem zaúčtování tisku, skenu, faxu či kopií

Obrázek 5.1: SafeQ Terminal Proffesional (převzato z [19])

Je zřejmé, že terminály jsou nezbytnou komponentou v aplikaci SafeQ. Je však velice důležité si uvědomit, že některé typy MFD teoreticky externí terminál nepotřebují, neboť jeho funkcionalitu lze efektivně implementovat v samotném zařízení pomocí vývojových

nástrojů poskytovaných přímo výrobcem (viz kap 4.2.1), ať už se jedná o grafické uživatelské rozhraní ovládané dotykovým displejem či možnosti autentizace.

Toto zjištění je hlavní motivací k vývoji komponenty pro komunikaci se serverem SafeQ, zde nazývané *komunikátor*. Tato komponenta se stane součástí generace vestavěných terminálů, běžících přímo v MFD. Zákazníci si již nebudou muset kupovat drahé externí terminály, místo toho jim bude nabídnuto toto levnější a spolehlivější řešení, šité na míru přímo jejich MFD. Na druhou stranu vývoj vestavěných terminálů klade nároky na flexibilitu a možnosti aplikačního rozhraní hostitelského MFD. V současné době proto z podporovaných MFD připadá v úvahu pouze Ricoh Embedded Software Architecture.

Životní cyklus vývoje komunikátoru se snaží kopírovat některé postupy unifikovaného procesu vývoje aplikací nastíněné v knize [1], především co se týče aplikace modelovacího jazyka UML. Cílem je vytvořit produkt, který odpovídá požadavkům kladeným na funkcionalitu a nefunkcionální omezení. Celý proces startuje neformální specifikací požadavků, pokračuje jejich analýzou a návrhem architektury. Ve fázi implementace je vytvořen spustitelný produkt, který je průběžně testován pomocí jednotkových testů. Nakonec je produkt nasazen do cílového prostředí.

Přestože zmíněný proces vývoje počítá s iteracemi, zde použitý model spíše odpovídá modelu vodopád s možnosti návratu na předchozí stupeň [10], který je vzhledem k rozsahu prací plně dostačující.

5.1 Neformální specifikace

Předmětem práce je komponenta, která realizuje aktivní prvek distribuované SOA infrastruktury (tzv. *komunikátor*). Tento prvek má za úkol poskytovat klientským aplikacím funkce middleware pro komunikaci s ostatními aplikacemi prostřednictvím výměny zpráv. Cílem je získat uniformní, škálovatelné rozhraní pro asynchronní komunikaci aplikací běžících převážně na vestavěných systémech. Přínosem bude usnadnění integrace nových zařízení do stávající infrastruktury.

Dále bylo požadováno zvážit všechny možnosti konfigurace této komponenty a implementovat nejvhodnější variantu pro danou platformu. Počítá se také s návrhem aplikace, jež má demonstrovat praktické využití komunikátoru pro ovládání daného vestavěného zařízení prostřednictvím komunikace s ostatními aplikacemi napojenými na tuto vrstvu. Tato aplikace poté v budoucnu poslouží jako základ ke komplexnějším systémům pro ovládání zařízení (vestavěný terminál).

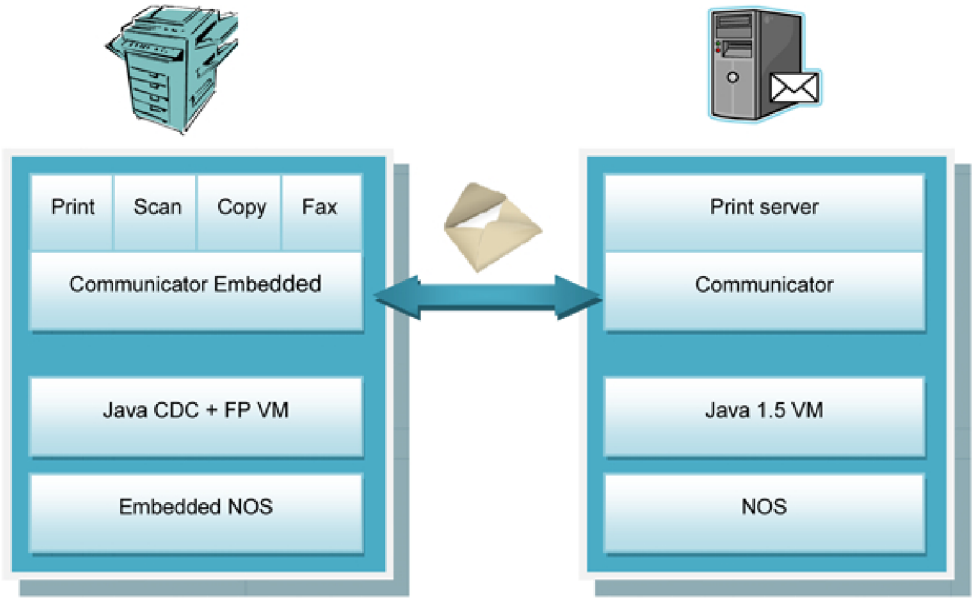
Vedle funkcionality byly mnohem větší nároky kladeny na nefunkcionální požadavky. Jejich splnění významně ovlivní návrh systému. Nefunkcionální požadavky rozdělujeme na dva typy: jedny jsou obecné požadavky, nezávislé na cílové platformě, které z větší části plynou z principů zasílání zpráv, druhé (tzv. požadavky externího rozhraní) berou obecně v potaz nároky na rozhraní vůči systémům třetích stran. V tomto případě budou předmětem těchto požadavků omezení architektury multifunkčního zařízení od firmy Ricoh, na kterém bude komponenta vyvíjena a testována. Zdrojem těchto požadavků je především manuál k multifunkčnímu zařízení. Kompletní výčet, který je součástí dokumentace k projektu, viz přílohy A, B a C.

Nejvýznamnějším nefunkcionálním požadavkem je asynchronnost komunikace. Ovlivňuje významně návrh aplikace, hlavně co se týče správy síťových spojení. Aplikace využívající komunikátor nesmí být blokovány po dobu výměny zpráv. Po předání zprávy komunikátoru musí aplikace pokračovat ve své činnosti. V opačném směru musí být na příchod zprávy zpětně upozorněna komunikátorem.

Jeden z problémů, pramenících z asynchronnosti komunikace, je způsob ošetření stavu, kdy zpráva nemůže být doručena (např. kvůli chybě spojení). Mechanismus zpětné notifikace chybového stavu klade nároky no logiku klientské aplikace, jeho implementace proto není zatím požadována. V každém případě musí být tato událost zaznamenána, aby správce systému mohl podniknout nutná nápravná opatření.

Co se týče implementace protokolu, bylo požadováno navrhnout vhodný formát pro přenos dat mezi komunikujícími systémy, který plně vyhovuje požadavkům na výměnu libovolného typu dat a který pokud možno odpovídá moderním komunikačním standardům.

Jeden z okrajových požadavků je šifrování komunikace. Vzhledem k omezení na verzi virtuálního stroje Javy v multifunkčním zařízení bude nutné použít rozšíření Java Secure Socket Extension, které je ve verzi Javy 1.4.2 již standardní součástí [18].



Obrázek 5.2: Schéma příkladu použití komunikátoru

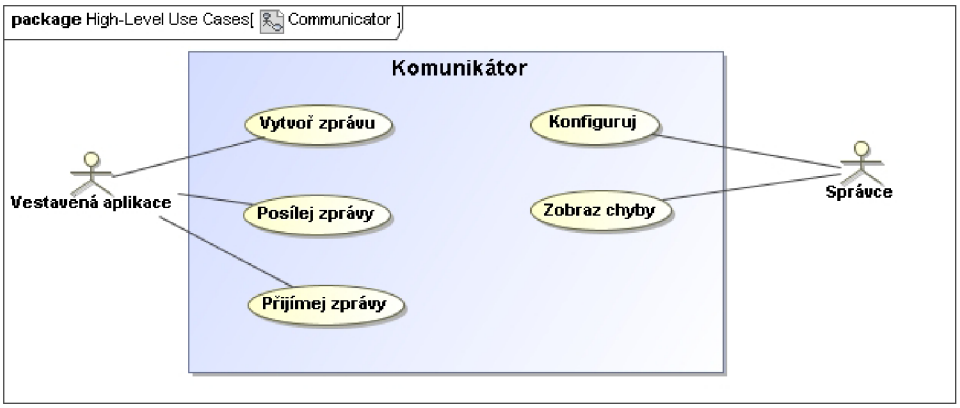
Obrázek 5.2 znázorňuje nasazení komunikátoru v cílovém prostředí, jakým je již zmiňované SafeQ. Upřesňuje vizualizaci rozhraní z obr. 4.2 mezi tiskovým serverem a MFD. Komunikátor běží ve virtuálním stroji na MFD a poskytuje middleware služby aplikacím pro tisk, skenování, kopírování a faxování, integrované ve vestavěném terminálu multifunkčního zařízení. Tyto aplikace komunikují se vzdáleným komunikátorem na straně tiskového serveru pomocí zpráv, které mohou obecně zapouzdřovat jakékoliv aplikačně-závislé příkazy. Zde to bude např. zjištění stavu zařízení, spuštění tisku, autentizační proces atd. Zprávy však také mohou sloužit k replikaci dat, neboť bude nutné sdílet data tiskových úloh (např. popis stránky v jazyce tiskárny).

V cílovém prostředí jakým je SafeQ nepotřebujeme, aby mezi sebou komunikovaly dvě tiskárny, prezentované řešení tedy není typickým integračním problémem. Smyslem je zajistit jednotnou komunikaci serveru s multifunkčními zařízeními a minimalizovat úsilí k integraci nových zařízení do stávající architektury. Pokud využijeme prostředků a principů nastíněných v teoretickém úvodu práce, nejen že usnadníme proces integrace nových aplikací, umožníme ale také jejich jednodušší údržbu a vývoj nových verzí.

Kapitola 6

Analýza požadavků

Diagram případů použití na obr. 6.1 shrnuje nejzákladnější funkcionalitu komunikátoru. Hlavními aktéry jsou aplikace běžící na vestavěných zařízeních. Aplikace musí mít možnost vytvářet zprávy a plnit je užitečnými daty (*Vytvoř zprávu*). K tomu je potřeba definovat mechanismy pro generování zpráv a formát zpráv. Komunikátor musí být dále schopen zasílat zprávy lokálních aplikací (*Posílej zprávy*) a přijímat zprávy vzdálených aplikací (*Přijímej zprávy*).



Obrázek 6.1: Diagram případů použití komunikátoru

Dalším aktérem je správce, který je zodpovědný za dodání informací potřebných pro funkci infrastruktury pro doručování zpráv (*Konfiguruj*). Zároveň musí mít možnost zobrazit informaci o chybových stavech komponenty, potažmo celé infrastruktury (*Zobraz chyby*).

Jádro funkcionality komunikátoru tvoří výměna zpráv. Než však rozebereme příslušné případy použití, je nutné podat vysvětlení k terminologii použité v prostředí komunikátoru:

- *Relace* (*sessions*) — Objekty, které vymezují přístup aplikace ke komunikátoru. Zprávy jsou zasílány a přijímány skrze relace.
- *Kanál* (*channel*) — Pojmenovaná virtuální cesta mezi dvěma aplikacemi (viz 3.4.2).
- *GUID* (*Globally Unique Identifier*) — Řetězec znaků a čísel, který jednoznačně určuje zařízení v síti (např. MFD).

Název	Posílej zprávy
Aktéři	Vestavěná aplikace
Popis	Producentská aplikace posílá zprávy vzdálené konzumentské aplikaci skrze relaci, ustavenou nad určitým virtuálním kanálem. Název kanálu určuje cílovou aplikaci, identifikátor GUID určuje adresované zařízení v síti.
Počáteční podmínky	Aplikace má zprávy k odeslání
Koncové podmínky	Aplikace odeslala zprávy
Hlavní tok	**Zaslání zprávy** 1. Případ použití se spustí, když chce aplikace odeslat zprávy. 2. Aplikace požádá komunikátor o vytvoření relace. 3. Aplikace předá objekty zprávy komunikátoru skrze relaci. 4. Aplikace ukončí relaci. 5. Aplikace pokračuje ve vykonávání svého kódu. 6. ***Pro všechny předané zprávy***: 6.1 Komunikátor rozhodne o cílové destinaci zprávy a zařadí ji do patřičné fronty 6.2 Komunikátor serializuje zprávu do proudu XML dat. 6.3 Komunikátor odvysílá XML data přes patřičný kanál do cílové destinace.
Vedlejší toky	—
Výjimky	**Chyba překladu: Neznámý typ zprávy** 1. Zpráva je přesunuta do kanálu „mrtvých“ zpráv **Chyba odeslání: Neznámý GUID nebo služba** 1. Zpráva je přesunuta do kanálu „mrtvých“ zpráv **Chyba spojení** 1. Zpráva je přesunuta do kanálu „mrtvých“ zpráv

Tabulka 6.1: Detail případu Posílej zprávy

- *Producent* — Koncový bod aplikace, která posílá zprávy.
- *Konzument* — Koncový bod aplikace, která přijímá zprávy.
- *Kanál „mrtvých“ zpráv* — Záznam o chybách v komunikaci.

Důležitým rozhodnutím, které bylo dále potřeba učinit, byla otázka formátu dat zpráv putujících po síti. Nakonec bylo rozhodnuto o použití jazyka XML, který má pro tento účel výhodné vlastnosti. XML je prostředek k odstranění vazeb mezi komunikujícími systémy, protože má tu vlastnost, že sám o sobě popisuje data, která reprezentuje. Navíc je to standardizovaný formát dat nezávislý na platformě, čímž přispívá k interoperabilitě (více viz kap. 3.5). Pro tento účel bude nutno definovat XML aplikaci popisující zprávy (viz kap. 7.3).

Jak lze vyčíst z detailu případu užití v tab. 6.1, zaslání zprávy je proces. Producent musí nejdříve vytvořit relaci nad určitým kanálem a zkonstruovat objekty zpráv. Aplikace předávají zprávy komunikátoru, který je ukládá do patřičné fronty k odeslání. Po předání

všech zpráv aplikace ukončí relaci a pokračuje ve své činnosti, což je realizace asynchronní komunikace. Komunikátor postupně serializuje přijaté zprávy do formátu XML a odesílá je prostřednictvím síťových spojení, které jsou dány adresací GUID.

Název	Přijímej zprávy
Aktéři	Vestavěná aplikace
Popis	Konzumentská aplikace přijímá zprávy od vzdálené producentské aplikace.
Počáteční podmínky	Konzumentská aplikace je registrována pro příjem zpráv na určitém virtuálním kanále. V kanálu se nacházejí zprávy.
Koncové podmínky	Konzumentská aplikace přijala zprávy.
Hlavní tok	**Přijetí zprávy** 1. Případ použití se spustí, když komunikátor obdrží zprávu od vzdálené producentrské protistrany. 2. ***Pro všechny zprávy v kanálu***: 2.1 Komunikátor rekonstruuje příchozí proud XML dat do objektu zprávy. 2.2 Komunikátor vyhledá konzumenta zprávy z registrovaných na základě informaci v hlavičce zprávy. 2.3 Komunikátor předá objekt zprávy konzumentovi. 2.4 Konzument zpracuje data z objektu zprávy.
Vedlejší toky	—
Výjimky	**Konzument nenalezen** 1. Zpráva je přesunuta do kanálu „mrtvých“ zpráv **Konzument nerozumí zprávě** (po kroku 2.3) 1. Zpráva je přesunuta do kanálu „mrtvých“ zpráv

Tabulka 6.2: Detail případu Posílej zprávy

Příjem zprávy je proces opačný (tab. 6.2): je nutné akceptovat příchozí spojení, zkonstruovat objekt zprávy a na základě informací v hlavičce zprávy (název kanálu) nalézt konzumenta. Aby komunikátor mohl nalézt konzumenta, musí umožnit aplikacím registrovat se k odběru zpráv. Aplikace, které již dále nemají zájem o příchozí zprávy, se odregistrují.

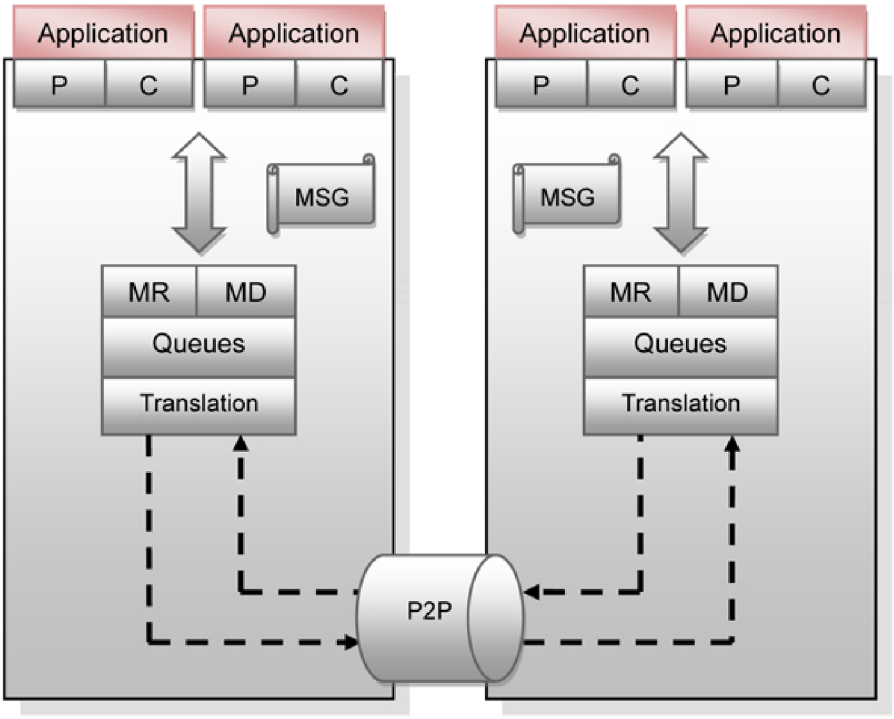
Součástí komunikátoru má být konfigurační rozhraní (týká se případů použití *„Konfiguruj“* a *„Zobraz chyby“*), které umožní definovat infrastrukturu middleware přiřazením IP adres jednotlivým GUID, popř. dodat jinou konfigurační informaci pro nasazení v určitém cílovém prostředí. Dále bude umožňovat zobrazení obsahu záznamu chybové komunikace (kanál „mrtvých“ zpráv).

Případ použití *„Vytvoř zprávu“* vynucuje návrh a implementaci rozhraní pro generování zpráv. Toto rozhraní zahrnuje návrh a implementaci zpráv potřebných v cílovém prostředí (sběrnice zpráv). Objekty zpráv implementují jednotné rozhraní a jsou nositeli uživatelské informace, dodané aplikací (viz kap. 3.4.1). V aplikačním rozhraní jsou tedy využívány klientskými aplikacemi pro zapouzdření a získání přenášené aplikační informace.

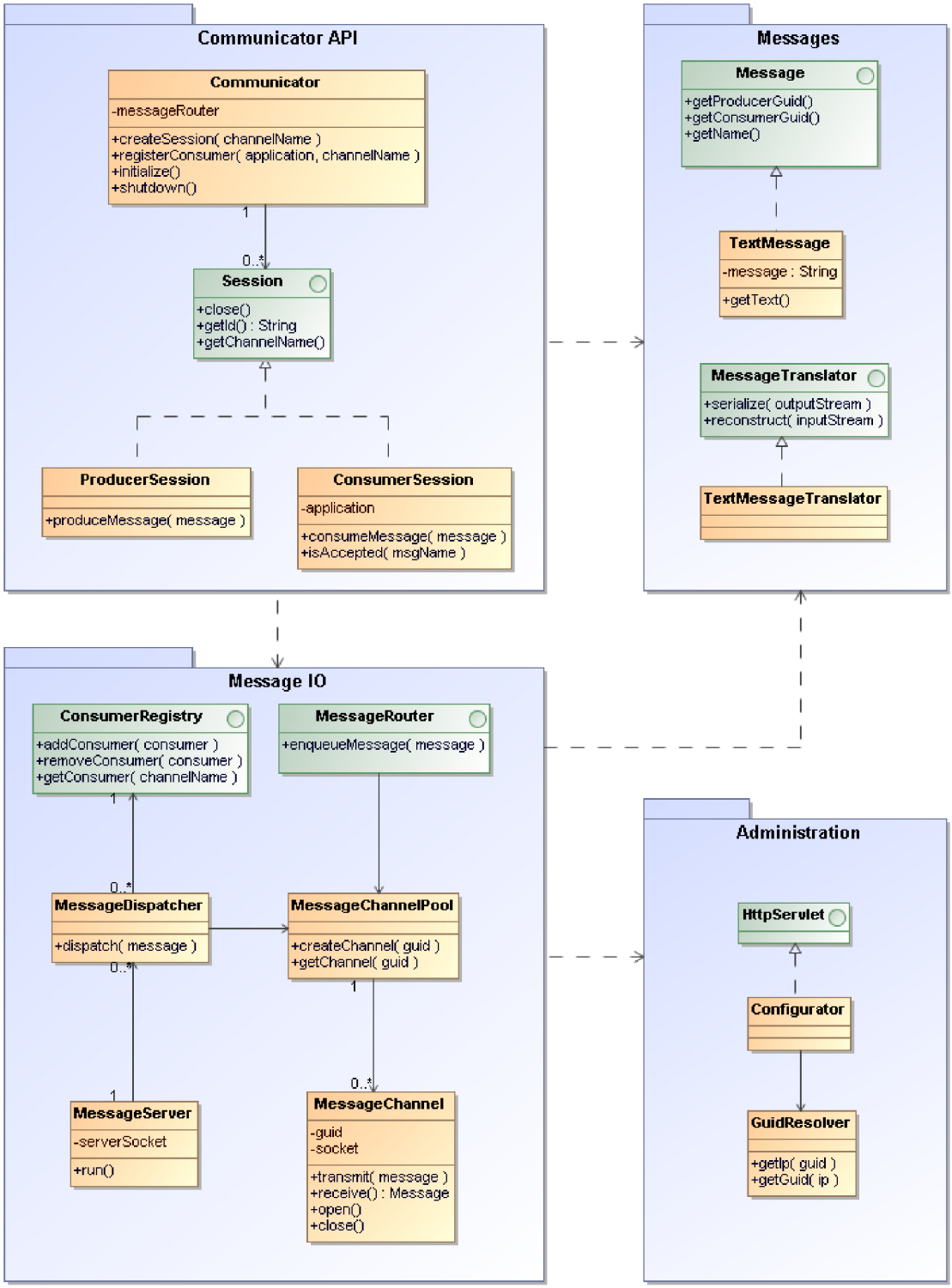
Obrázek 6.2 znázorňuje koncept systému dvojice komunikátorů. Aplikace budou používat komunikátor skrz rozhraní producenta (`P`) a konzumenta (`C`), která dohromady tvoří kon-

cový bod (viz kap. 3.4.3). Nechť producent realizuje aplikační logiku zasílání zpráv (tvorba zpráv z aplikačních dat, ustavení relace atd.) a konzument implementuje reakci na příjem zprávy (zpracování dat ze zprávy). Jádro komunikátoru nechť tvoří systém front pro odchozí a příchozí zprávy (`Queues`), směrovač zpráv (`MR`), rozdělující zprávy od producentů do odchozích front, a dispečer zpráv (`MD`), který naopak přiděluje příchozí zprávy konzumentům. Jednotka `Translation` zajišťuje serializaci objektů zpráv do formátu XML a naopak. Logika `P2P` reprezentuje rozhraní pro tvorbu a správu obousměrných TCP spojení.

Na základě specifikace požadavků a výše provedené analýzy byl sestrojen analytický diagram tříd (viz obr. 6.3). Úkolem tohoto diagramu je ilustrovat logické rozdělení navrhovaného systému podle funkcionality do analytických balíčků a tříd a těm přiřadit odpovědnosti za provedení klíčových činností. Diagram také naznačuje vazby mezi stanovenými entitami. Uvedený postup reflektuje snahu dostát základním požadavkům dobrého objektového návrhu — zachování koheze tříd a minimalizace jejich vzájemných vazeb.



Obrázek 6.2: Schéma komunikace dvou komunikátorů



Obrázek 6.3: Diagram analytických tříd a balíčků

Kapitola 7

Návrh systému

Metodologie stanovená na začátku předchozí kapitoly udává jako fázi následující po analýze požadavků fázi návrhu architektury systému a jednotlivých jeho komponent. V této fázi poslouží diagram analytických tříd a balíčků sestrojený na konci předchozí kapitoly (viz obr. 6.3) jako zdroj návrhových tříd. Dojde k jeho detailnějšímu rozpadu na třídy a rozhraní s vazbou na standardní API použitého jazyka, v tomto případě Javy ME. Jakmile budou tyto třídy stanoveny spolu s předpisem jejich použití (neboli realizacemi případů použití z diagramu 6.1), je možné přistoupit k jejich implementaci.

Nutno dodat, že v praxi se kompletní diagram návrhových tříd realizuje jen zřídka, a to v případech, kdy je použit pro generování zdrojových kódů tříd. Cílem tvůrce návrhu je podat pokud možno jasnou představu o struktuře systému, přičemž model návrhu je průběžně aktualizován v závislosti na skutečnostech, které lze odhalit až při implementaci (např. detaily použití externích knihoven).

Jednotlivými částmi komunikátoru se budeme zabývat v následujících podkapitolách.

7.1 Aplikační rozhraní

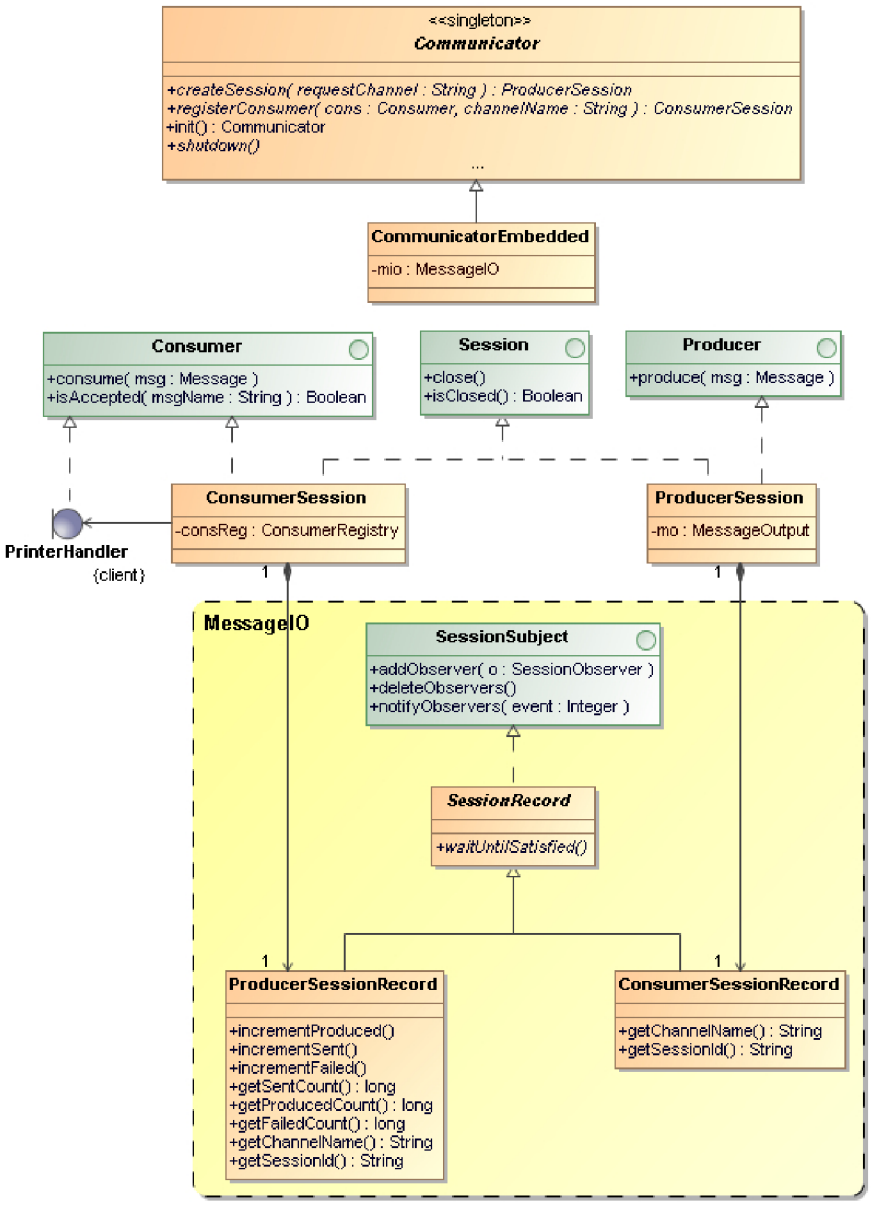
Architektura komunikátoru je organizována do vrstev. Nejvyšší vrstvu tvoří aplikační rozhraní komunikátoru (`Communicator API`), na nižší úrovni se nachází vrstva `MessageIO` (viz kap. 7.2).

Jádro API leží v konceptu relací (*sessions*). Idea relací vychází z nefunkcionálních požadavků. Komunikátor musí být schopen zajistit práci s více spojeními, zároveň však minimalizovat dobu, po kterou jsou spojení držena, a také jejich počet, aby nedocházelo k plýtvání prostředky a zahlcování infrastruktury. Aplikace může zasílat resp. přijímat zprávy jedině skrze relaci, přičemž platí, že aplikace musí ukončit relaci, když sama ví, že již nebude zasílat resp. přijímat další zprávy.

Relace jsou vždy tvořeny nad pojmenovaným kanálem. Název kanálu reprezentuje virtuální cestu mezi komunikujícími aplikacemi, která komunikuje určitý druh informace. Bývá to jednoznačný, snadno zapamatovatelný řetězcový identifikátor (pro kanál komunikující tiskové zprávy např. „printerRequest“). Tento identifikátor je přenášen v hlavičce zprávy a slouží k identifikaci vzdáleného konzumenta, přičemž platí, že na jednom kanále může na daném zařízení naslouchat právě jeden konzument.

K identifikaci uzlů v síti slouží GUID, což je jednoznačná identifikace cílového zařízení (MFD, server). Na uzlu, jemuž je přiřazen GUID, běží jedna instance komunikátoru, která přijímá požadavky z middleware vrstvy od ostatních GUID v síti.

Obrázek 7.1 znázorňuje návrh vrstvy aplikačního rozhraní. V následující části budou popsány stěžejní rozhraní a třídy s ohledem na poskytovanou funkcionalitu.



Obrázek 7.1: Návrhový diagram tříd API

Communicator je abstraktní třída, která tvoří jádro aplikačního rozhraní komunikátoru. Poskytuje metody pro vytvoření relace (`createSession`) a registraci konzumenta (`registerConsumer`) nad určitým kanálem. Protože přístup ke kanálům je realizován prostřednictvím relací, vracejí obě metody objekty implementující rozhraní (`Session`).

Třída je navržena jako *Jedináček* (*singleton*), což zaručuje jedinou globálně přístupnou instanci komunikátoru [4] (pomocí funkce `init`).

ProducerSession, ConsumerSession reprezentují rozhraní pro producentskou a konzumentskou relaci. Relace obecně slouží k vymezení přístupu ke komunikátoru potažmo určitému kanálu.

Producentská relace umožňuje klientské aplikaci zasílat zprávy skrz rozhraní `Producer` a metodu `produce`. Objekt této relace uchovává referenci na rozhraní nižší vrstvy pro další zpracování zprávy a její odeslání (rozhraní `MessageOutput`). Po odeslání všech zpráv je nutné relaci ukončit metodou `close`, což umožní uvolnění prostředků zabraných relací v nižších vrstvách. Zápis zpráv do uzavřené relace je považováno za neplatnou operaci.

Konzumentská relace implementuje rozhraní `Consumer`, zpracování příchozích zpráv metodou `consume` však deleguje na objekt poskytnutý při registraci (v diagramu příklad `PrinterHandler`). Konzument je registrován pro příjem zpráv, pokud je uložen v registru konzumentů (`ConsumerRegister`). Po uzavření konzumentské relace dojde k vyjmutí konzumenta z registru, tzn. jeho odhlášení od kanálu.

Oba typy relací uchovávají záznam o relaci. Tyto záznamy reprezentují relaci v nižších vrstvách komunikátoru, jejich účel je rozebrán v následující kapitole 7.2.

7.2 Vstupně-výstupní vrstva

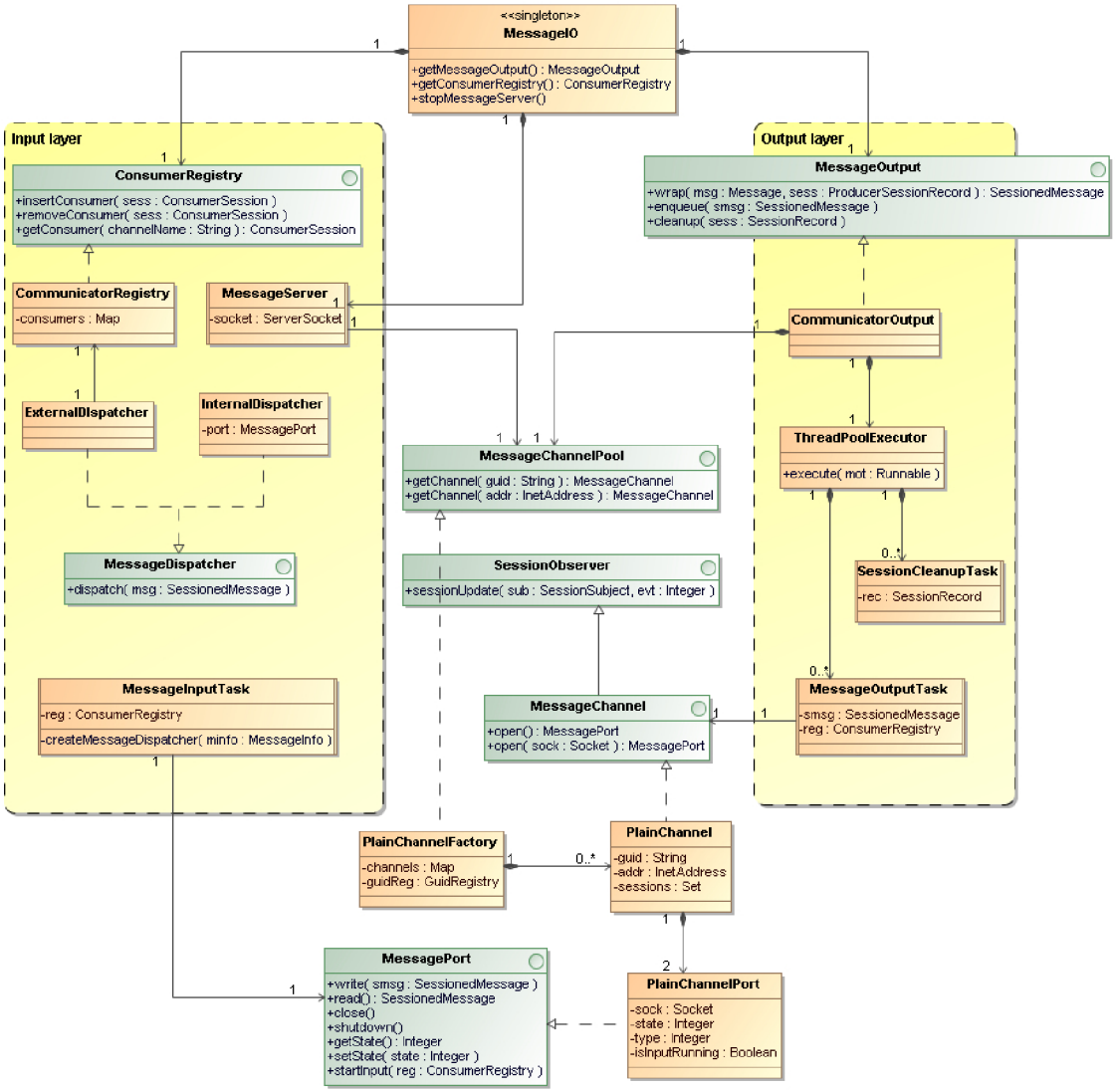
Vstupně-výstupní vrstva (`MessageIO`) zajišťuje přesun aplikačních dat zapouzdřených ve zprávě z lokálního uzlu na vzdálený. Tato činnost zahrnuje:

- Směrování odchozích zpráv ke vzdálenému uzlu na základě identifikace GUID.
- Směrování příchozích zpráv k lokálnímu konzumentovi na základě názvu kanálu.
- Správu síťových spojení na základě stavu relací.
- Transformaci odchozích objektů zpráv do proudu XML dat (serializace)
- Transformaci příchozího proudu XML dat do objektů zpráv (rekonstrukce)

Návrh infrastruktury tříd a rozhraní pro první tři činnosti jsou znázorněny na diagramu 7.2. Zvýrazněny jsou třídy přímo zodpovědné za vyřizování odchozích (`Output layer`) a příchozích (`Input layer`) zpráv.

MessageIO je třída implementující návrhový vzor *Jedináček*, jejímž úkolem je inicializovat vstupní a výstupní vrstvu a poskytnout rozhraní pro přístup k těmto vrstvám. Inicializace zahrnuje i spuštění serveru zpráv, který obsluhuje příchozí spojení.

MessageOutput je rozhraní pro přístup k výstupní vrstvě. Je využíváno objekty producentských relací k umístnění zprávy do odchozí fronty pomocí metody `enqueue`. Parametr této metody je objekt zapozdřující předanou zprávu a informace o relaci, ve které vznikla (`SessionedMessage`). Tento objekt je vytvořen metodou `wrap`, která kromě objektu zprávy, vytvořeného producentskou aplikací, vyžaduje informace o relaci zapouzdřené v objektu `ProducerSessionRecord`, který si udržuje každá producentská relace. Informace o relaci jsou dále součástí hlavičky přenášené zprávy a slouží především k nalezení konzumentské aplikace ve vzdáleném uzlu.



Obrázek 7.2: Návrhový diagram tříd vstupně-výstupní vrstvy

ThreadPoolExecutor je implementace nástroje pro efektivní správu vláken (tzv. *thread pool*) z balíku `java.util.concurrent`, který je standardní součástí JDK 1.5[1]. Protože použitá verze Javy neobsahuje tento balík, byla využita knihovna `backport.util.concurrent`, implementovaná specielně pro účel využití tohoto mocného rozhraní v nižších verzích Javy[2]. Účelem této třídy je redukce režie spojené s vytvářením vláken jejich opětovnout recyklací pro další úkoly (reprezentovány pomocí rozhraní `Runnable`). Jsou vykonávány dva typy úkolů: odeslání zprávy (`MessageOutputTask`) a uvolnění prostředků relace (`SessionCleanupTask`).

MessageOutputTask reprezentuje úkol odeslání zprávy. Tato činnost zahrnuje především serializaci objektu zprávy do proudu XML dat a odeslání přes standardní síťové roz-

[1] JSR 166 — http://jcp.org/aboutJava/communityprocess/final/jsr166/index.html
[2] http://backport-jsr166.sourceforge.net

hraní Javy. K tomu potřebuje jednak objekt zprávy (`SessionedMessage`) a jednak informace o destinaci zprávy (`MessageChannel`).

MessageChannelPool, MessageChannel slouží ke správě a zapouzdření informací o vzdálené destinaci zprávy s určitým GUID a představuje jádro směrovače zpráv. Ačkoliv se v názvu těchto tříd objevuje slovo „kanál“, nijak významově nesouvisí s kanály jako cestami mezi aplikacemi. Zde se jedná spíše o cesty mezi dvěma uzly s odlišným GUID.

Třída `MessageChannelPool` slouží k získání objektu kanálu `MessageChannel` na základě řetězce GUID (využíváno výstupní vrstvou) nebo na základě IP adresy (využíváno vstupní vrstvou). Obousměrný překlad zajišťuje třída `GuidRegistry`. `MessageChannelPool` je aplikací tvořivého návrhového vzoru *Továrna*, který obecně umožňuje lépe strukturovat logiku tvorby objektů s jedním rozhraním avšak několika možnými implementacemi [4].

Třída `MessageChannel` slouží při odesílání a přijmu zprávy k získání tzv. *portu* (`MessagePort`). Objekty kanálů jsou využívány aktivními třídami `MessageOutputTask` pro nasměrování odchozí zprávy do patřičného síťového spojení resp. třídou `MessageServer` pro přidělení vstupního bodu vláknům `MessageInputTask`. Tímto způsobem dochází k efektivnímu sdílení síťových spojení mezi relacemi. V současné době jsou v rámci Továrny implementovány pouze kanály nad klasickými TCP sockety (`PlainChannel`), do budoucna je plánováno použití šifrovaných SSL socketů (`SecuredChannel`).

MessagePort reprezentuje vysokoúrovňovou abstrakci nad TCP sockety pro zápis (metoda `write`) či čtení (metoda `read`) zprávy do výstupního resp. z vstupního proudu. V případě portů je nutné si uvědomit, že může nastat situace, kdy obě vzdálené strany navazují spojení zároveň. Proto obsahuje `MessageChannel` dva porty: primární pro odchozí spojení a sekundární pro příchozí spojení. Pro odchozí zprávy je zpravidla přidělován primární port (metodou `MessageChannel.open` bez parametru, čímž se naváže spojení), pokud však již spojení mezi uzly existuje, je přidělen sekundární port. Metoda `MessageChannel.open` s parametrem `socket` nenavazuje spojení, pouze slouží k obalení TCP socketu do abstrakce portu a vrací vždy sekundární port (to je využíváno při navázání spojení vláknem serveru). Cílem této aplikační logiky je eliminovat situace, kdy jsou zbytečně otevřeny oba porty v určitém kanálu.

SessionObserver, SessionSubject, SessionCleanupTask jsou rozhraní a třída pro správu síťových spojení. Základem je návrhový vzor *Pozorovatel*, díky kterému jsou objekty `MessageChannel` notifikovány v případě uzavření relace [4]. Tyto objekty si dále udržují množinu závislých relací, a pokud velikost této množiny klesne na nulu, je zřejmé, že neexistuje relace, která by potřebovala využít příslušné spojení, a proto je možné jej uzavřít. Notifikaci a uzavírání spojení má na starost právě aktivní třída `SessionCleanupTask`, jejíž vlákno je spuštěno vždy po zavření relace. Aby tento mechanismus fungoval bezchybně, musí toto vlákno čekat na zpracování všech zpráv předaných výstupní vrstvě v rámci relace, což zajišťuje funkce `waitUntilSatisfied` v třídách uchovávající záznamy o relacích (`SessionRecord`). Uzavírání spojení dále podléhá protokolu využívající tzv. *interní zprávy*, o kterých je psáno dále v kapitole o tvorbě zpráv.

MessageServer je implementaci klasického konkurentního serveru pro obsluhu spojení na

TCP portu, jehož číslo je dáno konfigurací. Zpracování příchozího datového toku deleguje nově vytvořeným vláknům `MessageInputTask` v podobě objektu `MessagePort`.

MessageInputTask je aktivní třída zpracovávající vstupní proud XML dat, rekonstruuje z něj objekty zpráv a předává je registrovaným konzumentským aplikacím. Každé vlákno `MessageInputTask` čte ve smyčce pomocí funkce `read` z portu zpráv a končí v momentě ukončení spojení.

MessageDispatcher slouží k doručení rekonstruovaného objektu zprávy konzumentům. Na základě názvu kanálu v hlavičce zprávy může zprávu vyřídit buď interní dispečer (`InternalDispatcher`), který zpracovává interní zprávy (takové, které jsou určeny pouze pro výměnu informací mezi komunikátory) nebo externí dispečer (`External-Dispatcher`), který konzumenta vyhledává v registru `ConsumerRegistry`.

ConsumerRegistry je registr odkazů na konzumentské relace registrované pro odběr zpráv na určitém kanálu. Název kanálu (např. „printerRequest“) slouží jako klíč do asociativního pole, přičemž hodnotou je reference na konzumentskou relaci, která deleguje zpracování volané metody `consume` na objekt konzumenta poskytnutý při registraci metodou aplikačního rozhraní `registerConsumer`.

7.3 Zprávy

Tato kapitola navazuje na předchozí doplněním informací o posledních dvou odpovědnostech vstupně-výstupní vrstvy. Jedná se o serializaci a rekonstrukci objektů zpráv. Nejprve je však důležité rozebrat návrh rozhraní pro tvorbu zpráv z pohledu klientské aplikace a implementaci nových zpráv z pohledu návrháře sběrnice zpráv.

7.3.1 Formát zprávy

Následující úsek XML kódu znázorňuje podobu zprávy putující po síti ve formě řetězce textových XML dat. Protože je tato XML aplikace velmi jednoduchá a pro potřeby použití komunikátoru to nutné není, schéma pomocí některého z jazyků pro popis XML metadat (např. DTD) není definováno.

```
<message>
  <header>
    <srcGuid>XXXX-XXXX-XXXX-XXXX</srcGuid>
    <dstGuid>XXXX-XXXX-XXXX-XXXX</dstGuid>
    <requestChannel>requestName</requestChannel>
    <replyChannel>replyName</replyChannel>
    <corelationId>1234</corelationId>
    <sequenceNum>1</sequenceNum>
  </header>
  <body>
    ...aplikační zpráva...
  </body>
</message>
```

Zpráva se skládá ze dvou částí: hlavičky a těla. Hlavička je společná pro všechny typy aplikačních zpráv a obsahuje povinná pole, která jednak slouží komunikátoru k nalezení cílového uzlu (pole zdrojové GUID `srcGuid` a cílové GUID `dstGuid`) případně konzumentské aplikace (pole `requestChannel`) v cílovém uzlu, a jednak koncovému bodu konzumentské aplikace pro udržení kontextu komunikace. Z pole `replyChannel` se konzument

dozví, na kterém kanálu producent naslouchá pro zprávy s odpovědí, pole `corelationId` udává identifikaci relace, ve které byla tato zpráva doručena a `sequenceNum` udává pořadí zprávy vyprodukované v relaci.

Tělo zprávy slouží již jen konzumentské aplikaci a zapouzdřuje aplikační data nebo příkaz. Podle obsahu těla zprávy jsou rekonstruovány objekty zpráv, které slouží k přímé manipulaci s aplikačními daty. Kořenový element aplikační zprávy nese její název, obsah je dán implementací příslušných tříd zajišťují její serializaci (`MessageExternalizer`) a rekonstrukci (`MessageBuilder`). Příkladem uveďme aplikační zprávu s žádostí o tisk úlohy (`PrintJobMessage`).

```
<body>
  <PrintJob>
    <jobinfo>
      ...info o tisknutém dokumentu...
    </jobinfo>
    <url>http://10.0.5.51/test/tespage.pcl</url>
    <binary>
      ...base64 zakódovaná binární data...
    </binary>
  </PrintJob>
</body>
```

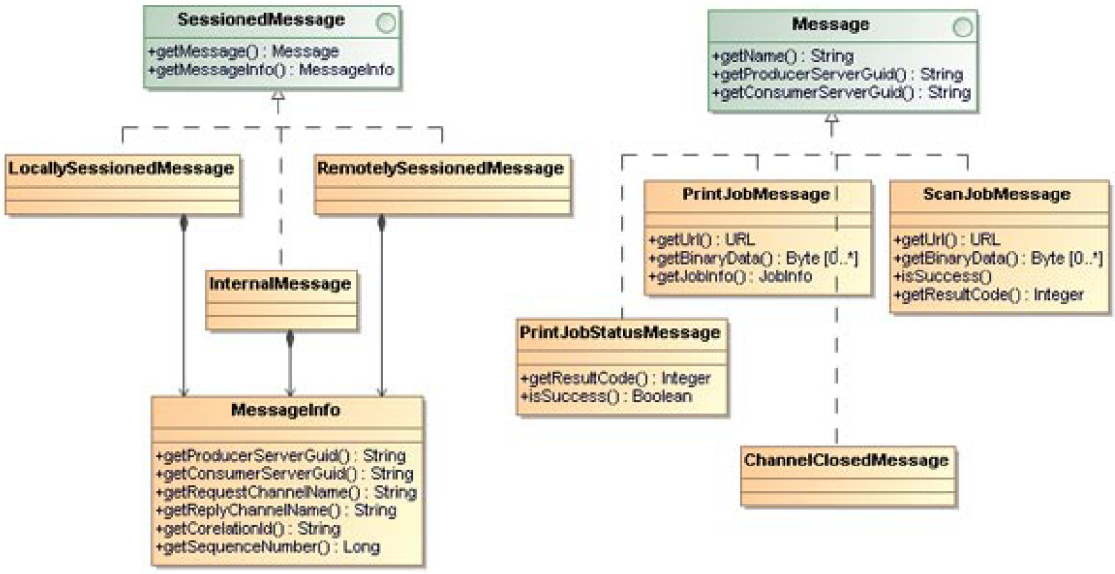
Součástí této zprávy jsou informace o tisknutém dokumentu jako je počet stran, formát velikosti papíru, jazyk popisující dokument (např. PCL, Postrscript, PDF) atd., a samozřejmě odkaz na soubor s daty tiskové úlohy, případně data nese přímo zpráva v CDATA sekci `binary` (v tomto případě je však nutné binární data vhodně zakódovat).

Aby mohla aplikace využívat zprávy, musí být definováno určité aplikační rozhraní. To umožňuje vytvářet objekty zpráv, plnit je daty a následně předávat výstupní vrstvě (metoda `produce` třídy `ProducerSession`) či naopak přistupovat k datům obsaženým v přijaté zprávě (metoda `consume` třídy `ConsumerSession`). Diagram 7.3 znázorňuje třídy, jež jsou součástí tohoto API. Základem jsou dvě hlavní rozhraní `Message` a `SessionedMessage`:

Message reprezentuje zprávu na úrovni kódu producenta a konzumenta a je součástí veřejného API komunikátoru. Návrhář sběrnice zpráv (tj. množiny potřebných aplikačních zpráv) vytvoří třídy implementující toto rozhraní. Zprávy tak mohou být nezávisle implementovány, popř. distribuovány v odděleném Java archívu (JAR).

SessionedMessage reprezentuje zprávu na úrovni vstupně-výstupní vrstvy (`MessageIO`). V objektu třídy `MessageInfo` zapouzdřuje informace o relaci, ve které zpráva vznikla. Tyto informace jsou dále součástí hlavičky zprávy putující po síti. Třída `LocallySessionedMessage` představuje zprávu předanou lokálnímu komunikátoru a určenou k odeslání ve výstupní vrstvě, naopak `RemotelySessionedMessage` rekonstruovanou zprávu ve vstupní vrstvě adresovaného komunikátoru.

InternalMessage reprezentuje typ interní zprávy, která slouží čistě pro interní komunikaci dvou komunikátorů a tím pádem není tvořena producenty ani předávána konzumentům. Sem spadá např. zpráva pro vyjednání uzavření spojení (`ChannelClosedMessage`). Tuto zprávu odesílá komunikátor v momentě, kdy neexistují žádné relace závislé na určitém spojení. Vzdálená strana odpovídá toutéž zprávou v tomtéž případě.



Obrázek 7.3: Návrhový diagram tříd zpráv

7.3.2 Serializace a rekonstrukce

Vrstva zabezpečující serializaci objektů zpráv (převod do textové XML reprezentace) a jejich opětovnou rekonstrukci představuje část vstupně-výstupního rozhraní na nejnižší úrovni.

Práce s formátem XML na aplikační úrovni předpokládá zapojení tzv. XML parseru, což je komponenta pro zpracování XML dat dle určitého paradigmatu. Pro tento účel byl zvolen parser kXML 2 implementující paradigma *pull-parsing* XmlPull 1.1, vyvinuté jako výkonná alternativa pro parsování i generování kódu kompletních XML dokumentů[3]. Oproti modelu DOM API je toto rozhraní mnohem méně paměťové náročné, neboť neukládá celý XML dokument do paměti, a na rozdíl od SAX API je mnohem snadněji použitelné pro dokumenty s rozvětvenější strukturou, neboť kód provádějící parsování reflektuje strukturu dokumentu[4].

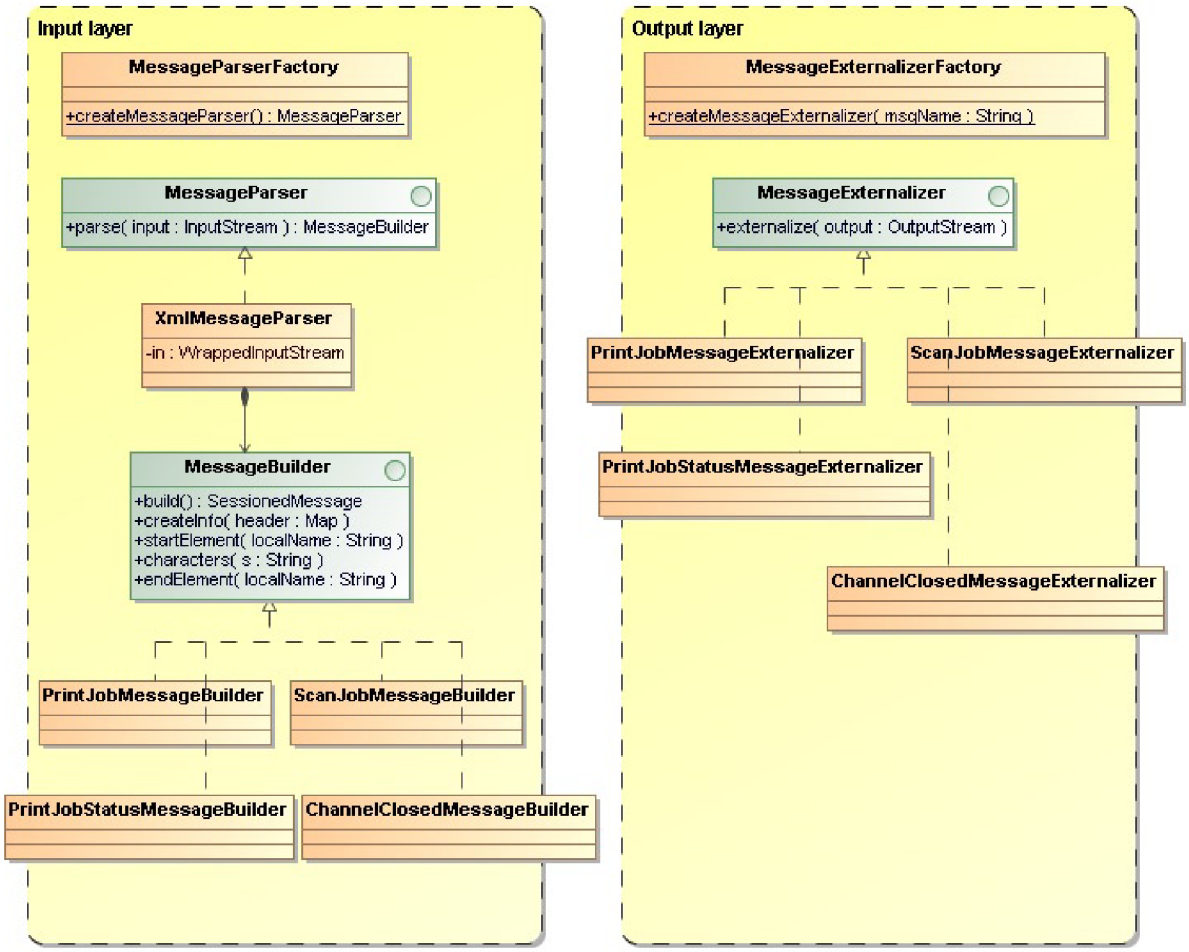
Obrázek 7.4 znázorňuje návrh této vrstvy:

MessageExternalizer je rozhraní pro třídy přímo implementující logiku serializace dat určité zprávy do výstupního proudu socketu. Toto rozhraní obsahuje jedinou metodu `externalize`, jejíž parametr je výstupní proud `OutputStream` standardní knihovny `java.io`. Toto rozhraní je nutno implementovat pro každou aplikační zprávu, jejíž objekty chceme v komunikaci využívat, neboť určuje její strukturu a přítomné datové prvky (to stejné platí pro rozhraní `MessageBuilder`).

MessageBuilder je protějšek rozhraní `MessageExternalizer` ve vstupní vrstvě, neboť určuje předpis pro rekonstrukci příchozího XML proudu do objektu aplikační zprávy.

[3] http://xmlpull.org/history/index.html

[4] DOM parser vytváří obraz stromové struktury kompletního XML dokumentu v paměti, naproti tomu SAX parser postupně prochází strom dokumentu a vyvolává události zpracovávané metodami naprogramovanými vývojářem XML aplikace. Protože struktura dokumentu není kompletně ukládána do paměti, je nutné pro její validaci určitým způsobem udržovat aktuální pozici při průchodu dokumentu.



Obrázek 7.4: Návrhový diagram tříd pro serializaci a rekonstrukci

Třída je zodpovědná za zpracování těla příchozí zprávy, přičemž objekty jsou tvořeny třídou `MessageParser` na základě názvu zprávy v těle. Dodaná data jsou sestavena do objektu zprávy pomocí metody `build`.

MessageParser se stará o parsování vstupního proudu dat s ohledem na definovanou strukturu zprávy (parsuje hlavičku, tělo přenechává objektům `MessageBuilder`). V současné době se počítá pouze s formátem XML (`XmlMessageParser`), v jehož případě je použita nadstavba objektu `InputStream`, získaného ze socketu, speciálně určená pro čtení většího počtu XML dokumentů bez nutnosti pokaždé otevírat nové spojení (`WrappedInputStream`)[5].

7.4 Konfigurační rozhraní

Konfigurace komunikátoru je v současné podobě realizována prostřednictvím konfiguračního souboru dodávaného v distribučním archívu. Formát souboru odpovídá standardnímu Java

[5] Autor Andy Clark, IBM — http://xerces.apache.org/xerces2-j/samples-socket.html

Properties, který se skládá se z posloupnosti direktiv ve tvaru

```
klíč=hodnota,
```

oddělených novým řádkem.

Tento soubor obsahuje převážně informace o infrastruktuře uzlů s danými identifikátory GUID, lze však konfigurovat i jiné vlastnosti komunikátoru, např. číslo portu, na kterém naslouchá, název kanálu pro interní zprávy atd. Soubor může potom vypadat např. takto:

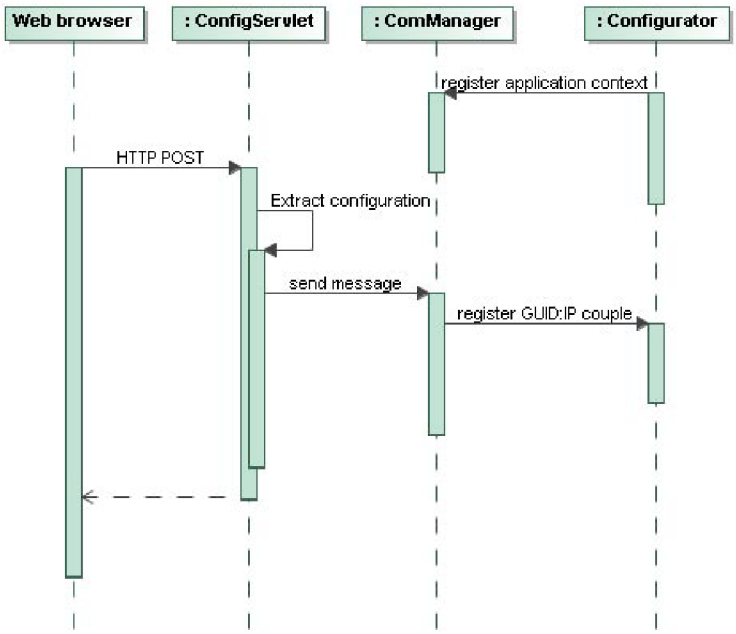
```
communicator.serverPort=15000
communicator.destPort=13500
communicator.internalChannelName=internal

communicator.guid.XXXX-XXXX-XXXX-XXXX=10.0.5.51
communicator.guid.YYYY-YYYY-YYYY-YYYY=10.0.10.228
communicator.guid.ZZZZ-ZZZZ-ZZZZ-ZZZZ=10.0.10.214
```

Nutno dodat, že v případě konfigurace pomocí souboru je tvorba infrastruktury GUID značně nepraktická, neboť každá změna v síti uzlů vyžaduje modifikaci konfiguračního souboru, opětovné sestavení instalačního archívu a instalaci na všechna MFD. Naštěstí v případě platformy SDK/J existuje mnohem efektivnější řešení.

To spočívá v návrhu a implementaci konfigurační aplikace typu klient-server. Základem serverové aplikace, běžící v rámci frameworku OSGi, je jednoduchý HTTP servlet, který získává konfigurační informace o infrastruktuře prostřednictvím dat předaných HTTP metodou POST. Tenký klient na straně administrátora MFD běží v libovolném webovém prohlížeči a umožňuje zadávat konfigurační data přes grafické uživatelské rozhraní. Serverová aplikace běžící na multifunkčním zařízení tato data zpracovává a předává komunikátoru. Proces předání dat ze serverové aplikace do xletu vyžaduje speciální typ meziaplikační komunikace, kterou však platforma SDK/J umožňuje [16]. Sekvenční diagram 7.5 znázorňuje způsob realizace tohoto typu konfigurace realizován.

Základem je třída `ComManager`, která je součástí platformy SDK/J. Ta umožňuje výměnu dat mezi aplikacemi obou zmíněných typů (xlet a servlet). Příjemce dat se nejdřív musí registrovat objektu `ComManager` pod svým produktovým číslem (viz kapitola 8). Vysílací strana poté může přes `ComManager` předávat libovolné objekty aplikacím adresovaným tímto produktovým číslem. `Configurator` je potom třída, která je součástí konfiguračního rozhraní komunikátoru a implementuje rozhraní `GuidRegistry`, využívané pro překlad dvojic GUID:IP (viz třída `MessageChannelPool` ve vstupně-výstupní vrstvě).

Popsaný způsob konfigurace je v současnosti ve fázi analýzy možných rizik a omezení. Pokud by se jej podařilo v budoucnosti bez problému realizovat, mohl by sloužit i jako prostředek pro diagnostiku běhu komunikátoru. Během odesílání resp. přijímání zprávy totiž mohou nastat výjimky, které vedou k přerušení hlavního toku a chybovému stavu komunikace. V takovém případě je zpráva přesunuta komunikátorem případně konzumentem do tzv. kanálu „mrtvých“ zpráv. Tento kanál uchovává informace o zprávách, které nemohly být doručeny, a slouží jako podrobnější záznam pro odhalení chyb komunikace. Webové konfigurační rozhraní by mohlo sloužit k výpisu popř. analýze obsahu tohoto interního kanálu.



Obrázek 7.5: Sekvenční diagram konfigurace pomocí aplikace typu klient-server

7.5 Vestavěný terminál

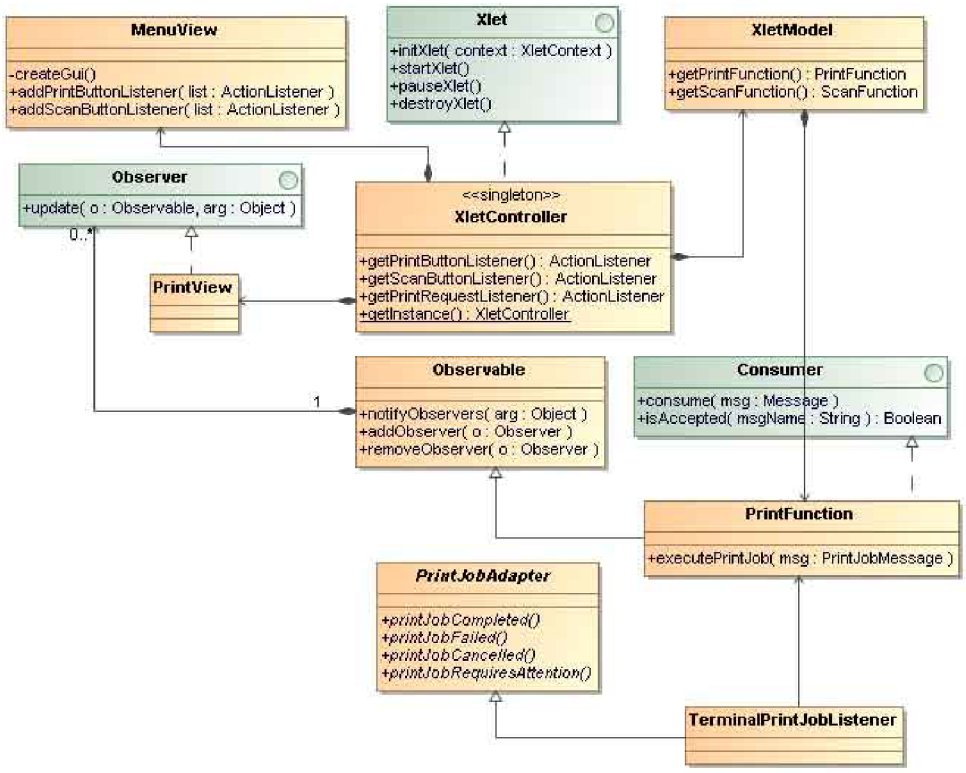
V rámci praktické demonstrace funkcionality komunikátoru byla navržena a implementována aplikace v prostředí správy tisku. Jedná se o základ vestavěného terminálu nové generace pro MFD jako softwarovou variantu externího terminálu (viz obr. 5.1). Aplikace běží na multifunkčním zařízení Ricoh Aficio MP 2550 (obr. 8.1) jako xlet a pro komunikaci s tiskovým serverem využívá komunikátor jako knihovnu.

V současné podobě umožňuje vestavěný terminál správu účtování tisku a skenování dokumentů, nicméně do budoucna jsou možnosti téměř neomezené. Budoucí vývoj tak bude směřovat k implementaci softwarového terminálu, který plně odpovídá možnostem původního externího terminálu. Příkladem lze navrhnout protokol a příslušné zprávy pro účtování kopírování či faxování, nebo pro autentizaci prostřednictvím integrované čtečky karet.

Uvedený typ multifunkčního zařízení disponuje barevným dotykovým LCD panelem, který umožňuje operátorovi manipulovat s jednoduchým grafickým uživatelským rozhraním vestavěného terminálu. Základem pro návrh se proto stal vzor *Model-View-Controller* (MVC), který slouží v aplikacích s grafickým uživatelským rozhraním pro oddělení aplikační logiky od její prezentační části. Obrázek 7.6 znázorňuje návrh testovací aplikace s návazností na standardní třídy aplikačního rozhraní MFD (Device SDK API).

Xlet je rozhraní MFD API, které reprezentuje kontext aplikace typu xlet. Vývojář xletu implementuje jednotlivé metody pro přechod mezi stavy xletu (viz obr. 4.5). Je nutné si uvědomit, že operátor MFD může jednotlivé běžící xlety libovolně pozastavovat a spouštět a xlet musí s těmito přechody počítat. Implementaci xletu poskytuje třída `XletController`.

XletController implementuje vedle vstupního bodu xletu také kontrolér v MVC struktuře



Obrázek 7.6: Návrhový diagram tříd vestavěného terminálu

aplikace. Ten je zodpovědný za zpracování podnětů z uživatelského rozhraní (přesněji z jednotlivých pohledů) a za přechody mezi obrazovky (pohledy) aplikace. K tomuto účelu poskytuje sadu tříd pro zpracování uživatelských událostí (events), které odpovídají třídám standardní sady pro tvorbu uživatelského rozhraní AWT (tzv. action listeners).

XletModel představuje aplikační byznys logiku terminálu. Tato logika se může nacházet v různých stavech v závislosti na tom, jaké funkce terminálu jsou užívány. V současnosti je implementována pouze funkce pro správu tisku a skenování. Na diagramu 7.6 je příkladem znázorněn návrh funkcionality tisku.

PrintFunction je část modelu, která se stará o vyřizování požadavků na tisk dokumentu. Implementuje rozhraní `Consumer`, což jí umožňuje přijímat zprávy s tiskovými úlohami (`PrintJobMessage`) od serveru na určitém kanálu (v tomto případě „print-Request“). Zároveň dědí ze třídy `Observable` (součást standardní knihovny `java.-util`), aby mohl pohled (view) reagovat na asynchronní změny stavu modelu (např. příchod zprávy, dokončení tisku) a realizovat změny v GUI (např. zobrazení modálního okna, deaktivace tlačítek).

PrintJobAdapter je abstraktní třída MFD API pro zpracování událostí o stavu tiskové úlohy. Jakmile je např. tisk fyzicky dokončen je vyvolána metoda `printJobCompleted` v děděné třídě (`TerminalPrintJobListener`) a ta může patřičně zareagovat. V tomto případě je zaslána zpráva `PrintJobStatusMessage` o výsledku tiskové úlohy zpět

serveru a provedena notifikace pohledu.

MenuView, PrintView jsou pohledy v architektuře MVC, přesněji řečeno jednotlivé obrazovky grafického uživatelského rozhraní. Jejich zodpovědností je vykreslování uživatelských primitiv jako např. tlačítka a popisky a registrace obsluh uživatelských událostí. Některé pohledy zároveň mohou být příjemci asynchronních událostí od modelu prostřednictvím rozhraní `Observer`. Pohled `MenuView` vykresluje obrazovku hlavního menu, `PrintView` obrazovku s přijatými tiskovými úlohami.

Kapitola 8

Nasazení

Tato kapitola si klade za cíl seznámit čtenáře s postupem nasazení komunikátoru do prostředí distribuované tiskové aplikace a ukázat jeho praktické použití. Funkčnost bude předvedena na modelu klient-server pomocí testů sestrojených v testovacím prostředí jUnit. Testy budou spouštěny na testovací pracovní stanici v roli tiskového serveru a budou komunikovat pomocí komunikátoru s aplikací vestavěného terminálu běžící v MFD Ricoh Aficio MP 2550 (viz obr. 8.1).

Obrázek 8.1: Ricoh Aficio MP 2550

8.1 Instalace

Aplikace (typu server a xlet) jsou do multifunkčního zařízení instalovány buď přímo z pamětí Secure Digital nebo z webového serveru v lokální síti. V Embedded Software Architecture slouží jedna SD karta jako úložiště pro operační systém a běhové prostředí

virtuálního stroje Javy a na druhou jsou instalovány aplikace zabalené do JAR, doplněné tzv. DALP soubory, což jsou XML soubory s konfiguračními informacemi.

Instalaci lze provést buď přímo z panelu zařízení, nebo přes webové administrátorské rozhraní zařízení v síti. Druhý přístup je mnohem pohodlnější a obzvlášť vhodný v případě, kdy provádíme ladění aplikace. Obrazovou dokumentaci k oběma způsobům instalace lze nalézt v přílohách této práce.

Schéma 8.2 zobrazuje strukturu instalačního archívu pro instalaci přes webové rozhraní včetně externích knihoven použitých v aplikaci. Jak lze vidět, instalační archív obsahuje kromě aplikace vestavěného terminálu (`ComEmbTestApp-100.jar`) a knihovny s komunikátorem (`ComEmb-100.jar`) také knihovnu pro logování běhu aplikace Micrologu[1], pull parser kXML 2[2]a rozhraní pro programování s vlákny[3]. `ComEmbTestApp-100.dalp` obsahuje konfigurační informace pro nalezení archívu s xletem (obsah viz příloha).

```
320400100.zip
    |----sdk
          |----dsdk
                |----dist
                      |----320400100
                              |----microlog-logger-1.1.1-me.jar
                              |----kxml2-2.3.0.jar
                              |----backport-util-concurrent.jar
                              |----ComEmb-100.jar
                              |----ComEmbTestApp-100.jar
                              |----ComEmbTestApp-100.dalp
```

Schéma 8.2: Adresářová struktura instalačního archívu

Instalace aplikace se provádí pod přiděleným produktovým číslem (320400100). Archív s implementací xletu (`ComEmbTestApp-100.jar`) musí být digitálně podepsán certifikátem vydaným na základě oficiální žádosti přímo firmou Ricoh, která se tak chrání proti zneužití poskytnutého aplikačního rozhraní.

8.2 Použití

Demonstrace bude provedena na příkladu odloženého tisku dokumentu ve formátu PCL. Pro konkrétnější informace ohledně použití aplikačního rozhraní komunikátoru lze nahlédnout do příloh této práce (viz sekvenční diagram E.1). Snímky uživatelského rozhraní panelu multifunkčního zařízení vztahující se k demonstraci použití jsou umístněny tamtéž.

Po úspěšné instalaci vestavěného terminálu s komunikátorem a jeho spuštění je na LCD panelu multifunkčního zařízení zobrazena obrazovka jednoduchého menu s dvěma tlačítky pro tisk a skenování (viz příloha F.1(a)).

Již v tomto momentě je komunikátor registrován na kanálu „printerRequest“ pro příjem zpráv. Přijaté tiskové úlohy uživatel zobrazí stiskem tlačítka s obrázkem tiskárny v menu.

[1]Implementace log4j API pro Java ME - http://microlog.sourceforge.net/snapshot

[2]http://kxml.sourceforge.net

[3]Zpětná implementace balíku `java.util.concurrent` z Javy SE 1.5 do starších verzí Javy - http://backport-jsr166.sourceforge.net

Jakmile přijde z testovací stanice zpráva `PrintJobMessage`, je zobrazeno tlačítko, které umožňuje pokračovat vytisknutím přijaté úlohy (viz příloha F.1(b)). Příchozí zpráva může vypadat například takto:

```
<message>
  <header>
    <srcGuid>XXXX-XXXX-XXXX-XXXX</srcGuid>
    <dstGuid>ABCD-1234-ABCD-1234</dstGuid>
    <requestChannel>printRequest</requestChannel>
    <replyChannel>printReply</replyChannel>
    <corelationId>2346ad27d7568ba9896f1b7da6b5991251debdf2</corelationId>
    <sequenceNum>1</sequenceNum>
  </header>
  <body>
    <PrintJob>
      <jobinfo>
        <language>PCL</language>
        <pages>64</pages>
        <format>A4</format>
      </jobinfo>
      <url>http://10.0.5.51/test/testpage.pcl</url>
      <binary encoding="base64"></binary>
    </PrintJob>
  </body>
</message>
```

Po stisku tlačítka tiskové úlohy se objeví dialogové okno s informacemi o přijaté tiskové úloze. Uživatel poté potvrdí, zda chce skutečně vytisknout zvolený dokument a po potvrzení je úloha fyzicky vytištěna (viz příloha F.1(c))

Po vytištění je testovací stanici zpět zaslána zpráva s informací o stavu vytištěné úlohy `PrintJobStatusMessage`. Server poté může úlohu zaúčtovat:

```
<message>
  <header>
    <srcGuid>ABCD-1234-ABCD-1234</srcGuid>
    <dstGuid>XXXX-XXXX-XXXX-XXXX</dstGuid>
    <requestChannel>printerReply</requestChannel>
    <replyChannel></replyChannel>
    <corelationId>2346ad27d7568ba9896f1b7da6b5991251debdf2</corelationId>
    <sequenceNum>2</sequenceNum>
  </header>
  <body>
    <PrintJobStatus>
      <code>JOB_SUCCESS</code>
    </PrintJobStatus>
  </body>
</message>
```

V poslední fázi se iniciátor komunikace (testovací stanice) snaží uzavřít spojení. Předtím však posílá zprávu `ChannelClosedMessage` po interním kanálu (`internal`), kterou se dotazuje, zda je možné uzavřít spojení. Pokud není spojení na straně MFD právě využíváno žádnou jinou relací, je zpráva `ChannelClosedMessage` zaslána zpět a spojení je uzavřeno. Strany považují spojení za uzavřené, pokud obě přijaly i odeslaly zprávu `ChannelClosedMessage`.

Kapitola 9

Závěr

Integrace aplikací je komplexní problém, který nabývá mnoha podob, jehož řešení bere v úvahu mnohá kritéria a dilemata. Jedním z nejdůležitějších kroků k efektivní integraci je však volba vhodných prostředků, které co možná nejvíce redukují vzájemné závislosti mezi aplikacemi. Bylo uvedeno několik možných přístupů k integraci, přičemž princip komunikace pomocí zpráv byl prezentován jako nevhodnější varianta vzhledem k cílovému prostředí.

V rámci této práce byla kompletně navržena a implementována softwarová komponenta (komunikátor) realizující komunikační vrstvu integrující funkcionalitu multifunkčních kancelářských zařízení na platformě Java ME do cílového prostředí pro správu firemního tisku podniku, čímž navázala na předchozí práci, zabývající se převážně teoretickými východisky pro tento typ integrace a charakteristikou cílového prostředí. Tato komponenta byla úspěšně nasazena na platformu Embedded Software Architecture zařízení od firmy Ricoh, kde byla použita spolu s jednoduchou implementací softwarového terminálu pro realizaci zabezpečeného odloženého tisku (získání dat a potvrzení stavu tiskových úloh vzdálené aplikaci v roli tiskového serveru).

Současný stav vývoje komunikátoru si vyžaduje především implementaci četných zátěžových testů, které by odhalily možné chybové stavy v komunikaci mezi MFD a tiskovým serverem. Plného využití by komunikátor dosáhl ve spojení s vestavěným terminálem, který by umožnil uživateli autentizovat se vůči tiskovému serveru přímo na MFD. Z tohoto pohledu je prezentované řešení zabezpečeného odloženého tisku nedokonalé, neboť je třeba navrhnout jak potřebné autentizační zprávy a autentizační logiku na straně tiskového serveru, tak uživatelské rozhraní pro vestavěný terminál. To platí i pro další funkcionalitu mimo tisk, jako je podpora skenování, která je aktuálně ve fázi implementace, a dále kopírování a faxování (aktuálně ve fázi analýzy). Tímto způsobem by projekt vestavěného terminálu navázal na tuto práci.

Protože je komunikátor od počátku navrhován pro obecnější použití, je teoreticky možné jej využít i v jiných prostředích než ve správě tisku. Lze jej využít všude tam, kde je potřeba uniformní škálovatelné rozhraní pro obousměrnou komunikaci aplikací, ať už se jedná např. o průmyslové využití či P2P sítě pro sdílení dat. Podmínkou je vytvoření specifické množiny potřebných příkazových a dokumentových zpráv a případná implementace na obecněji využitelné platformy, než je Java ME.

Přínos této práce spatřuji nejen v předpokládaném komerčním využití v prostředí aplikace SafeQ, ale také v osobním rozvoji z hlediska znalostí technik unifikovaného procesu vývoje aplikací, programovacího jazyka Javy a rozšíření obzorů v oblasti možností dnešních vestavěných systémů, konkrétně multifunkčních kancelářských zařízení, a správy tisku ve větších firmách.

Literatura

[1] Arlow, J.; Neustadt, I.: *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. Addison Wesley, druhé vydání, July 2005.

[2] Chappell, D.: *Enterprise Service Bus*. Sebastopol, CA: O'Reilly, June 2004.

[3] Erl, T.: *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ: Prentice Hall PTR, August 2005.

[4] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. M.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, Boston: Addison Wesley, 2001.

[5] Harold, E. R.; Means, W. S.: *XML in a Nutshell*. Sebastopol, CA: O'Reilly, třetí vydání, September 2004.

[6] Hohpe, G.; Woolf, B.: *Enterprise Integration Patterns: Designing, building, and deploying messaging solutions*. Addison Wesley, 2004.

[7] Kaye, D.: *Loosely Coupled: The Missing Pieces of Web Services*. Marin County, CA: RDS Press, 2003.

[8] Kosek, J.: XML schémata. srpen 2005, [Online, navštíveno 18.4.2009].
URL http://msdn.microsoft.com/en-us/library/ms978710.aspx

[9] McLaughlin, B. D.; Edelson, J.: *Java and XML*. Sebastopol, CA: O'Reilly, třetí vydání, December 2006.

[10] Melonfire, C.: Understanding the pros and cons of the Waterfall Model of software development. *TechRepublic*, 2006, [Online].
URL http://articles.techrepublic.com.com/5100-10878_11-6118423.html

[11] MSDN: Entity aggregation. 2009, [Online].
URL http://msdn.microsoft.com/en-us/library/ms978573.aspx

[12] MSDN: Functional integration. 2009, [Online].
URL http://msdn.microsoft.com/en-us/library/ms978578.aspx

[13] MSDN: Integrating layer. 2009, [Online].
URL http://msdn.microsoft.com/en-us/library/ms978710.aspx

[14] MSDN: Portal integration. 2009, [Online].
URL http://msdn.microsoft.com/en-us/library/ms978588.aspx

[15] Ricoh: *Introduction of Embedded Software Architecture: Embedded Software Architecture Version 4.10*. Ricoh Co., Ltd.

[16] Ricoh: *Xlet Type Developer's Guide (Xlet): Embedded Software Architecture Version 4.10*. Ricoh Co., Ltd.

[17] Ráb, J.: Přednáška předmětu Prostředí distribuovaných aplikací: Principles and characteristics of distributed systems and environments. 2009, [Online].

[18] SDN: *Java Secure Socket Extension: Reference Guide*. Sun Microsystems, Inc., [Online].
URL http://java.sun.com/j2se/1.4.2/docs/guide/security/jsse/JSSERefGuide.html

[19] Uddin, M.: *SafeQ: Technická dokumentace v3.1.05*. Y Soft, Ltd., srpen 2007.

[20] Wikipedia: Business process. 2009, [Online; Last modified on 30 March 2009, at 07:30].
URL http://en.wikipedia.org/wiki/Business_process

[21] Wikipedia: Information systems. 2009, [Online; Last modified on 3 January 2009, at 20:32].
URL http://en.wikipedia.org/wiki/Information_system

[22] Wikipedia: Middleware. 2009, [Online; Last modified on 3 January 2009, at 18:19].
URL http://en.wikipedia.org/wiki/Middleware

[23] Wikipedia: Service-oriented architecture. 2009, [Online; Last modified on 3 January 2009, at 00:36].
URL http://en.wikipedia.org/wiki/Service-Oriented_Architecture

Seznam použitých zkratek a symbolů

SOA – Service-oriented architecture

HR – Human Resources

API – Application Programming Interface

CORBA – Common Object Request Broker Architecture

RMI – Remote Method Invocation

JMS – Java Message Service

SOAP – Simple Object Access Protocol

HTTP – Hypertext Transfer Protocol

CSV – Comma-separated Values

SQL – Structured Query Language

PCI – Peripheral Component Interconnect

RSS – Really Simple Syndication

XHTML – Extensible Hypertext Markup Language

SVG – Scalable Vector Graphics

DTD – Document Type Definition

XSD – XML Schema Definition

SGML – Standard Generalized Markup Language

LPR – Line Printer Deamon Protocol

PCL – Printer Command Language

LDAP – Lightweight Directory Access Protocol

SDK – Software Development Kit

LCD – Liquid Crystal Display

CDC – Connected Device Configuration

AWT – Abstract Window Toolkit

OSGi – Open Services Gateway initiative

JSP – JavaServer Pages

UML – Unified Modeling Language

GUID – Globally Unique Identifier

P2P – Peer To Peer

CDATA – Character Data

JAR – Java Archive

MVC – Model-View-Controller

DOM – Document Object Model

SAX – Simple API for XML

PDF – Portable Document Format

CRM – Customer Relationship Management

MFD – Multi-function Device

XML – Extensible Markup language

Seznam příloh

Příloha A

Funkcionální požadavky

Name: FR-1: Create session
Description: The communicator shall create a session on request of application.
Analysis: The main purpose of sessions shall be to provide a single interface for using the communicator and to define an interval in which single inter-application communication is running, allowing effective manipulation with related network connections. The session shall have a state which determines its liveliness.

Name: FR-2: Return session ID
Description: A session shall have its ID which identifies the current context of communication.
Analysis: The communicator shall return session ID if and only if session is successfully established. The session ID is a fixed length securely unique number.

Name: FR-3: Close sessions
Description: The communicator shall close a session on request of application.
Analysis: When the sending application terminates the session, the network connections that are not currently used by the other sessions are closed, and all resources released. The applications are responsible for closing created sessions.

Name: FR-4: Handle inactive session
Description: The communicator shall release inactive sessions.
Analysis: A session is considered inactive if it is not used by owner application for a long time period. All inactive sessions shall be automatically closed. An attempt to process messages via such session shall throw an exception. Closing an inactive session shall be considered empty operation.

Name: FR-5: Register consumers
Description: An application shall register to the communicator to be notified as the messages arrive.
Analysis: The consumer shall obtain a session when successfully registered and be able to receive messages until the corresponding session is closed. The consumer shall be able to receive messages of compatible types only

Name: FR-6: Unregister consumers
Description: An application shall unregister from the communicator when no longer needs to receive messages.

Analysis: To unregister, a consumer application simply closes its consumer session obtained by registration. Then the application will no longer be able to receive messages.

Name: FR-7: Construct message objects
Description: The communicator shall expose interface for constructing various kinds of messages.
Analysis: A message object shall comply with the uniform message interface to allow unifying the message creation process.

Name: FR-8: Accept message objects
Description: The communicator shall accept message data from producer applications via an active session.
Analysis: The producer shall obtain a session from the communicator before it is able to send messages. Accepted messages are translated to a canonical format.

Name: FR-9: Route messages
Description: The communicator shall resolve network destination of accepted messages.
Analysis: The routing information is provided by GUID.

Name: FR-10: Translate Messages
Description: The communicator shall translate message objects into protocol messages in a canonical format.
Analysis: The communicator constructs a header which contains information necessary for proper identification of the message (e.g. receiver and sender identification, type of data in the message, ...) and body holding data an application data. The canonical format shall be XML.

Name: FR-11: Discover message consumer
Description: The communicator shall identify consumer application which is a received message intended for.
Analysis: On the consumer's side the right one from the registered consumers is recognized according to the message's recipient name. Additionally, each consumer shall mind the compatible message types.

Name: FR-12: Parse Messages
Description: The communicator shall parse data from the canonical format into a specific message object.
Analysis: The type of the message is determined by the name field in the canonical message header.

Name: FR-13: Deliver message objects
Description: The communicator shall deliver parsed message object to application identified as the recipient.
Analysis: The application shall implement some kind of listener interface and is notified when the message arrives. The message data are passed via method of interface. The consumer shall have a fully initialized session when consuming a message. When such a session should be created is going to be specified in the design phase.

Name: FR-14: Create channels
Description: The communicator shall create a new channel only when channel for specified destination does not exist.
Analysis: The channel is a route for messages between two communication endpoints identified by GUID. It is based on IP sockets. A channel is created when there are messages to be transmitted and no corresponding channel is available.

Name: FR-15: Close channels
Description: The communicator shall close the channel and release all related resources when there are no sessions using it.
Analysis: The corresponding socket shall be closed along with the channel. There shall be an adequate timeout between the events when the channel is unused and closed to prevent possible frequent socket connection establishments and closings.

Name: FR-16: Configure communicator
Description: The administrator shall provide the communicator with configuration information.
Analysis: The configuration information comprises GUID endpoint addresses and associated IP addresses, port number of the communicator and other information related to the communicator's functions. The configuration interface shall consist of client side web interface and a configuration application on the side of the communicator.

Name: FR-17: Configuration client
Description: The graphical user interface for setting the configuration properties.
Analysis: The client is web-based. The administrator shall use it to connect to the communicator-side configuration application via web browser.

Name: FR-18: Configuration application
Description: Configuration application on the side of the communicator.
Analysis: The configuration application shall accept configuration information from logged clients a use it to configure the communicator.

Name: FR-19: Logging
Description: The communicator shall expose communication log to the administrator.
Analysis: The log shall be accessible online via remote logger and stored in file on MFD's HDD.

Příloha B

Nefunkcionální požadavky

Name: NR-1: Encrypted Messages
Description: The system shall encrypt message data using SSL.
Analysis: Considering the Ricoh SDK/J platform and corresponding JRE, the data encryption shall be realized using Java Secure Socket Extension (JSSE 1.0.3).

Name: NR-2: XML structure
Description: The data transferred between applications on the protocol message level shall be formatted using XML.
Analysis: The format of protocol messages shall be determined by a metadata description document (XML schema).

Name: NR-3: Connection persistence
Description: The connection persists for a necessary amount of time only.
Analysis: While sending or receiving messages the communicator shall keep the connection open for a minimum necessary amount of time. Otherwise the channels maintaining the connection shall be closed.

Name: NR-4: Handle multiple connections
Description: The communicator shall effectively handle multiple connections.
Analysis: To reduce memory consumptions and overhead of constant thread creation, a simple thread pool pattern shall be used.

Name: NR-5: Handle network errors
Description: The communicator shall effectively handle network connection errors and delays.
Analysis: The communicator shall exert the best effort to connect to the remote application. In case of need it shall poll a number of times until successfully connected. There shall be limitations on the time period for which the I/O may block. Unsuccessful network communication is logged.

Příloha C

Požadavky externího rozhraní

Name: EI-1: Deployment
Description: The communicator shall be designed for deployment on MFDs.
Analysis: The primary platform for development and testing of the initial version of the communicator shall be Ricoh Embedded Software Architecture.

Name: EI-2: Energy mode transitions
Description: The communicator shall cope with transitions between MFD's energy modes.
Analysis: On Ricoh, Xlet applications supporting Energy Save mode is prohibited of writing to HDD/SD while in Energy Save Mode. Before writing to HDD/SD the recover from Energy Save mode must be made to prevent HDD/SD from damage. In other words, the application must prevent from transiting to Energy Save mode while writing to HDD/SD.

Name: EI-3: Port number limitation
Description: The communicator shall use only port numbers available at host MFD.
Analysis: The communicator running on Ricoh shall use a port number higher than 50000.

Name: EI-4: Maximum available HDD space
Description: The communicator shall manage limited HDD space.
Analysis: As for Ricoh, the maximum available disk space per Xlet is 1GB.

Name: EI-5: Memory restrictions
Description: The communicator shall effectively manage memory resources of host device.
Analysis: The communicator running on Ricoh shall mind currently configured heap and stack sizes.

Name: EI-6: Process initialization
Description: The communicator shall be initialized within a defined amount of time.
Analysis: Ricoh Xlet initialization timing restriction is set to 3 seconds.

Name: EI-7: Process start
Description: The communicator shall be started within a defined amount of time.
Analysis: Ricoh Xlet start timing restriction is set to 1 second.

Name: EI-8: Process finish
Description: The communicator process shall be terminated within a defined amount of time.
Analysis: Ricoh Xlet process end timing restriction is set to 0.5 second.

Příloha D

Obrazová instalační dokumentace

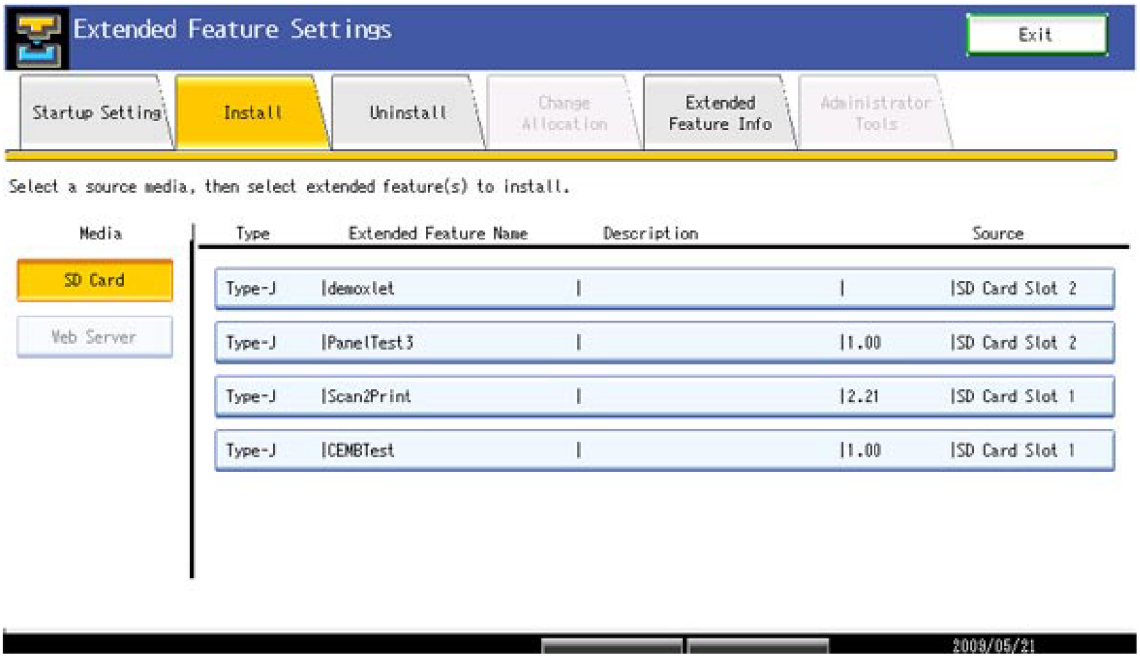
Tato kapitola přílohy prezentuje oba způsoby instalace komunikátoru, potažmo vestavěného terminálu na MFD Ricoh Aficio MP 2550, prostřednictvím snímků webového rozhraní a instalační aplikace přímo v zařízení.

Snímek D.1 zobrazuje ovládací panel zařízení Ricoh Aficio MP 2550 se spuštěným systémovým menu, z něhož je možné se dostat do instalačního menu, zobrazené na snímku D.2.

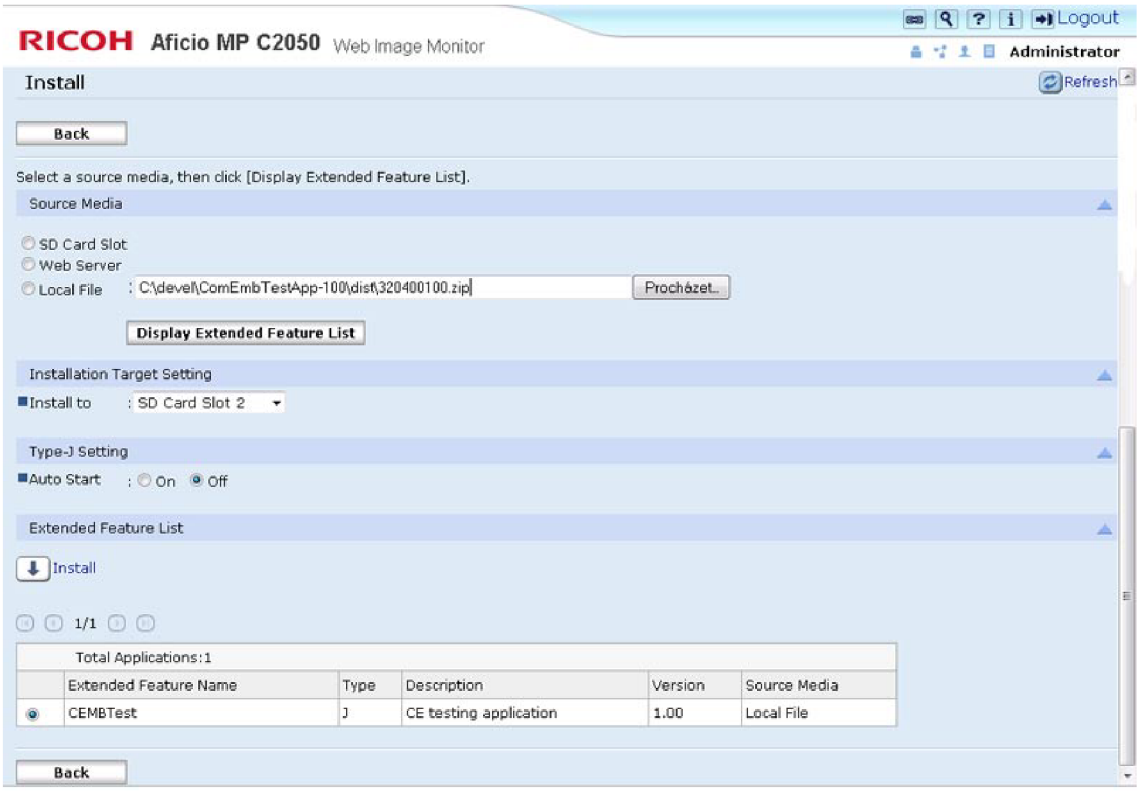
V tomto menu lze instalovat aplikace s určitého umístnění (SD karta nebo webový server). Po zvolení instalačního zdroje jsou zobrazeny dostupné instalační archívy. Zvolením archívu je zobrazen dialog dotazující se na umístnění instalované aplikace (lze instalovat opět na SD kartu nebo i na pevný disk). Po nainstalování musí být nakonec aplikace nastartována (Startup Setting).

Webové instalační rozhraní (viz snímek D.3) je část administrátorského rozhraní, do kterého se lze přihlásit zadáním WWW adresy tiskárny v síti do libovolného webového prohlížeče. Možnosti odpovídají instalačnímu menu v MFD, zde je však navíc možnost poslat instalační archív přímo z lokálního počítače.

Obrázek D.1: Ovládací panel Ricoh Aficio MP 2550 se systémovým menu



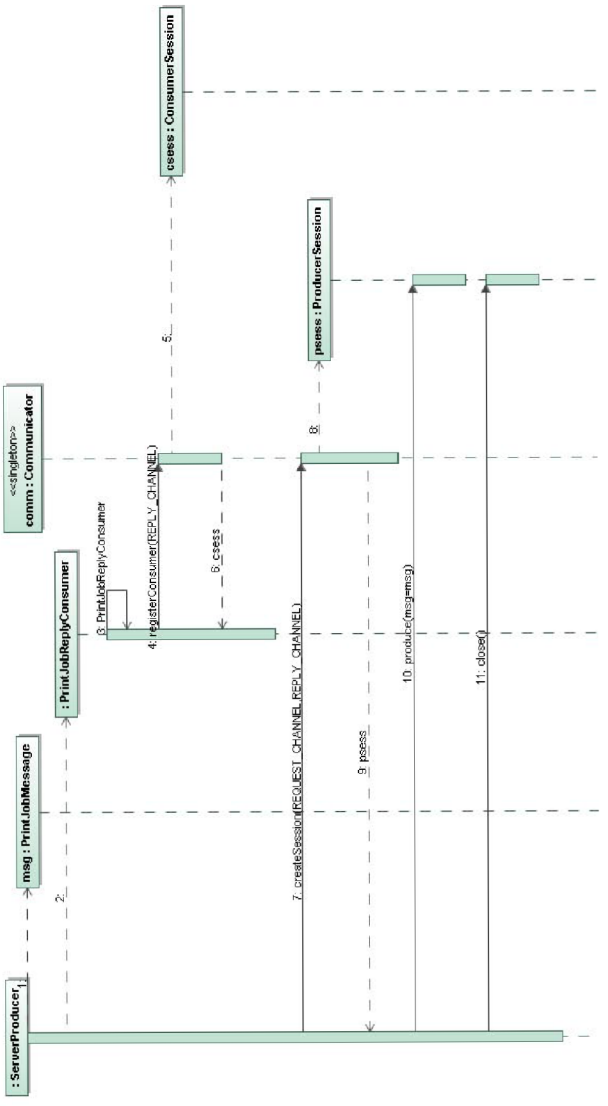
Obrázek D.2: Instalační aplikace na MFD



Obrázek D.3: Webová instalační aplikace Web Image Monitor

Příloha E

Použití API komunikátoru



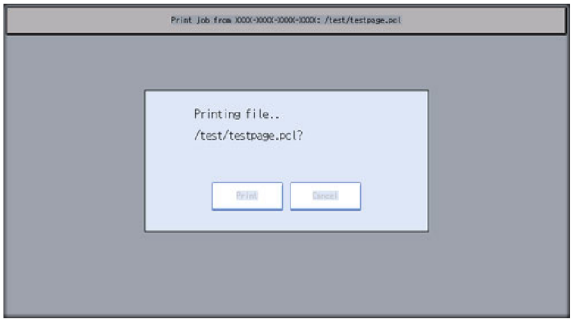
Obrázek E.1: Sekvenční diagram použití aplikačního rozhraní komunikátoru

Příloha F

GUI vestavěného terminálu

(a) Menu vestavěného terminálu

(b) Fronta úloh odloženého tisku



(c) Potvrzení tisku zvolené úlohy

Obrázek F.1: Uživatelské rozhraní