Porovnání knihoven pro vzdálené volání procedur vhodných pro embedded systémy

Ondřej Sedláček

Bakalářská práce
2024

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2023/2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Ondřej Sedláček**
Osobní číslo:	**A21276**
Studijní program:	**B0613A140020 Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Porovnání knihoven pro vzdálené volání procedur vhodných pro embedded systémy**
Téma práce anglicky:	**Comparison of Libraries for Remote Procedure Call Suitable for Embedded Systems**

Zásady pro vypracování

1. Vypracujte literární rešerši na dané téma.
2. Popište vlastnosti eRPC a Nanopb knihovny (architektura, komunikační vrstvy, implementace).
3. Vyhodnoťte kritické faktory pro volbu mezi eRPC a Nanopb a navrhněte kritéria pro jejich porovnání.
4. Implementujte vzorové aplikace pro obě knihovny a vytvořte testovací scénáře typické pro embedded aplikace.
5. Ověřte funkci knihoven na vytvořených vzorových aplikacích a vyhodnoťte výsledky jejich porovnání.

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

Seznam doporučené literatury:

1. AIMONEN, Petteri. *Nanopb – protocol buffers with small code size*. [online]. c2011 [citováno 2023-11-10]. Dostupné z: https://jpa.kapsi.fi/nanopb.
2. NXP. *EmbeddedRPC / erpc*. [online]. c2016-2023 [cit. 2023-11-10]. Dostupné z: https://github.com/EmbeddedRPC/erpc.
3. NXP. *LPCXpresso55S69/55S28 Development Boards, User manual* [online]. Rev. 1.6. 2022 [citováno 2023-11-10]. Dostupné z: www.nxp.com.
4. NXP. *LPC55S6x, Product data sheet* [online]. Rev. 2.4. 2022 [citováno 2023-11-10]. Dostupné z: www.nxp.com.
5. TANENBAUM, Andrew S. a VAN STEEN, Maarten. *Distributed systems: principles and paradigms*. 2nd ed. Upper Saddle River: Pearson Education, 2007. ISBN 978-0132392273.

Vedoucí bakalářské práce: **Ing. Petr Dostálek, Ph.D.**
Ústav automatizace a řídicí techniky

Konzultant bakalářské práce: **Ing. Michal Princ, Ph.D.**

Datum zadání bakalářské práce: **5. listopadu 2023**
Termín odevzdání bakalářské práce: **13. května 2024**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
DĚKAN
L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 5. ledna 2024

Prohlašuji, že

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

Prohlašuji,

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

Ve Zlíně, dne 09. 05 .2024

Ondřej Sedláček, v. r.
podpis studenta

ABSTRAKT

Cílem bakalářské práce je prozkoumat možnosti knihoven použitelných pro vzdálené volání procedur a vhodných pro použití v embedded systémech. Konkrétně se zaměřením na komunikaci mezi jádry. Zejména tedy provést komplexní analýzu specializované knihovny eRPC a knihovny Nanopb, která sama o sobě funkčnost vzdáleného volání procedur nezastává, ale posloužila jako stavební kámen pro toto paradigma. Bylo nutné a otestovat aplikace implementované v obou knihovnách a na základě nasbíraných dat vyhodnotit jejich benefity a nedostatky.

Klíčová slova: eRPC, Nanopb, vzdálené volání procedur, knihovny, embedded, porovnání, multi-core, distribuované systémy

ABSTRACT

The aim of the bachelor's thesis is to explore the possibilities of libraries usable for remote procedure calls and suitable for use in embedded systems, with a specific focus on inter-core communication. In particular, to perform a comprehensive analysis of the specialized eRPC library and the Nanopb library, which does not itself provide the functionality of remote procedure calls but served as a building block for this paradigm. It was necessary to test applications implemented in both libraries and evaluate their benefits and drawbacks based on the collected data.

Keywords: eRPC, Nanopb, remote procedure call, libraries, embedded, comparison, multi-core, distributed systems

Tímto bych chtěl poděkovat panu Ing. Petru Dostálkovi, Ph.D., a panu Ing. Michalu Princovi, Ph.D., za jejich odborné rady, které mi usnadnili psaní této práce.

Dále pak děkuju rodině za to, že mi umožnila studium.

Prohlašuji, že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

Koncepce distribuovaných systémů v softwaru je díky svým benefitům nezbytnou součástí dnešního světa. Ať už se jedná o vývoj webových aplikací, cloudových služeb nebo embedded systémů, každé jednotlivé odvětví má v zásadě trochu rozdílné požadavky na funkcionalitu. Tím pádem vzniká potřeba pro specializované knihovny, které jsou navrženy tak, aby těmto požadavkům dokázaly vyhovět.

Při vývoji těchto knihoven se často k návrhu různých komponentů přistupuje stejným způsobem, a to i přes značné rozdíly v charakteristikách užití, pro které jsou navrhovány. Je tedy častým jevem, že různé knihovny sdílí podobné cíle a požadavky na jejich vlastnosti. To vyvolává otázku, do jaké míry je lze kombinovat a čerpat z již realizovaných konceptů využitých v ostatních knihovnách. Tato otázka se ještě více nabízí v situacích, ve kterých je software navržený pro stejné prostředí. V těchto případech často různé knihovny kladou důraz na řešení stejné problematiky.

Paradigma vzdáleného volání procedur je díky svojí efektivnosti rozšířené do mnoha prostředí. Jedním z nich je použití v embedded systémech. Vývoj softwaru pro embedded aplikace představuje v mnoha ohledech, výzvu. Je nutné brát v úvahu všechna omezení, které může konkrétní platforma představovat. Tato práce zkoumá použitelnost softwaru právě ve zmíněném prostředí. Přesněji řečeno to, do jaké míry dokáže nástroj vyvinutý pro embedded systémy obstát při použití, ke kterému nebyl přímo směrován. Hlavně tedy jestli klady spojené s návrhem pro prostředí dokážou předčít zápory vzniklé z obecnějšího zamýšleného použití nástroje.

Cílem bakalářské práce je provést komplexní analýzu nástrojů použitelných pro vzdálené volání procedur v embedded systémech. Přesněji tedy knihovny Nanopb, která nebyla pro RPC přímo určena a specializované knihovny eRPC. Hlavním aspektem je zjistit, do jaké míry dokáže Nanopb obstát ve scénáři, pro který byla navrhnuta eRPC a zdali má takové použití význam. Toho bude docíleno pomocí testování vlastností obou knihoven za stejných podmínek.

I. TEORETICKÁ ČÁST

1 VZDÁLENÉ VOLÁNÍ PROCEDUR

Základem komunikace mezi distribuovaným systémem je výměna zpráv, které nějakou formou předávají informace potřebné pro správný chod. Zde ale nastává problém, protože samotné procedury pro přijetí a odeslání zprávy žádným způsobem komunikaci mezi distribuovaným systémem neukrývají. To není ideální, protože by distribuované systémy měly dosáhnout takzvané transparentnosti přístupu. [1]

Cílem transparentnosti je schovat skutečnost, že vůbec probíhá nějaká komunikace mezi systémy, tedy že procesy a zdroje jsou mezi nimi rozdělené. Dalo by se se tím pádem konstatovat, že pokud distribuovaný systém úspěšně schovává fakt, že se za ním ukrývá komplexní kombinace více systémů, dá se považovat za transparentní. [1]

Zmíněná transparentnost přístupu je typem transparentnosti, která se zaměřuje na skrytí rozdílů v reprezentaci dat a v uživatelském přístupu k datům. Je velmi běžné, že součástí distribuovaného systému budou počítačové systémy s rozdílnou architekturou nebo operačním systémem. Tím pádem by nemuselo docházet ke shodě způsobu, jak tyhle jednotlivé počítačové systémy budou reprezentovat data. Například by se mohlo stát, že rozdílné operační systémy budou mít vlastní konvence pojmenovávání souborů a způsob manipulace s nimi. To však není žádoucí, protože to značně komplikuje sdílení společných dat. Ideální tedy je najít způsob, jak celou tuto problematiku obejít, tak aby byla skryta před uživatelem a aplikacemi. [1]

Konkrétně tímto se zabývá paradigma Vzdáleného volání procedur, tedy umožněním komunikace mezi distribuovanými systémy tak, aby byla dodržena transparentnost přístupu. Je zde snaha komunikaci mezi systémy co nejvíce přiblížit takovému volání procedury, aby připomínalo lokální volání procedury. [1]

1.1 Jednoduchá procedura RPC

Z charakteristiky RPC tedy vyplývá, že by komunikace měla napodobovat volání lokální procedury. Pokud chce programátor zavolat proceduru například pro čtení ze souboru, načte se rutina procedury z knihovny a vloží se do programu. Dalo by se tedy říct, že tato procedura představuje prostředníka mezi uživatelským kódem a operačním systémem. Programátor této operace dosáhne pouhým zavoláním odpovídající procedury. Je tedy patrné, že je zmíněné volání lokální procedury velmi nenáročné a celý její průběh zůstává před programátorem ukrytý. [1]

Pokud vznikne potřeba proceduru zavolat na dálku, bude ideální udržet, pokud možno co největší podobnost s transparentností, kterou lze vidět u volání lokální procedury. Zpracování výše zmíněné procedury čtení ze souboru jakožto procedury volané na dálku bude vypadat následovně. Bude zde klientská strana, která bude vysílat požadavek s případnými daty na zpracování. Dále strana serveru, která na základě volané procedury a přiložených dat či parametrů požadavek zpracuje a následně informuje klientskou stranu o úspěšném provedení nebo pošle zpracovaná data zpět uživateli. V tomto případě, ve kterém je pouze z klientské strany posílán požadavek na přečtení dat ze souboru na straně serveru, se očekávají přečtená data, protože samotné potvrzení o provedení procedury by pro klienta nemělo význam. [1]

Pokud je cílem pomocí určité procedury získat data nebo zajistit provedení služby od serveru, je potřeba najít vhodný způsob, jak se stranou serveru komunikovat. U lokální procedury je to docela snadné, ale u vzdálené zde se objevuje výše zmíněný problém spojený s možnou rozdílností komunikujících systémů. Vzdálené volání procedur tuto problematiku řeší tím způsobem, že namísto toho, aby komunikace se stranou serveru byla součástí procedury, tak používá na obou stranách knihovny, ve kterých je množina dohodnutých procedur. [1]

V praxi to pak funguje tak, že namísto konkrétní procedury je zavolaný klientský stub, tedy zástupce, který má za úkol zabalit požadavek, poslat ho na stranu serveru a následně pak zpracovat odpověď poslanou ze serveru. Tento zástupce bude z programátorského hlediska zavolaný podobně jako lokální procedura, ale namísto požadavku na operační systém o přečtení dat je samotné zavolání procedury zabaleno spolu s dodatečnými parametry do dohodnutého formátu a odesláno na stranu severu. Na straně serveru jej dostane na starost stub serveru, což je ekvivalent klientského stubu, který má za úkol zpracovat požadavek klienta, rozbalit ho a zavolat teď už pro server lokální proceduru. Po zavolání procedury se klientská strana zablokuje a čeká na odpověď ze strany serveru. Dá se tedy říci, že tím pádem na venek působí, jako by danou proceduru klientská strana provedla sama, protože během zpracovávání procedury stranou serveru je zaneprázdněna až do doby, kdy nemá k dispozici výsledek. [1]

Celé volání vzdálené procedury proběhne následovně: Klientská strana zavolá proceduru skrze klientský stub, ten zformuje zprávu a zavolá operační systém. Zpráva je následně poslána na stranu serveru, kde ji dostane stub serveru. Stub serveru rozbalí zprávu a zavolá ekvivalentní proceduru s přiloženými parametry, server pak provede určenou proceduru lokálně. Ve chvíli, kdy je procedura dokončená, předá server parametry zpět svému stubu,

který je zabalí a pošle je zpět na stranu klienta. Strana klienta čeká zablokovaná, dokud jí nedojde zpráva ze serveru. Tu následně předá stubu klienta, který zprávu rozbalí a předá výsledek procedury klientovi. [1]

2 GENERÁTORY RPC

V systémech využívajících Vzdáleného volání procedur jsou RPC generátory důležitým nástrojem, který má za úkol vygenerovat klientské a serverové stuby, jež pak slouží jako prostředníci mezi lokální a vzdálenou stranou. Tyto stuby mají za úkol serializaci a deserializaci požadavků a následných odpovědí tím způsobem, že konvertují potřebná data do formátu, který bude vhodný pro přenos mezi stranami, zatímco obsah zprávy zůstane snadno zpracovatelný na straně příjemce. Výhodou generování stubů automaticky je, že je programátor systémů využívajících RPC ušetřen od povinnosti psaní repetitivního kódu manuálně, což by za takových podmínek mohlo vést k většímu množství chyb programu a značně by to zvýšilo dobu vývoje. Automatické generování tedy nejen usnadňuje vývoj, ale díky tomu, že abstrahuje komplexnost jazykové a strojové heterogenity, navíc umožňuje bezproblémovou komunikaci v různých programačních jazycích běžících na různých architekturách. Pokud jsou totiž komunikační komponenty napsány v rozdílných jazycích, nebo jsou založeny na rozdílných systémech, musí být program takový, aby jeho funkčnost nebyla rozdílností ovlivněna. [2]

2.1 Interface Description Language (IDL)

Ve většině případeů u systémů RPC zařizuje heterogenitu programu Interface Description Language (IDL). Ten v nich hraje kritickou roli, protože zprostředkovává možnost popsání rozhraní mezi vzdálenými komponenty způsobem, který je nezávislý na různých programovacích jazycích a platformách. [2] Narozdíl od ostatních programovacích jazyků nejde použít pro samostatné implementování rozhraní, ale pouze slouží jakožto forma pro její definování. [3] Použití IDL v praxi bude následující. Pokud bude třeba využít IDL pro definování funkcionality správy bankovního účtu, bude následující soubor IDL obsahovat parametry jako číslo účtu nebo aktuální stav účtu. Dále může obsahovat definice operací, jako třeba odeslání platby nebo vytvoření žádosti o platbu. Nicméně samotná funkce nebude implementována v IDL souboru, ale mimo něj v jazyce, který bude pro implementaci zvolen. [3]

Mimo základní definice datových typů proměnných a operací IDL často umožňují i rozšířené vlastnosti jako například jednocestné operace, které umožňují asynchronní komunikaci mezi komponenty, kde volající strana není blokovaná, zatímco čeká na odpověď. [3] Podobné funkce však nemusejí být pravidlem a liší se podle druhu IDL.

2.1.1 Jednoduchá definice rozhraní IDL

IDL rozhraní zprostředkovává popis funkcionality objektu takovým způsobem, aby z něj bylo možné vyprodukovat program, který dané rozhraní využije. Pokud teda bude potřeba rozepsat podrobnosti výše zmíněného příkladu bankovního účtu, bude třeba aby rozhraní obsahovalo popis operací a parametrů samotného objektu. [3]

```
// IDL
interface Account {
       // Attributes to hold the balance and the name
       // of the account's owner.
       attribute float balance;
       readonly attribute string owner;

       // The operations defined on the interface.
       void makeDeposit(in float amount,
                        out float newBalance);
       void makeWithdrawal(in float amount,
                        out float newBalance);
};
```

Obrázek 1. Příklad IDL rozhraní [3]

Jak je znázorněno na obrázku 1. rozhraní účtu v první řadě obsahuje parametry objektu „balance“ s datovým typem float a dále parametr „owner“ datového typu string.[3]

Je tedy zjevné, že se samotná definice u většiny IDL jazyků drží osvědčených syntaxí, které jsou navržené tak, aby byly pro programátora intuitivní a bylo snadné tento soubor obsahující IDL rozhraní napsat snadno manuálně.

Dále rozhraní obsahuje definici dvou operací „makeDeposit“ a „makeWithdrawal“, které obsahují nejen standardní vstupní parametr s příslušným datovým typem, jak by tomu bylo u standardní definice funkce v běžném programovacím jazyce, ale i výstupní parametr ve stejném formátu. Parametry operace jsou označeny klíčovými slovy „in“ a „out“ podle charakteristiky směru, ve kterém budou posílány například mezi klientskou stranou a stranou serveru. Pokud bude tedy rozhraní z obrázku 1. použito v sytému RPC, bude operace „makeWithdrawal“ realizována tak, že klientská strana pošle zprávu, která bude mimo ostatní data obsahovat parametr „amount“ datového typu float. Klientská strana bude následně očekávat zprávu, která bude obsahovat parametr „newBalance“ stejného datového typu. [3]

Mimo jiné klíčová slova „in“ a „out“ lze v určitých případech použít také klíčové slovo „inout“. Toto slovo označuje parametry v případě, že je očekávané, že parametr bude použitý obousměrně.

Všechna zmíněná klíčová slova mají mimo účel sloužící pro zajištění správné funkčnosti programu, do kterého je IDL soubor přeložen, také vlastnost samostatné dokumentace. Jinými slovy lze pomocí těchto klíčových slov pro programátora snadno určit který parametr slouží k jakému účelu a tím zvýšit přehlednost napsaného kódu jednoduchým způsobem. [3]

2.1.2 Jednocestné operace

Za normálních okolností je při volání vzdálené procedury klientská strana blokována pod dobu vykonávaní operace ze strany serveru, proto se velmi nabízí definovat v IDL rozhraní operaci, která se tomuto blokování v případě potřeby vyhne. Taková operace potom nemůže obsahovat parametr s klíčovým slovem „out“ nebo „inout“, protože klientská strana na žádnou odpověď ze strany serveru nečeká.

2.1.3 Datové typy a struktury

Většina jazyků pro vývoj rozhraní IDL běžně používá standardní datové typy. Mezi ně patří celočíselné typy jako long nebo short, typy s pohyblivou desetinnou čárkou jako float a double, a samozřejmě také typy char a boolean. Větší variace pak vzniká u datového typu string, který může nebo nemusí být vázaný v závislosti na konkrétní specifikaci. Mimo standartní typy jdou však jako parametr operace použít struktury, které svazují dohromady více atributů a vytváří tak praktičtější balíček potřebných dat.

2.2 Generátory stubů

Stuby jsou v distribuovaných systémech kódové entity, které slouží jako lokální zástupci vzdálených a volajících komponent, tedy strany klienta a serveru, jsou tím pádem nedílnou součástí vzdáleného volání procedur. Zodpovědností stubů je konverze, čili serializace a deserializace mezi různými formáty vhodnými buď pro zpracovávání obsažených dat nebo pro jejich přenos. Dále pak mají na starost přenos samotných dat po síti. [2]

Stuby jsou vytvářeny takzvaným Generátorem stubů. Moderní generátory stubů nejprve samy určí, zda je možná konverze reprezentace formátu a také ověří kompatibilitu rozhraní procedury. Stuby jsou pak konstruovány pomocí jazykově závislých šablon, které zachycují podstatnou syntaktickou a sémantickou strukturu datových typů v podporovaném jazyce. Samotný generátor stubů je nezávislý na jazyce.[2]

Generátor stubů přijímá od parserů popisy rozhraní procedur, které jsou zpracované a jako výstup vygenerují stuby a rutiny pro serializaci a deserializaci. Pro odvození funkčnosti a kontroly kompatibility jsou použity příslušné tabulky. Generátor stubů využívá těchto tabulek a šablonování zdrojového kódu k vytváření stubů. Šablony jsou prezentovány jako procedury a šablony pro serializaci a deserializaci ukazatelů. Ty obsahují rekurzivní algoritmus pro průchod strukturou založenou na ukazatelích, který zpracovává jak mezistrukturální, tak vnitrostrukturální aliasy a cykly. Pro snížení režie volání procedury může být kód šablony vložen do stubu.[2]

2.2.1 Tabulka kompatibility typů

K posouzení kompatibility typů parametrů a jejich odvození slouží Tabulka kompatibility typů. Je to struktura dat, která obsahuje informace o tom, jaké datové typy jsou kompatibilní mezi různými programovacími jazyky. Pro každý pár programovacích jazyků existuje v tabulce záznam, který obsahuje kompatibilní skalární typy nebo konstruktory typů. Jako kompatibilní typy jsou pak definovány ty, které reprezentují stejný abstraktní typ. Například typy odpovídající abstraktnímu typu integer, jako jeC int, Fortran INTEGER a Pascal integer, jsou definovány jako kompatibilní. V rámci stejného jazyka může existovat více typů odpovídajících stejnému abstraktnímu typu. Tato tabulka je tedy využívána jako nástroj k posouzení, zda jsou typy parametrů u různých programovacích jazyků kompatibilní. Tabulka se dále používá v situaci, ve které je potřeba převést rozhraní procedury definované v jednom nativním jazyce na odpovídající rozhraní v jazyce účastníka na druhé straně. Tabulka tak musí najít kompatibilní typ pro každý parametr v prostředí. Může se stát, že bude tabulka obsahovat více možností vhodných typů, v takovém případě bude vybrán typ, který nevyžaduje žádnou konverzi formátu. [2]

2.2.2 Tabulka mapování typů

Další z tabulek je takzvaná Tabulka mapování typů. Tato tabulka obsahuje data o tom, jak jsou jednotlivé datové typy mapovány na konkrétní formáty dat v rámci kompilátorů podporovaných v dané síti. Všechny záznamy obsažené v tabulce obsahují informace vztahující se k mapování konkrétních datových typů na konkrétní formáty dat, které jsou používány při kompilaci a provádění programu v daném prostředí. [2]

Tato tabulka slouží jako nástroj pro generátor stubů a zajišťuje správné konvertování formátů dat mezi různými programovacími jazyky a architekturami komponentů distribuovaných systémů. Tabulka mapování typů je použita při generování stubů proto, aby bylo z informací, které obsahuje, správně určeno, jaký formát dat nutné použít pro daný datový typ v konkrétním prostředí. Tím je zajištěno, že jsou data při přenosu v distribuovaném systému správně interpretována a nedojde během předávání k žádné ztrátě informace nebo chybné interpretaci. [2]

Obsah tabulky zahrnuje klíčové informace o reprezentaci datových typů v různých rozhraních a o tom, jakým způsobem je nutné provádět konverze mezi formáty dat při komunikaci mezi různými komponenty napsaných v rozdílných jazycích a fungujících na rozdílných hardwarových platformách. Díky tabulce mapování typů je zajištěna efektivní komunikace mezi heterogennimi částmi distribuovaného systému. [2]

2.2.3 Šablony

Šablony jsou v kontextu struktury generátoru stubů segmenty textu, které obsahují přesné definice struktur jednotlivých druhů stubů, dále pak rutin každého skalárního typu a typu konstruktoru pro jejich serializaci a deserializaci. Také obsahují rutiny serializace a deserializace pro běžně používané složené typy jako jsou například jednorozměrná pole. Každý jazyk má svůj vlastní soubor šablon, které obsahují syntaxi daného jazyka a také informace o standardních knihovnách, algoritmech a deklaracích specifických pro něj. [2]

Generátor stubů používá tyto šablony po generování konkrétních stubů specifických pro daný jazyk. Při jejich generování jsou informace rozhraní procedury použity k výběru a substituci příslušných šablon. Tyto šablony tak umožňují serializaci a deserializaci libovolného typu, včetně ukazatelových parametrů nebo rekurzivních parametrů, a to i přesto, že struktura těchto parametrů není až do běhu samotného programu známa. Tímto způsobem je dosaženo jednoduchosti návrhu generátoru stubů a zároveň je zajištěna její flexibilita a nezávislost na konkrétním programovacím jazyku. [2]

Během procesu generování stubů umožňuje šablona proces podobný macro expanzi. Toho lze docílit pomocí speciálních proměnných, které jsou během generování nahrazeny příslušnými kódy. Díky tomu umožňuje vnořenou substituci šablon. Jestli nastane situace, ve které bude třeba nahradit proměnnou v šabloně konverzní rutinu pro formát skalárního typu, tak generátor stubů vyhledá referenci na tuto rutinu v tabulce konvertibility a následně příslušnou substituci provede. Šablony tedy generátoru stubů umožňují konzistentní generování

kódu pro komunikaci mezi komponenty v distribuovaném systému i přesto, že jsou napsány v rozdílných programovacích jazycích.[2]

3 PROTOCOL BUFFERS

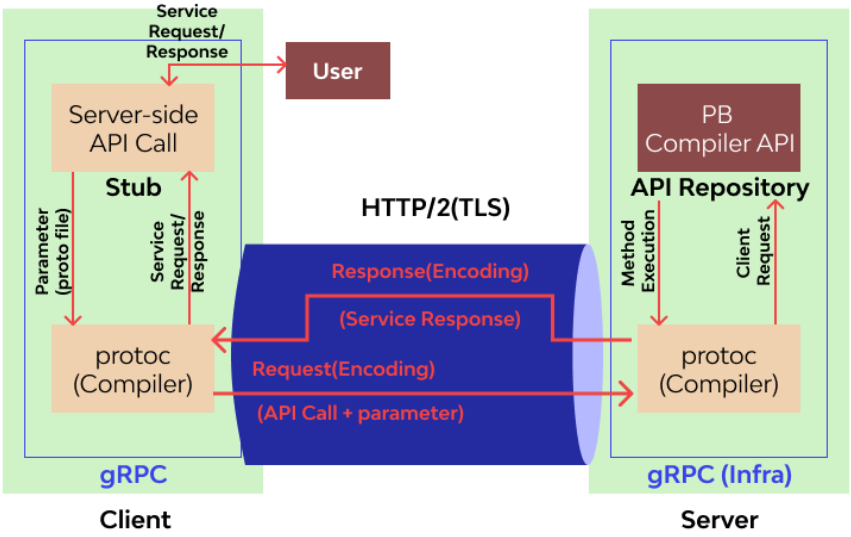
Protocol Buffer, často také známý jako „Protobuf“, je formát serializace dat, který je využívaný zejména kvůli svojí vlastnosti efektivního převodu dat do binární podoby. Tento formát je navržen tak, aby data byla co možná nejkompaktnější a co nejsnadněji přenosná po síti. Protocol Buffer datová struktura je definovaná pomocí souborů s příponou .proto ve kterých je obsažen popis datových struktur ve formátu, který je pro programátora snadno čitelný a snadno upravitelný v případě potřeby. Tento formát je podobný XML, ale je obvykle efektivnější a rychlejší díky použití binární reprezentace dat. Protocol Buffer umožňují serializaci dat z různých programovacích jazyků a platforem do podoby bytových streamů. Díky tomu přenáší výrazně menší velikost dat než ostatní formáty. [4]

Protocol Buffer slouží nejen jako samotný formát dat, ale také zahrnuje popis struktury dat a jejich služeb. V souborech s příponou .proto může Protocol Buffer mimo definice datových struktur popisovat také rozhraní a jednotlivé metody služeb. Díky tomu se jeví jako vhodný nástroj pro vytváření metod Vzdáleného volání procedur, pro komunikaci mezi stranou klienta a stranou serveru. Díky svojí rychlosti, nízké náročnosti ze strany velikosti dat a možnosti použití v různých prostředích je Protocol Buffer velice efektivní a flexibilní prostředek pro serializaci a deserializaci dat do binární podoby. [4]

3.1 Protocol Buffers v RPC

Příkladem použití Protocol Buffers ve Vzdáleném volání procedur je velmi rozšířená knihovna vyvinutá společností Google jménem gRPC. Tato knihovna se zaměřuje na využití paradigmatu RPC, ale narozdíl od konkurenčních knihoven používá koncept Protocol Buffers namísto běžných nástrojů pro serializaci a deserializaci přenášených dat. Díky tomu knihovna dosahuje všech výše zmíněných benefitů. Například pokud je gRPC porovnáno s komunikací skrze REST+JSON, dosahuje gRPC desetinásobné rychlosti, čehož je dosaženo kombinací HTTP/2 a právě Protocol Buffers.[5]

Protocol Buffer v gRPC ve většině případů využívá novější IDL jazyk proto3, který má zjednodušenou syntaxi a podporuje širokou řadu jazyků. [6]



Obrázek 2. Architektura gRPC [5]

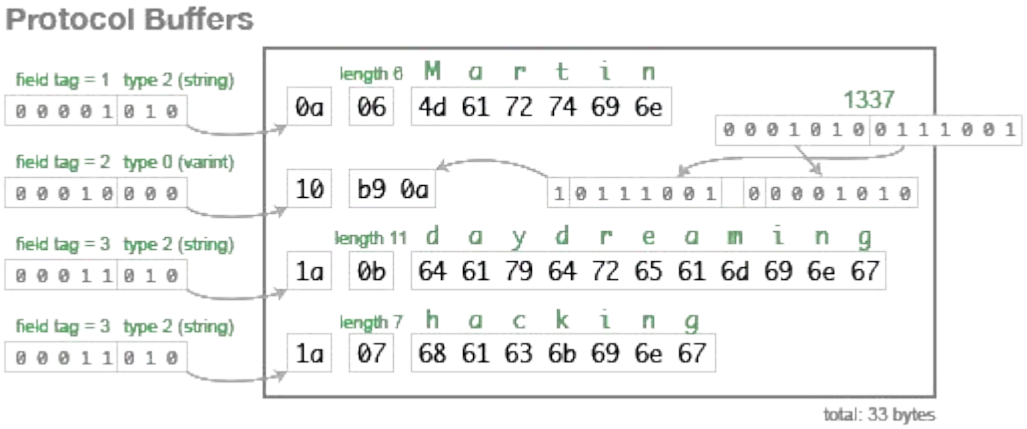
3.2 Použití Protocol Buffers

Pro použití Protocol Buffers je potřeba nejprve vytvořit IDL soubor .proto. Tento soubor obsahuje definice datových struktur a zpráv a slouží k definování schématu dat, které budou serializovány a deserializovány pomocí Protocol Buffers. Tyto zprávy ve formě bytových streamů jsou pak dále rozesílány v distribuovaném systému. V souboru jsou obsaženy názvy a parametry dané zprávy a jejich datové typy a také čísla polí, které připomínají standardní strukturu s tím rozdílem, že každý parametr má přiřazený svůj vlastní klíč v podobě čísla pole, který je označený číslem v desítkové soustavě podle pořadí, ve kterém je parametr definován. Jakožto standartní IDL definuje .proto soubor také služby neboli operace, které mají definovaný vstupní parametr, jež by se dal přirovnat k parametru funkce, a výstupní parametr, který představuje očekávanou odpověď, tedy výstup funkce. V tomto případě nebudou vstupy a výstupy jednoduché datové typy, ale budou je představovat celé zprávy. Na úvodu .proto souboru je uveden používaný syntax, například proto2 nebo proto3.[4]

3.3 Obsah byte streamu

Ve zprávách serializovaných pomocí Protocol Buffers jsou data reprezentována pomocí byte streamů. Zmíněný stream obsahuje sérii páru uložených za sebou v řadě, ve kterém je jednou částí z páru klíčové pole a druhou je samotná hodnota, která reprezentuje přenášenou informaci. Binární verze zprávy používá číslo pole jakožto klíč. Konkrétní název a deklarovaný

typ lze určit až při deserializaci na straně příjemce, která má odkaz na definici .proto souboru. Pokud má tedy strana příjemce k dispozici soubor definicí, tak ví, že první hodnota zprávy bude označená číslem 0 a bude ekvivalentní k parametru označeným v definičním souboru stejným číslem 0. Podle toho dokáže určit název parametru. Při deserializaci zprávy parser musí přeskočit pole, které nerozpoznává. Tímto způsobem lze do zprávy přidávat nové pole, aniž by se porušily staré programy, které o nich nevědí. Klíčem pro každý pár ve zprávě jsou dvě hodnoty, číslo pole z .proto souboru a datový typ. Datový typ poskytuje informace o délce parametru, díky čemuž lze snadno najít jeho konec a tím pádem lokaci dalšího parametru.[4]



Obrázek 3. Reprezentace dat Protocol Buffers[23]

4 MULTI-CORE SYSTÉMY

V dnešním světě se zvyšující náročností softwaru kladeny čím dál větší požadavky na výkon hardwaru. U procesorů toho lze docílit tím způsobem, že se výrobci budou snažit zvýšit počet tranzistorů, který je možné umístit na 1 mm čtvereční plochy procesoru. Tím se dá spolehlivě zvýšit výkon, ale vytváří to i řadu problémů. Jedním z těchto problémů je teplota, která s větším množstvím tranzistorů na čipu pochopitelně roste, což zvyšuje nároky na jejich chlazení. I přesto, že se způsoby chlazení procesorů vyvíjí také, díky čemuž roste jejich efektivita, nestačí držet krok s rychlostí zvyšování počtu tranzistorů na jednom čipu.[7]

Zvláště v embedded systémech se chlazení jeví jako problém, protože jeden z častých požadavků s vyšší náročností na provedení je právě velikost. Jinými slovy, aby bylo celé zařízení co nejmenší, což častokrát neumožňuje volný průtok vzduchu a ani instalaci objemných chladících zařízení.

Vícejádrový procesor je integrovaný obvod, který je složen ze dvou nebo více procesorů. Tento koncept umožňuje vyšší výkon, nižší energetickou náročnost, a hlavně pak možnost paralelního zpracovávání úloh.[7]

4.1 Paralelní zpracování

Paralelní zpracování je metoda zpracovávání, ve kterém je k vyřešení jedné úlohy využito současně více procesorů nebo jader. Tento přístup rozděluje jednu komplexní úlohu na menší části, které jsou následně zpracovávány souběžně. Hlavním cílem je zkrátit dobu zpracování a tím urychlit oproti sériovému zpracování dobu dokončení dané úlohy. Kromě samotného zvýšení výpočetního výkonu nabízí paralelní zpracování také výhodu v podobě vyšší propustnosti a lepší odolnosti proti chybám. Mimo to představují systémy využívající paralelního zpracování také lepší poměr cena/výkon, než jak je tomu u sériových. [8]

Ideálně by měla být křivka nárůstu výkonu v poměru s počtem použitých jader procesoru lineární. V praxi je tomu bohužel jinak a nárůst je často omezen díky vyšší náročnosti na výkon spojené s paralelizmem, tedy výpočetní složitostí rozdělování určené úlohy mezi procesory. Důležitou roli hraje také to, jak dobře je samotná aplikace přizpůsobena k paralelnímu zpracování. Efektivita tak z velké části závisí na tom, jak snadno jdou úkoly rozdělit na části, které lze paralelně zpracovávat. Pokud má aplikace malý nebo žádný vrozený paralelizmus, tedy míru, do jaké ji lze rozdělit na části zpracovávatelé paralelně, pak je možné dosáhnout pouze malého nebo dokonce i žádného zrychlení. [8]

4.1.1 Typy paralelismu

Schopnost programu provádět úlohy v paralelním pořadí, tedy paralelismus programu, se dělí na tři kategorie. Jsou jimi: paralelismus na úrovní instrukcí, paralelismus na úrovni vláken a paralelismus na úrovni dat. V případě paralelismu na úrovni instrukcí mohou být samotné instrukce prováděny jak paralelně, tak sekvenčně. U paralelismu na úrovni vláken jsou různá vlákna té samé úlohy předávaná procesoru pro simultánní provádění. Paralelismus na úrovni dat pak pojednává o společných datech sdílených mezi prováděnými procesy. Tento paralelismus umožňuje redukovat čas, který by byl spotřebován přistupováním k paměti a k jejímu načítání. Díky tomu zvyšuje výkon.[7]

4.2 Řešení paměti v multi-core systémech

Architektura paměti, zvláště pak v embedded multi-core systémech, má značný vliv na výkon, spotřebu energie a obecnou efektivity celého systému. Z hardwarového hlediska zabírá nejvíce prostředků právě paměť, a to vyžaduje zvláštní pozornost při její organizaci na čipu. [9]

Paměťová organizace v embedded multi-core systémech se z pravidla dělí podle svého zaměření na daný typ trhu pro který je určen, tedy liší se podle svojí specializace. Obecně lze však tyto paměti rodělit na primární a sekundární.[9]

Primární paměť je paměť, jež adresují jádra a která obsahuje aktuální sadu dat. Tato data jsou zpracovávána stejně jako programový kód. Primární paměť může zahrnovat hlavní paměť, která je obvykle realizována technologií DRAM, a hierarchii menších a rychlejších vyrovnávacích pamětí SRAM nebo pamětí typu Scratchpad SPRAM, které se liší v tom, že obsahují kopie některých dat z hlavní paměti. [9]

Sekundární paměť, což je třeba flash paměť, slouží v embedded systémech pro dlouhodobé ukládání dat. Jako příklad lze uvést ukládání obrázků v digitálních fotoaparátech.[9]

4.2.1 Koherence sdílené paměti

Jedním způsobem, kterým mezi sebou můžou jednotlivá jádra komunikovat je sdílení paměti. To však představuje při návrhu hierarchie paměti komplikaci, protože není snadné se vyhnout problému s koherencí paměti. Vzhledem k tomu, že má každé jádro svoji vlastní mezipaměť, tak se může stát, že ne všechny kopie dat v nich budou nejaktuálnější. Pro příklad je dobré si představit procesor se dvěma jádry, kde každé jádro načetlo blok do své

privátní mezipaměti. Když jedno jádro uloží hodnotu na určité místo a druhé jádro se pokusí tuto hodnotu přečíst ze své mezipaměti, dostane verzi dat, která už není aktuální, pokud není jeho záznam v mezipaměti invalidován a nedojde k chybě mezipaměti. Tato chyba by měla přimět druhý záznam mezipaměti jádra k aktualizaci. Zmíněná situace však může způsobit problémy s korektností vykonávané aplikace. Systém se dá považovat za koherentní pouze tehdy, pokud všechny kopie určité lokace hlavní paměti, které se nachází v mezipaměti jader zůstávají konzistentní a aktualizované po změně obsahu této paměťové lokace. Toho lze dosáhnou pomocí takzvaných protokolů koherence, které mají za úkol koherenci udržovat. Udržovaní koherence představuje provádění speciálních akcí v případě, pokud je přepsán obsah lokace, který se zároveň nachází jakožto kopie v mezipaměti ostatních jader. [9]

5 KNIHOVNA ERPC

Knihovna eRPC (Embedded Remote Procedure Call) je knihovna vyvinutá společností NXP. Tato knihovna se od ostatních běžně používaných knihoven pro vzdálené volání procedur liší tím, že je optimalizována pro prostředí, kde jsou komponenty navzájem těsně propojené. eRPC využívá pro své vzdáleně volané funkce čistě jazyk C a má velmi malou velikost kódu, která nepřesuje 5 kB. Díky svým vlastnostem nabízí eRPC nepřehlédnutelné benefity při použití v oblasti multi-core systémů, k čemuž byla také v první řadě knihovna navržena. [10] Knihovna eRPC je použitelná pro komunikaci v podstatě v jakémkoliv prostředí, ve kterém je strana serveru reprezentována procesorem připojeným libovolným komunikačním kanálem. Není však navržena pro vysoce efektivní komunikaci distribuovaného systému realizovaného skrze síť. [10][11]

5.1 IDL knihovny eRPC

Knihovna eRPC využívá IDL jako prostředek, který lze použít jako vstup do jejího generátoru erpcgen. Autor programu v něm pomocí intuitivních pravidel sestaví struktury zpráv a nastaví logiku komunikace pro vzdálené volání procedur. [12]

5.1.1 Základní syntaxe

IDL použité v eRPC knihovně se drží podobných pravidel, kterými se řídí programovací jazyk C, případně C++. Například identifikátory musí začínat základním písmenem abecedy nebo podtržítkem a můžou používat alfanumerické znaky a podtržítka. [12]

Co se týče literálů, podporuje IDL tři typy hodnot: integer, floating point a string. Integer může obsahovat předponu, který udává soustavu, jakou hodnota reprezentuje, pro hexadecimální soustavu je to předpona ‘0x’ a pro binární je jím ‘0b’, základní soustava je desítková, která předponu nepotřebuje. Dále je možné opatřit integer hodnotu jednou z přípon, jsou jimi standartně U, UL a ULL a narozdíl od programovacích jazyků C a C++ nejsou tyto přípony case-sensitive, tedy lze je psát malými písmeny bez rozdílu. [12]

Co se týče typu floating point, opět se od standardního zápisu liší pouze tím, že není case-sensitive, jinak podporuje exponent, který se skládá z písmene ‘E’, případného znaménka +/- a decimálního čísla. [12]

Stringy se pak od zápisu v C/C++ neliší vůbec nijak, musí být uchovány v uvozovkách, podporován je i běžný znak ‘\’ který umožňuje zápis escape sekvencí. [12]

Tabulka 1. Operátory podporované IDL knihovny eRPC [12]

Operátor	Funkce	Operátor	Funkce
+	sčítání	^	binární XOR
-	odčítání	<<	bitový posun doleva
*	násobení	>>	bitový posun doprava
/	dělení		
%	modulo	+	žádná operace (unární)
&	binární AND	-	záporná hodnota (unární)
|	binární OR	~	binární inverze (unární)

5.1.2 Datové typy a jejich použití

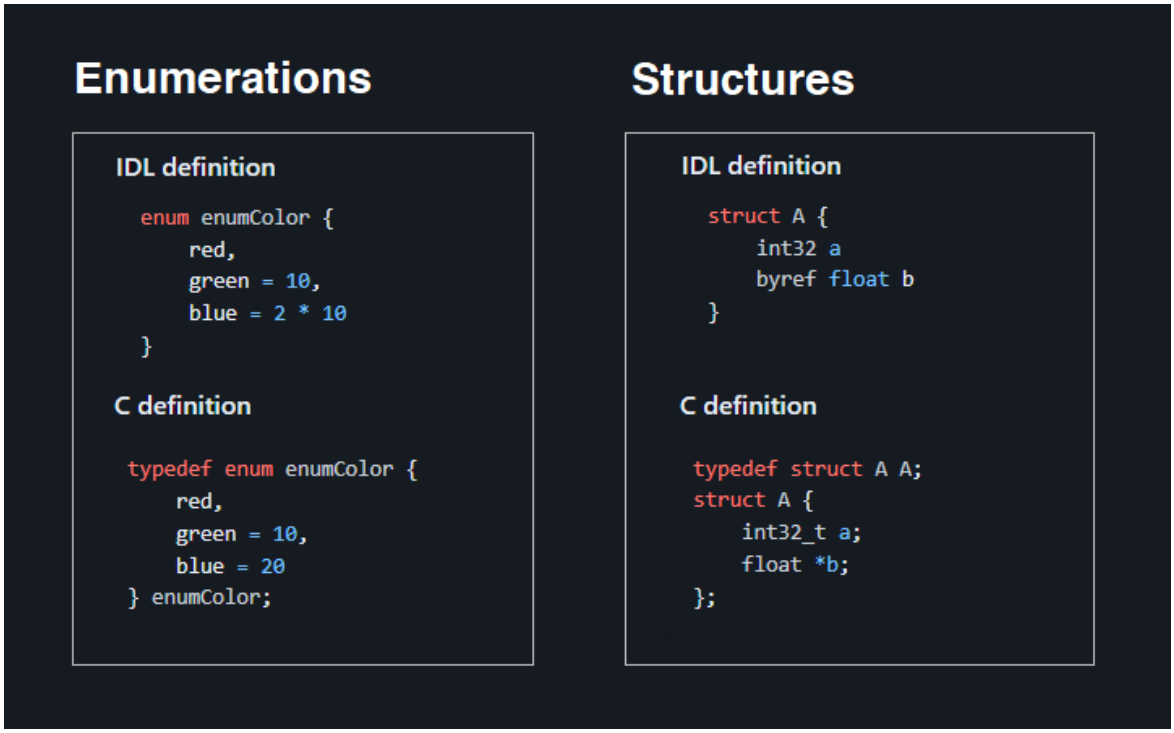
Ve většině případů je nutné do zprávy krom požadavku na provedení konkrétní procedury zahrnut i data, se kterými má daná procedura pracovat. Díky tomu podporuje knihovna eRPC celou řadu základních datových typů, které vychází z obecně rozšířené charakteristiky.

Tabulka 2. Porovnání podporovaných datových typů IDL eRPC [12]

IDL	C	Java	IDL	C	Java
bool	bool	Boolean	uint8	uint8_t	short*
int8	int8_t	byte	uint16	uint16_t	int*
int16	int16_t	short	uint32	uint32_t	long*
int32	int32_t	int	uint64	uint64_t	N/A
int64	int64_t	long	float	float	float
string	char*	String	double	double	double

Pokud je potřeba zorganizovat množinu základních datových typů do formy, ke které lze přistupovat jednotně, umožňuje IDL použití struktur. Struktury jsou v IDL tvořeny pomocí klíčového slova „struct“ s následným názvem struktury, podle tohoto názvu je pak dále struktura referencována. Narozdíl od definice v jazyce C se však nepoužívá slovo „typedef“. Pro libovolného člena struktury je možné použít klíčové slovo „byref“, to je vhodné, pokud se chceme vyhnout kopírování dat. Použití „byref“ umožní generátoru serializovat člena struktury pomocí reference. [12]

Další podporovaný typ datové struktury je enumerace, ta opět používá zjednodušený zápis oproti zápisu v jazyce C, kde je vynecháno klíčové slovo typedef a jméno enumerace již není potřeba uvádět na konci definice. [12]



Obrázek 4. Zápis struktur a enumerací v IDL eRPC

Mimo struktury podporuje IDL použití polí a seznamů. Co se týče polí, tak se jeho velikost uvádí v hranatých závorkách hned za datovým typem narozdíl od zápisu v jazyce C, kde se velikost uvádí až za názvem pole. Podporované jsou i vícerozměrné pole. Vzhledem k charakteristice účelu IDL je vždy nutné pevně specifikovat velikost pole. Bez specifikace velikosti pole by nebyl korektně vygenerován kód pro manipulaci s ním. Není možné tedy s polem dynamicky manipulovat uprostřed programu. [12]

Vzhledem k tomu, že jazyk C sám o sobě nenabízí možnost použití seznamů, tak je potřeba, aby jej generátor vytvořil synteticky. Toho dosahuje pomocí kombinace ukazatele na pole a proměnné, která obsahuje počet prvků ve struktuře. Pro zápis v IDL „list<tnt32>“ tedy generátor vygeneruje strukturu, která bude obsahovat dva atributy. [12]

5.1.3 Anotace

Úloha anotací spočívá v doplňování dodatečných informací k prvkům v dokumentu. Tyto informace neslouží jako součást programu samotného, ale umožňují například konfiguraci

nástrojů, které s daným dokumentem pracují. [13] V kontextu IDL umožňují anotace upřesnění charakteristiky prvků pro generátor, tak aby výstup odpovídal požadavkům na vlastnosti kódu. [12]

Zápis anotací se skládá ze znaku „@“ a názvu anotace. Pokud generátor nerozpozná název anotace, bude ji ignorovat. IDL podporuje celou řadu anotací pro většinu prvků v souboru, každá z nich se musí zapsat ke svému příslušnému prvku pro který je určena. Například lze pomocí anotace „@max_length(value) “ nastavit maximální velikost pro string. [12]

Pokud je potřeba připnout anotaci k celému programu, je zde možnost použít klíčového slova „program“. Jeho použití v souboru není nutností a samo o sobě nezpůsobí žádné generování kódu, nicméně jeho použití je vhodné, pokud je potřeba například vypnout určité chybové hlášení, generovat crc hodnotu pro ověření kompatibility nebo nastavit použití sdílené paměti. [12]

5.1.4 Formulace požadavku

Definice datových typů a možného obsahu zpráv nemá význam bez definice samotných volatelných operací, tyto operace jsou v IDL souboru nazývány funkce. Existují dvě základní podoby definice funkce, pokud není potřeba aby strana klienta dostala od strany serveru odpověď, použije se definice bez návratové hodnoty, tedy jednocestné funkce. Ta je označena klíčovým slovem „oneway“. [12]

Pokud by měla funkce vracet hodnotu, je potřeba použít příslušný zápis s návratovým datovým typem. Jako návratový typ lze použít i void. Použití návratového typu void se od jednocestné funkce liší tím, že zatímco při použití funkce s návratovým typem void klientská strana čeká, při použití jednocestné funkce není klientská strana žádným způsobem blokována. Funkce se standardně zahrnují do rozhraní. [12]

Speciálním typem funkcí jsou takzvané callback funkce. Vyznačují se tím, že používají callback namísto datového typu. Tento callback má podobné vlastnosti jako datový typ, lze ho tedy použít stejným způsobem, ale rozdíl je v tom, že předává referenci na funkci ve formě parametru. [12]

5.1.5 Generování kódu z IDL souboru

Knihovna eRPC využívá pro generování hlavičkových souborů svůj vlastní generátor erpcgen. [14] Tento generátor generuje zvlášť soubory pro stranu klienta a pro stranu serveru, díky tomu snižuje velikost paměťové stopy.

Po vytvoření vstupního souboru je samotné použití generátoru závislé na operačním systému zařízení, na kterém je nástroj spouštěn. V zásadě je ale mimo vyžadovaný software nutné umístit příslušný IDL soubor do zdrojového adresáře generátoru (nebo následně specifikovat jeho lokaci) a spustit generátor pomocí příkazu z příkazového řádku nebo alternativy. [12]

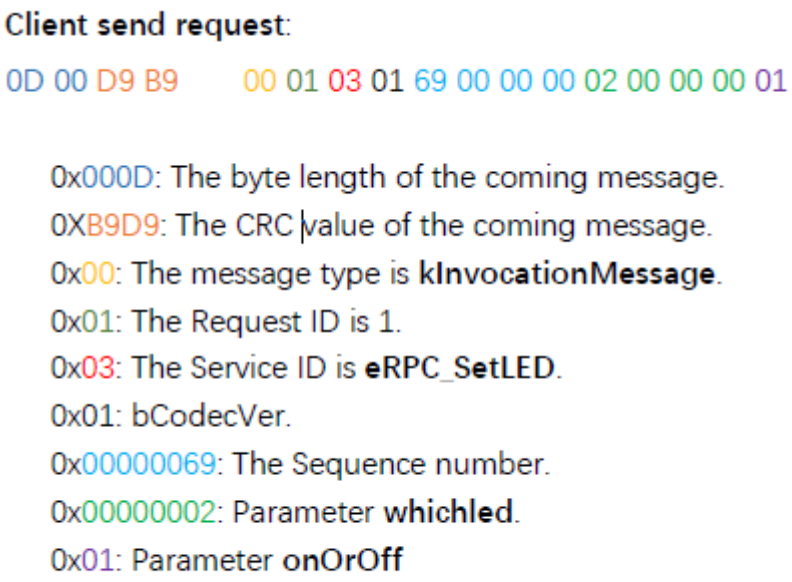
5.2 eRPC protocol

Knihovna eRPC umožňuje použití mnoha druhů kódování zpráv, v základu ale k serializaci zpráv používá svůj vlastní protokol. Tak jako u Protocol Buffers používá eRPC určitou formu byte streamu pomocí kterého jsou data přenášena mezi stranami distribuovaných systémů. [15]

5.2.1 Obsah zprávy

Samotné volání procedury ze strany klienta není v eRPC řešeno pouze jednou celistvou zprávou ale dvojicí podzpráv. První zpráva, označována jako hlavičková, má fixně danou velikost čtyřech bajtů a obsahuje dvě informace, při čemž jsou pro každou vyhrazeny dva bajty. Počáteční dva bajty obsahují informaci o velikosti druhé zprávy, tato informace pak slouží stubu příjemce pro účely deserializace. Druhé dva bajty obsahují CRC hodnotu, která slouží pro porovnání toho, jestli nebyla zpráva při přenosu poškozena.

Obsahem druhé zprávy jsou už všechna potřebná data pro provedení vzdálené procedury. Tato data obsahují mimo samotné parametry pro zpracování také: typ zprávy, request ID, service ID, parametr bCodecVer automaticky nastavenou na fixní hodnotu 1. Mimo parametry pro zpracování tedy zpráva obsahuje celkem 12 bajtů dat sloužících pro správné zpracování požadavku. [15]



Obrázek 5. Příklad obsahu byte streamu [15]

5.3 Infrastruktura

Zdrojový kód infrastruktury knihovny eRPC je strukturován do pěti hlavních složek, jsou jimi složky: config, infra, port, setup a transports. Každá složka obsahuje svoje specifické soubory, které jsou organizovány tak aby co nejvíce usnadnily vývoj softwarů používajícího knihovnu eRPC. [12]

5.3.1 Config

Složka config obsahuje soubor erpc_config.h, který je přímo určen k uživatelským úpravám nastavení funkčnosti knihovny. Většiny úprav lze dosáhnout pomocí jednoduchých editací, například zakomentováním nebo odkomentováním části kódu. Dále slouží jako zdroj částí kódu, jež programátor může snadno zkopírovat a vložit do svého kódu. [12]

5.3.2 Infra

Adresář infra obsahuje C++ kód, který je určený pro sestavování aplikace klienta a serveru. Poskytuje základní funkcionalitu pro serializaci a deserializaci dat, správu bufferů a implementaci kodeků. [12]

5.3.3 Port

Tato složka obsahuje portovací vrstvu eRPC, která umožňnuje adaptaci na různá prostředí. Obsahuje specifické implementace použitelné pro integraci dalšího softwarového a hardwarového systému. [12]

5.3.4 Setup

Adresář setup obsahuje sadu rozhraní v jazyce C, které obalují kód napsaný v jazyce C++. Rozhraní poskytují inicializační rutiny pro stranu klienta a stranu serveru. Tyto soubory v podstatě slouží k tomu, aby výrazně usnadnili implementaci kódu založeného na jazyce C. [12]

5.3.5 Transports

V této složce jsou deklarovány třídy určené pro komunikaci, které umožňuji propojit aplikace využívající eRPC knihovnu s ovladači pro jiné transportní média. Některé jsou určeny pouze pro hostující PC a jiné zase výhradně pro embbeded systémy nebo multi-core systémy. Dále implementuje také metody komunikace ale ve všech případech to musí být minimálně funkce pro odeslání a přijetí zprávy. [12]

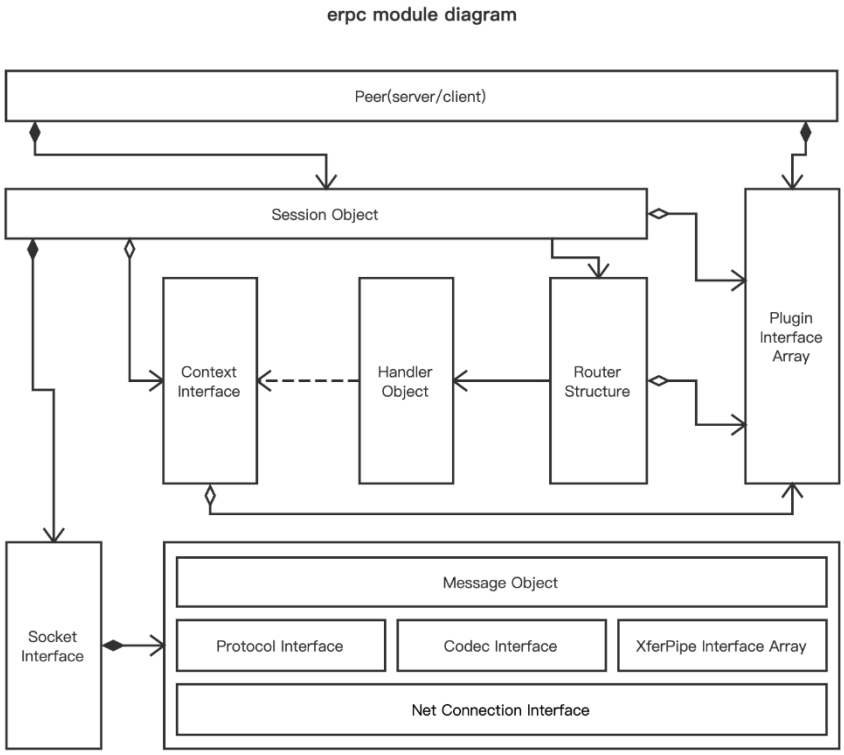
5.4 Transportní vrstva eRPC

Knihovna poskytuje jednotné rozhraní (API) pro jak serverovou, tak i klientskou stranu. To umožňuje programátorům psát kód, který je snadno přenositelný mezi serverovou a klientskou částí aplikace, aniž by museli psát oddělený kód pro každou stranu. [16]

Transportní vrstva se stará o poskytnutí efektivního flexibilního mechanismu pro komunikaci mezi klienty a serverem. Nabízí velkou kompatibilitu se širokým výběrem běžně používaných síťových protokolů. [16]

Podporovanými přenosovými médii jsou mimo jiné například: TCP, UDP, Unix sockety, KCP, QUIC, nebo WebSockety. Tato flexibilita umožňuje vývojářům zvolit ideální prostředek v závislosti na charakteristice konkrétního použití. Kromě možností výběru přenosových medií podporuje knihovna eRPC také volbu mezi několika nástroji pro kódování zpráv včetně JSON, Protobuf, Thrift a XML. [16]

Celkově lze říct, že transportní vrstva knihovny eRPC poskytuje robustní řešení pro komunikaci v distribuovaných systémech bez ohledu na charakteristiku. [16]



Obrázek 6. Diagram modulů eRPC [16]

6 KNIHOVNA NANOPB

Nanopb je knihovna, která zprostředkovává serializaci a deserializaci zpráv do formátu byte streamů, podobně jako to dělá Protocol Buffers ale s minimálními nároky na operační paměť a s malou velikostí kódu. Knihovna je primárně uzpůsobena pro použití v aplikacích embedded systémů, zejména pak pro běh na 32-bitových procesorech. [17]

6.1 IDL knihovny Nanopb

Knihovna Nanopb využívá IDL pro deklaraci zpráv do .proto souboru, který následně slouží jako vstup pro Nanopb generátor. Jakožto knihovna, která implementuje určitou formu Protocol Buffers, používá i stejné syntaxe. Generátor Nanopb zpracovává soubory využívající syntaxe proto2 a proto3. Mimo to ale podporuje svoje vlastní rozšíření nanopb.proto, které umožňuje určitou formu přizpůsobení atributů. [17]

6.1.1 Proto3

Proto3 je IDL jazyk zaměřený na deklaraci zpráv pro Protocol Buffers a je následníkem jazyku proto2. Oproti ostatním IDL se vyznačuje tím, že ve struktuře zprávy označuje jednotlivé atributy čísly podle pořadí v jakém se v byte v streamu nachází. Podle těchto klíčových atributů následně stub pro deserializaci rozřazuje správné hodnoty k příslušným atributům zprávy. Každé číslo v jednotlivé zprávě musí být unikátní a musí se pohybovat v rozsahu 1 až 536 870 911. Jedinou výjimkou jsou čísla v rozsahu 19 000 až 19 999, která jsou rezervovaná pro Protocol Buffers implementaci. [4][18]

Kromě standardních datových typů umožňuje proto3 také deklaraci komplexnějších datových struktur. Přesto, že například přímo nepodporuje deklarace listů nebo polí, tak umožňuje alternativní způsob řešení skrze specifikaci „repeated“, která umožní do jednoho atributu uložit více hodnot stejného datového typu. [18]

Mimo jiné podporuje proto3 také deklarace vnořených zpráv, ve kterých umožňuje přidat do deklaraci obsahu jedné zprávy, deklarace zprávy druhé, jakožto atributu. [18]

```
message SearchResponse {
  message Result {
    string url = 1;
    string title = 2;
    repeated string snippets = 3;
  }
  repeated Result results = 1;
}
```

Obrázek 7. Deklarace vnořené zprávy [18]

6.1.2 Nanopb.proto

Rozšíření nanopb.proto slouží jako nadstavba pro syntax proto2 a proto3. Jeho cílem je adaptovat různé části deklarace zprávy na vhodnější použití v omezenějším prostředí embedded systémů. Jinými slovy, při správném použití snižuje nároky aplikace na výkon systému. Toho se snaží docílit zaměřováním se na oblasti, ve kterých je výkonem plýtváno a co nejvíce omezit tyto zbytečné nároky. [18]

Mimo jiné také rozšiřuje použitelnost komplexnějších datových struktur o dodatečná pole, které mohou sloužit jako atributy zpráv. K těmto zprávám umožňuje deklarace omezení velikosti pro řetězce a pole, díky čemuž dosahuje efektivnějšího využití zdrojů. [18]

Kromě nastavení parametrů u atributů zpráv nanopb.proto umožňuje obecně snadnější zacházení s pamětí, tedy s zajišťuje úspornější využití a správu paměti při serializování a deserializování zpráv. [18]

6.1.3 Generátor Nanopb

Generátor Nanopb využívá jako vstupní soubor .proto s deklaracemi zpráv a generuje z něj výstup o dvou souborech. Jedním je soubor s příponou .pb.h a druhý soubor s příponou .pb.c. Tyto soubory jsou určeny pro vložení jak ke zdrojům na straně klienta, tak na straně serveru a obsahují zpracované deklarace, díky kterým jsou obě strany schopné serializovat a deserializovat příchozí a odchozí zprávy. [18]

Samotné použití generátoru pak zahrnuje pouze umístění vstupního souboru do příslušného adresáře a spuštění generátoru skrze příkaz se specifikací názvu vstupního souboru. [18]

6.2 Serializace a deserializace zpráv

Nanopb nabízí řadu funkcí pro serializaci a deserializaci zprávy, nejprve je však nutné deklarovat stream, ve kterém bude funkce operovat. Následná operace pak závisí na struktuře obsahu zprávy. Serializovat jednoduchou zprávu z pravidla vyžaduje mnohem méně kódu než serializace komplexní zprávy. Například zprávu, která obsahuje jenom jeden nebo více jednoduchých atributů, lze serializovat jako celek. Na druhou stranu, pokud je obsahem zprávy komplexní struktura, která obsahuje zanoření zpráv, bude potřeba nejprve serializovat podzprávy z důvodu výpočtu jejich velikosti a následné zbývající atributy. [18]

Knihovna Nanopb naštěstí zahrnuje velké množství funkcí pro serializaci zpráv, takže lze přistupovat k různým zprávám na základě charakteristiky jejich obsahu. Mimo jiné také knihovna umožňuje nastavení funkce pomocí flagů a díky nim lze ovlivnit jakým způsobem bude ve zprávě indikována velikost zprávy. [18]

6.3 Infrastruktura

Pro použití Nanopb v programu, je nutné mimo soubory s příponami .pb.h a .pb.c zahrnout také několik dalších souborů. Pro všechny instance v distribuovaném systém je nezbytné, aby obsahovaly soubory pb.h, pb_common.h a pb_common.c, a to bez ohledu na to, jakou komunikační úlohu zastávají. Tyto soubory obsahují běžné definice a metody obsluhující iterátor, který je nezbytný ve všech případech užití Nanopb. [18][19]

Jinak tomu je u souborů specificky určených pro serializaci a deserializaci, jsou jimi pb_decode.h/c a pb_encode.h/c. Může se totiž stát, že instance distribuovaného systému bude pouze zasílat zprávy, ale už nebude očekávat odpovědi. V takovém případě odpadá nutnost pro soubory pb_encode.h/c. [18]

II. PRAKTICKÁ ČÁST

7 VYHODNOCENÍ FAKTORŮ PRO POROVNÁNÍ KNIHOVEN

Vzhledem k charakteru obou knihoven je nutné najít relevantní faktory, které budou u obou knihoven porovnatelné. To není zas tak jednoduché, protože obsahy obou knihoven se poměrně liší. Knihovna eRPC je komplexní knihovna, která je přímo určená pro implementaci vzdáleného volání procedur. Pokud ale přichází v úvahu knihovna Nanopb, tak ta, co se týče obsahu, nabízí spíše obecnější použití, a to pouze serializaci a deserializaci zpráv. Ačkoliv se k tomuto účelu také hodí a dá se tedy nazývat knihovnou použitelnou pro vzdálené volání procedur, je nutné za normálních okolností přistupovat ke knihovnám trochu rozdílným způsobem.

Práce se nicméně zaměřuje hlavně na zjištění možností knihovny Nanopb a snaží se ověřit domněnku, že by bylo možné použít tuto knihovnu ve stejném prostředí, pro které je navržená knihovna eRPC.

V základu se totiž obě knihovny snaží řešit tu samou problematiku, a to nabídnout flexibilní způsob komunikace v prostředí, které disponuje pouze omezenými prostředky. Snaží se tedy sloužit jako prostředek s nízkými nároky vhodný pro použití v embedded systémech.

7.1 Požadavky na testovací prostředí

Pro relevantní porovnání obou knihoven je nutné zvolit prostředí, které nasimuluje typické použití knihovny eRPC, tedy použití ukázkové testovací aplikace. Následně bude aplikace implementována s použitím Nanopb namísto knihovny eRPC. Cílem je v první řadě zjistit, do jaké míry lze Nenopb ve stejném případě použít a dále porovnat, jak si v porovnání s druhou knihovnou vedla. V poslední řadě je nutné, aby byla testovací aplikace měřitelná v ohledech, které jsou klíčové pro zvolení softwaru v embedded systémech.

7.2 Parametry k porovnání

Vzhledem k charakteristice obou knihoven je nutné vybrat ty parametry, které budou ovlivněny implementací knihoven, nemá tedy význam porovnávat přenosová média knihoven, protože knihovna nanopb jako taková žádné nástroje pro přenos neobsahuje. Takové měření by tedy záviselo čistě na vlastnostech knihovny použité pro přenos. Zvolenými parametry jsou: rychlost, velikost obsahu přenášené zprávy a footprint knihoven v aplikaci. Mimo faktické získávání bude také porovnána použitelnost a snadnost implementace.

7.2.1 Rychlost

Rychlost, přesněji tedy doba, za kterou je vzdálené volání procedury provedeno, je zvolena proto, že je to standardně parametr, na který je v embedded systémech kladený důraz díky jejich omezené výpočetní kapacitě. V dokumentaci knihovny Nanopb je uvedeno, že by měla díky sníženým nárokům na operační paměť naopak obětovat rychlost. [17] Tento test tím pádem ukáže, zda-li je zmíněný nárůst ve zpracování zpráv markantní a mohl by být rozhodujícím faktorem pro volbu mezi knihovnami.

7.2.2 Velikost přenášené zprávy

Samotná velikost zprávy může ovlivňovat hned několik faktorů, nižší velikost přenášené zprávy by mohla znamenat zvýšení rychlosti přenosu. Díky tomu by knihovna Nanopb nemusela být výrazně pomalejší v případě, že by byla její velikost zpráv nižší. Nicméně samotné volání procedur obvykle zahrnuje relativně malé obsahy zpráv, takže by se efekt nemusel projevit. To ale na relevantnosti nic nemění, protože nižší velikost zprávy znamená nižší zatížení sítě, což je v prostředí s nízkokapacitní sítí výhoda, nehledě na menší náchylnosti k chybovosti.

7.2.3 Footprint

Obě knihovny disponují malou velikostí kódu [12][17]. Vzhledem k tomu, že toto tvrzení je subjektivní, tak měření poslouží k porovnání toho, jak si knihovny vedou oproti konkurenci. To se týče velikosti souborů, které jsou knihovnami v programu vyžadovány, jak už pro generované, tak pro běžné.

7.2.4 Složitost implementace

V poslední řadě je důležité porovnat složitost implementace, tedy zjistit, jak moc se implementace skrze Nanopb liší od standardních RPC knihoven jako je eRPC. I přesto, že je zjevné, že rozdíl v knihovnách bude veliký, tak je nutné zjistit, jestli je vůbec možnost volby knihovny Nanopb jako takové realistická. Bez ohledu na možné benefity v oblasti výkonu je nutné, zaměřit se na možnosti použití ve praxi.

8 IMPLEMENTACE VZOROVÉ APLIKACE

Vzorová aplikace by měla obsahovat běžný způsob přenosu zprávy, který bude co nejvíce připomínat použití v praxi. Pro tento účel byla zvolena aplikace pro násobení dvou matic. Tento způsob implementace je v dokumentacích knihovny eRPC často používán jako příklad, který reprezentuje základní použití knihovny. Tato aplikace je vhodná pro měření vlastností díky několika faktorům. Zaprvé obsahuje na RPC proceduru velké množství přenášených dat, díky kterým lze snadněji rozlišit rozdíly v rychlosti a délce serializované zprávy. A za druhé na místo přenášení jednoduchého datového typu obsahuje komplexnější datovou strukturu. Díky tomu lze snadněji rozpoznat rozdíly v použitelnosti při tvorbě IDL souborů a při jejich serializaci.

Aplikace matrix multiply deklaruje dvě matice 5x5, které jsou na straně klienta naplněny hodnotami a následně poslány na stranu serveru ve formě serializované zprávy. Zde je pak zpráva deserializována a její obsah je použit pro parametry funkce, která matice vynásobí. Výsledná matice je poslána stejným způsobem zpět na stranu klienta.

8.1 Použitá platforma

Jako platforma byla zvolena vývojová deska LPCXpresso55s69. Důvodem pro tuhle volbu byl hlavně její dvoujádrový procesor s jádry ARM Cortex-M33 s frekvencí až 150 MHz. Dvoujádrový procesor byl pro výběr nutností, protože aplikace testuje použití RPC pro komunikaci mezi jádry. Jedno jádro bude tedy zastávat funkci serveru a druhé funkci klienta. Mimo to disponuje dostatečnou pamětí pro účely testování a vestavěným on-board debuggerem pro snazší ladění. [20][21]

8.2 Implementace pomocí knihovny eRPC

Aplikace matrix multiply je součástí programového balíčku příkladů pro LPCxpresso55s69. Bylo tedy potřeba jenom minimálních úprav kódu, protože je aplikace předem nakonfigurovaná pro běh na dané vývojové desce.

8.2.1 IDL generátoru knihovny eRPC

Definice procedury matrix multiply je v případě použití knihovny eRPC velice jednoduchá a celý obsah lze intuitivně napsat na několik řádků. Díky tomu, že používané IDL podporuje deklaraci vícerozměrných polí, tak není třeba matici nahrazovat jinou formou struktury, jak

by tomu bylo v ostatních případech. Samotný IDL soubor tedy obsahuje pouze deklaraci matice s pevnou velikostí a vstupních a výstupních parametrů procedury matrix multiply.

```
program erpc_matrix_multiply

/*! This const defines the matrix size. The value has to be the same as the
    Matrix array dimension. Do not forget to re-generate the erpc code once
the
    matrix size is changed in the erpc file */
const int32 matrix_size = 5;

/*! This is the matrix array type. The dimension has to be the same as the
    matrix size const. Do not forget to re-generate the erpc code once the
    matrix size is changed in the erpc file */
type Matrix = int32[matrix_size][matrix_size];


interface MatrixMultiplyService {
    erpcMatrixMultiply(in Matrix matrix1, in Matrix matrix2, out Matrix
result_matrix) -> void
}
```

Obrázek 8. Definice IDL eRPC [10]

8.2.2 Strana klienta

Strana klienta je obsažena v programu pro jádro 0. Funguje tím pádem jako master v hierarchii Primary/Secondary. Tato strana je spouštěna jako první a stranu serveru bootuje až ve svojí inicializační části. Následně pak před pokračováním čeká na odpověď ze strany serveru. Po inicializaci všech využívaných prostředků a navázání komunikace naplní matice hodnotami a pošle požadavek pro jejich vynásobení na stranu serveru skrze komunikační komponent RPMsg-Lite. Dále už jen čeká na hodnoty zaslané ze serveru. Celá komunikace je na straně klienta skrytá a procedura se chová jako běžná funkce.

```
for (;;)
{
    (void)PRINTF("\r\neRPC request is sent to the server\r\n");

    erpcMatrixMultiply(matrix1, matrix2, result_matrix);

    /* Check if some error occurred in eRPC */
    if (g_erpc_error_occurred)
    {
        /* Exit program loop */
        break;
    }

    (void)PRINTF("\r\nResult matrix");
    (void)PRINTF("\r\n=============\r\n");
    (void)print_matrix(result_matrix);
```

Obrázek 9. Implementace eRPC ze strany klienta [10]

8.2.3 Strana serveru

Po svojí vlastní inicializaci a signalizaci straně klienta o svojí připravenosti server ve smyčce čeká na příchozí zprávy. Pokud zpráva přijde, pozná to pomocí polling mechanismu a zprávu zpracuje na základě přidaných procedur v inicializační části. Pokud klient ukončí komunikaci, deinicializuje komunikaci. Veškeré serializace a deserializace se odehrávají mimo hlavní programovou část.

```
Primary core started

Matrix #1
=========
  33   12    0    2    6
  11   43    8   30   19
  17   22   30   49   14
   1   40   35   44   12
  13   10   26   29   25

Matrix #2
=========
  43   29    8    6   19
   1    5   43   16   32
  47   18   23   42   19
  27   24   20   10   34
   1   36   31   47   41

eRPC request is sent to the server

Message length: 208

Result matrix
=============
1491 1281 1006  692 1325
1721 2082 3310 2283 3536
3500 2823 3186 2862 3837
2928 2347 3785 3120 3952
2599 2491 2487 2795 3072
```

Obrázek 10. Výstup matrix multiply

8.3 Implementace pomocí Nanopb

Implementace pomocí knihovny Nanopb byla podstatně složitější, vzhledem k tomu, že knihovna nabízí pouze serializaci a deserializaci zpráv, tak sama o sobě neumožňuje žádnou transparentnost. Díky tomu je potřeba implementovat všechny kroky pro provedení vzdáleného volání procedury manuálně. V praxi by to znamenalo rozdělit program do více souborů pro větší přehlednost a použitelnost. Nicméně pro účely testovací aplikace bylo rozhodnuto, že všechny jednotlivé kroky budou provedeny v hlavních souborech. Díky tomu bude snadnější porovnat rozsah implementace s verzí aplikace napsané pomocí eRPC. Samotná implementace pak neobsahuje všechny prvky, které jsou využity v původní příkladové aplikaci, a to z toho důvodu, aby bylo možné zjistit možnosti co nejúspornějšího použití této knihovny. Nutností však je zachovat funkčnost v plném rozsahu.

8.3.1 IDL generátoru knihovny Nanopb

Zprávy definovaná pomocí IDL Nanopb lze zapsat několika různými způsoby, v případě napodobení obsahu zpráv z příkladu matrix multiply bylo ale zapotřebí udělat několik kompromisů. Největší dopad měla skutečnost, že používané IDL nepodporuje použití vícedimenzionálních polí, ani polí, samostatných. Alternativní definice pole je sice díky rozšíření Nanopb.proto značně zjednodušená, ale to se vztahuje pouze k jednodimenzionálním polím. Proto bylo nutné matice nahradit jinou strukturou, kterou bude možno iterovat co možná nejpodobněji.

K tomu se nabízí využití předpony repeated, která shlukuje více instancí jednoho datového typu. Jako datový typ lze však použít i komplexnější datovou strukturu, včetně datového typu s předponou repeated. Díky tomu lze dvoudimenzionální pole nasimulovat skrze vnořené použití repeated atributu, což se zpočátku jevilo jako ideální způsob napodobení. Bylo od něj však nakonec ustoupeno. Problematika této varianty definice spočívá v tom, že Nanopb vyžaduje serializaci podzpráv zvlášť. To zvyšuje komplikaci serializace s každou další vnořenou zprávou.

Místo toho byla zvolena možnost použití pouze jednoho repeated atributu o velikosti 25 prvků. Tato změna vyžadovala upravení funkcí pro iterace maticemi, nicméně stále zachovala počet přenášených prvků ve zprávě, takže by neměla mít na výsledky měření žádný vliv.

```
import "nanopb.proto";

message Matrix {
  repeated int32 row = 1 [(nanopb).max_count = 25, (nanopb).fixed_count = true, packed = true];
}

service MatrixMultiplyService {
  rpc MatrixMultiply(MatrixRequest) returns (MatrixResponse);
}

message MatrixRequest {
  Matrix matrix1 = 1;
  Matrix matrix2 = 2;
  int32 request_id = 3;
}

message MatrixResponse {
  Matrix result_matrix = 1;
}
```

Obrázek 11. Definice IDL Nanopb

Mimo samotné definice zpráv se v IDL souboru nachází také definice MatrixMultiplyService, jak lze vidět na obrázku 11. Tímto způsobem by byla ve standardních RPC knihovnách definována volatelná procedura, nicméně kvůli omezením knihovny nemá Nanopb pro takovou definici využití. Tím pádem v souboru slouží pouze pro účely dokumentace.

Informace o tom, jaká operace má být s daty na straně serveru provedena, byla zahrnuta do samotné zprávy MatrixRequest spolu s maticemi. Takový způsob organizace zprávy není běžný, ale byl zvolen pro testování možností úspornosti.

8.3.2 Strana klienta

Hierarchie mezi jádry je řešena stejným způsobem, jako je tomu v příkladě implementovaným pomocí eRPC. Po inicializaci a navázaní komunikace s protějším jádrem vytvoří RPMsg-Lite instance a endpoint. Pak program inicializuje samotnou zprávu MessageRequest a naplní ji daty pomocí funkce s upravenou iterací pro jednodimenzionální pole. Následně pak nastává zdlouhavý proces serializace zprávy.

```
// Encode matrix1
if (!pb_encode_tag(&stream, PB_WT_STRING, MatrixRequest_matrix1_tag)) {
    (void) PRINTF("Encoding failed for matrix1 tag\n");
    return 1;
}
if (!pb_encode_submessage(&stream, Matrix_fields, &request.matrix1)) {
    (void) PRINTF("Encoding failed for matrix1\n");
    return 1;
}

// Encode matrix2
if (!pb_encode_tag(&stream, PB_WT_STRING, MatrixRequest_matrix2_tag)) {
    (void) PRINTF("Encoding failed for matrix2 tag\n");
    return 1;
}
if (!pb_encode_submessage(&stream, Matrix_fields, &request.matrix2)) {
    (void) PRINTF("Encoding failed for matrix2\n");
    return 1;
}

// Encode request_id
if (!pb_encode_tag(&stream, PB_WT_VARINT, MatrixRequest_request_id_tag)) {
    (void) PRINTF("Encoding failed for request_id tag\n");
    return 1;
}
if (!pb_encode_varint(&stream, request.request_id)) {
    (void) PRINTF("Encoding failed for request_id\n");
    return 1;
}
```

Obrázek 12. Serializace zprávy pomocí Nanopb

Jak je z obrázku 12. patrné, každá podzpráva je serializována zvlášť. Tento postup platí i pro jednoduchý datový typ, kterým je request_id. Ten by nemusel být serializovaný zvlášť, nicméně vzhledem k tomu, že po serializaci matic zůstává jako jediný zbývající atribut, tak je to preferované řešení. Pokud by byl každý parametr jednoduchý datový typ, tak by pro serializaci stačil pouze jeden příkaz.

Tyto serializované atributy se průběžně nahrávají do streamu, který je následně poslán pomocí komunikačního média RPMsg-Lite na protější stranu.

Dále pak klientská strana čeká ve smyčce na odpověď od strany serveru. Příjem zprávy je řešen skrze zavolání callback funkce, která zkopíruje obsah zprávy do bufferu a přeruší čekací smyčku. Obsah bufferu je následně deserializován a výsledek vypsán na konzoli.

```
static int32_t my_ept_read_cb(void *payload, uint32_t payload_len, uint32_t src,
        void *priv) {
    int32_t *has_received = priv;
    response_length = payload_len;

    if (payload_len <= sizeof(msg_buffer)) {
        (void) memcpy((void*) &msg_buffer, payload, payload_len);
        *has_received = 1;
    }
    (void) PRINTF("Primary core received a msg\r\n");
    //(void)PRINTF("Message: Size=%x, DATA = %s\n", payload_len, msg_buffer);
    (void)PRINTF("\r\nMessage length: %d\n\r", payload_len);

    return RL_RELEASE;
}
```

Obrázek 13. Callback funkce pro příjem zpráv

```
MatrixResponse response = MatrixResponse_init_zero;
pb_istream_t stream = pb_istream_from_buffer(msg_buffer,
                                    response_length);

pb_status = pb_decode(&stream, MatrixResponse_fields, &response);
if (!pb_status) {
    (void)PRINTF("\r\nMessage decoding failed\r\n");
    return 1;
}
```

Obrázek 14. Deserializace zprávy z bufferu

8.3.3 Strana serveru

Strana serveru po inicializaci čeká na zprávu od klienta stejným způsobem jako protější strana pomoci callback funkce vyobrazené na obrázku 13. Pokud zprávu dostane definuje její prázdnou formu a následně do ní skrze deserializaci nahraje hodnoty. Na rozdíl od serializace je jednoduchá i přes komplexní strukturu zprávy.

Pokud by bylo za potřebí rozpoznat mezi větším množstvím různých struktur zpráv před deserializací, bylo by nutné deserializovat první atribut request_id, podle kterého by se následně určila charakteristika volané procedury. To je možné, protože má strana serveru informace o pozici atributu v byte streamu. Pro demonstrativní účely není však nutné tento systém implementovat.

V případě pouze jedné možné struktury zprávy je request_id deserializován spolu se zbytkem zprávy, pro výběr funkce byl pak použit jednoduchý switch příkaz znázorněn na obrázku 14. Ani tato implementace nebyla vyloženě nutností díky pouze jedné možnosti, ale nakonec bylo rozhodnuto tento příkaz ponechat pro lepší reprezentaci funkčnosti RPC.

```
//switch case handles the request service
switch (request.request_id) {
case MATRIX_MULTIPLY_SERVICE:

    MatrixMultiply(request.matrix1.row,request.matrix2.row,response.result_matrix.row);
    break;
default:
    // Handle unknown request ID
    char *str;
    str = "unknown request";
    (void) rpmsg_lite_send(my_rpmsg, my_ept, remote_addr, str,
           strlen(str) + 1, RL_DONT_BLOCK);
    return 1;
    break;
}
```

Obrázek 15. Řešení volby funkce na základě request_id

Před výběrem funkce je zadefinována prázdná zpráva, která následně poslouží jako jeden ze vstupních parametrů funkce. Do této zprávy je výsledná zpracovaná matice uložena.

Po zpracování maticí je výsledek opět serializován a odeslán zpět na stranu klienta. Po vykonání procedury strana serveru opět čeká ve smyčce na další příchozí zprávy.

9 SBĚR DAT PRO POROVNÁNÍ KNIHOVEN

9.1 Měření rychlosti

Vzhledem k tomu, že ani jedna z aplikací nevyužívá nástroje, které by blokovaly využití časovače, tak se pro měření času nabízí použití ovladače pro SysTick. SysTick funguje jako 24-bitový čítač, který generuje pravidelné přerušení. Obvykle na základě hodinového signálu systému. [22] Pokud aplikace zaznamená jeho hodnotu na začátku a na konci měřeného časového úseku, lze z něj uplynulý čas odečíst.

9.1.1 Měřený časový úsek

Pro relevantní porovnání je třeba určit stejný časový úsek pro obě implementace. Hlavním cílem je porovnat dobu provedení vzdáleného volání procedury. V případě implementace pomocí eRPC bude mezi začátkem a koncem pouze jeden řádek, a to samotné volání procedury.

Všechny aplikace byly měřeny v debug konfiguraci.

```
waitStart = SYSTICK_micros();

erpcMatrixMultiply(matrix1, matrix2, result_matrix);

currentTime = SYSTICK_micros();
(void)PRINTF("\r\nTime: %d\n\r",currentTime - waitStart);
```

Obrázek 16. Měření času pomocí SysTick

V případě implementace pomocí Nanopb byl zvolen úsek programu takovým způsobem, aby zahrnul veškeré operace, které jsou v eRPC prováděny transparentně. Začíná tedy ještě před serializací hodnot a končí v momentě deserializace odpovědi přijaté ze strany serveru.

9.1.2 Naměřené hodnoty

Tabulka 3. Naměřená data – porovnání rychlosti

eRPC	1,674 ms
Nanopb	1,501 ms

9.2 Měření velikosti zprávy

K změření velikosti zprávy, tedy počtu přenášených bajtů není potřeba žádný externí nástroj. Každá z aplikací obsahuje část v programu, která s hodnotou velikosti zprávy přímo pracuje, stačí ji tedy zobrazit na terminálu.

V případě implementace Nanopb operuje s touto hodnotou callback funkce. Je proto snadné ji získat. V případě použití eRPC už to tak snadné není, díky její transparentnosti. Dostat se k velikosti zprávy vyžaduje získat hodnotu přímo v transportní vrstvě, tedy přesně ze souboru, který obsahuje definici funkce pro odesílání zprávy. Hodnota byla získána ze souboru erpc_rpmsg_lite_transport.ccp. Data byla získávána z obou typů zpráv, jak z požadavku, tak z její odpovědi.

9.2.1 Naměřené hodnoty

Tabulka 4. Naměřená data – velikost zprávy

Implementace	Velikost požadavku	Velikost odpovědi
eRPC	208 B	108 B
Nanopb	60 B	54 B

9.3 Měření footprintu

Každá z knihoven vyžaduje určité soubory pro svůj běh, ty si při nahrání a při běhu nárokují místo v paměti RAM. Tyto soubory mohou a nemusí být exkluzivní pro stranu klienta nebo serveru a mohou být využity jiným způsobem. Proto je nutností získat data z obou části aplikace. Toho bylo docíleno pomocí analýzy mapovacích souborů, které obsahují data o footprintu jednotlivých částí aplikace. Samotné mapovací soubory jsou však uživatelsky nečitelné, proto bylo potřeba zvolit nástroj, který dokáže mapovací soubory konvertovat do čitelné podoby. K tomuto účelu byl vybrán nástroj amap, který umožňuje rozřadit data do přehledných organizovaných záložek a vyobrazí velikosti v bajtech.

9.3.1 Footprint eRPC

File	Size no .bss	Size	Num of records
./source/erpc_error_handler.o	25549	25550	29
./erpc/transports/erpc_rpmsg_lite_transport.o	25542	25547	48
./erpc/setup/erpc_client_setup.o	19121	19125	25
./erpc/setup/erpc_setup_mbf_rpmsg.o	20392	20408	34
./erpc/setup/erpc_setup_rpmsg_lite_master.o	19382	19742	21
./erpc/service/erpc_matrix_multiply_client.o	11598	11598	15
./erpc/infra/erpc_basic_codec.o	27996	27996	80
./erpc/infra/erpc_client_manager.o	15901	15902	25
./erpc/infra/erpc_message_buffer.o	11650	11650	23
./erpc/port/erpc_port_stdlib.o	6596	6596	16
./erpc/infra/erpc_crc16.o	3488	3488	12

Obrázek 17. Footprint eRPC strany klienta

File	Size no .bss	Size	Num of records
./source/erpc_error_handler.o	25189	25190	29
./erpc/transports/erpc_rpmsg_lite_transport.o	25470	25475	48
./erpc/setup/erpc_server_setup.o	21157	21157	35
./erpc/setup/erpc_setup_mbf_rpmsg.o	20341	20357	34
./erpc/setup/erpc_setup_rpmsg_lite_remote.o	13195	13555	20
./erpc/service/erpc_matrix_multiply_server.o	15249	15249	27
./erpc/infra/erpc_basic_codec.o	27958	27958	80
./erpc/infra/erpc_message_buffer.o	11637	11637	23
./erpc/infra/erpc_simple_server.o	15258	15258	23
./erpc/port/erpc_port_stdlib.o	6545	6545	16
./erpc/infra/erpc_crc16.o	3473	3473	12
./erpc/infra/erpc_server.o	13425	13426	20

Obrázek 18. Footprint eRPC strany serveru

Tabulka 5. Naměřená data – Footprint main souborů erpc

main soubor klienta	38468 B
main soubor serveru	29329 B

9.3.2 Footprint Nanopb

File	Size no .bss	Size	Num of records
./source/pb_decode.o	22662	22662	60
./source/pb_encode.o	15838	15838	50
./source/pb_common.o	7356	7356	21
./source/matrix.pb.o	2588	2588	15

Obrázek 19. Footprint Nanopb ze strany klienta

File	Size no .bss	Size	Num of records
./source/pb_decode.o	22654	22654	60
./source/pb_encode.o	15850	15850	50
./source/pb_common.o	7334	7334	21
./source/matrix.pb.o	2596	2596	15

Obrázek 20. Footprint Nanopb ze strany serveru

Tabulka 6. Naměřená data – Footprint main souborů Nanopb

main soubor klienta	62037 B
main soubor serveru	56322 B

10 ZHODNOCENÍ NASBÍRANÝCH DAT

10.1 Rychlost

Co se týče doby naměřené od začátku vzdáleného volání procedury do konce, výsledky obou knihoven dopadly podobně, implementace pomocí Nanopb dosáhla lepšího času pouze o necelé dvě desetiny milisekundy, což je vzhledem k rozdílnosti obou implementací zanedbatelná hodnota.

Lze očekávat, že pokud by tato implementace napodobovala příkladovou aplikaci v plném rozsahu, byla by doba provedení procedury delší. Implementace pomocí Nanopb zkoumala co nejúspornější možné použití, nezahrnula tedy některé prvky, jako například dělení zprávy na hlavičkovou a datovou část zprávy. Nicméně se dá konstatovat, že při volbě použití této knihovny by rychlost nebyla faktorem, který by byl důvodem pro výběr mezi knihovnami.

10.2 Velikost zprávy

Rozdíly ve velikostech přenášených zpráv byly značně větší. Na základě naměřených dat lze pozorovat hlavní rozdíly knihoven v přístupu ke skládání byte streamů.

Knihovna eRPC využívá značně neoptimalizovaný přístup, ve kterém přiděluje místo ve streamu na základě velikosti přenášeného datového typu. Tato skutečnost je patrná z naměřené hodnoty. První ze zpráv má velikost 208 bajtů, z toho lze snadno odvodit, že pouze využívá velikost datového typu int32, tedy 4 bajty. Každá matice zabírá 100 bajtů, podle vzorce počet přenášených prvků vynásobených velikostí jejich datového typu. Zbylých 8 bajtů je dedikovaných pro hlavičkovou část zprávy, specifikaci volané procedury a pro další podpůrné atributy, jak je popsáno v kapitole 5.2.1. Stejný vzorec je vidět i u velikosti druhé zprávy, která obsahuje 108 bajtů s tím rozdílem, že obsahuje pouze jednu matici.

Knihovna Nanopb využívá značně úspornějšího systému skládání byte streamů. Z naměřených dat je zřejmé, že přiděluje místo na základě velikosti hodnoty, nikoliv maximální kapacity datového typu. Díky tomu je rozdíl mezi velikostí obou jejich zpráv minimální i přes dvojnásobné množství přenášených dat. To je způsobené tím, že druhá zpráva obsahující výslednou matici obsahuje násobky hodnot, tedy víceciferná čísla.

Tento rozdíl by mohl být klíčovým aspektem pro volbu knihovny Nanopb v prostředí se velmi omezenou síťovou kapacitou.

10.3 Footprint

Porovnání footprintů obou aplikací je spíše orientační, ze shrnutí mapovacích souborů vyplývá očekávatelný výsledek. Tedy to, že úspornější raelizace pomocí knihovny Nanopb má menší footprint. To však samo o sobě nemusí sloužit jako rozhodovací faktor, protože samotná realizace přenosové vrstvy v implementaci Nanopb nebyla do sběru dat zahrnuta. To se také odráží na velikostech footprintů main souborů. Knihovna eRPC má díky své transparentnosti ve zmíněném ohledu mnohem menší stopu než knihovna Nanopb. Ta pro demonstrativní účel zahrnuje většinu svojí funkcionality právě do těchto souborů.

Výsledky však naznačují, že samotná implementace Nanopb výrazně nevybočuje z očekávaných hodnot, díky tomu by se dalo říct, že použití této knihovny nebude mít negativní dopad na celkový footprint, ale spíše naopak.

10.4 Použitelnost

Co se týče použitelnosti eRPC výrazně dominuje se svými možnosti oproti Nanopb. Použití Nanopb jakožto RPC knihovny je velmi uživatelsky nepřívětivé a bez potřebné nadstavby je použitelná jenom v extrémních případech. Knihovna nenabízí vůbec žádnou modularitu a programátor si musí všechny prvky krom serializace a deserializace obstarat jiným způsobem. To je ale díky charakteristice knihovny očekávatelné a akceptovatelné.

Nicméně samotná serializace u Nanopb je značně komplikovaná. Je to daň za způsob, kterým knihovna přistupuje ke tvoření byte streamů, díky nimž nabízí oproti knihovně eRPC benefit v podobě malé velikosti zpráv.

Knihovna Nanopb zaostává i ve složitosti deklarování IDL souborů, ve které nabízí méně možností než konkurence, což vede k nutnosti hledat alternativy k zápisům jinak běžných datových struktur.

10.5 Potenciál využití kombinace Nanopb a eRPC

Knihovna Nanopb disponuje velmi efektivním způsobem serializace a deserializace dat, nicméně samostatně je pro RPC téměř nepoužitelná. Oproti knihovně eRPC však nabízí určité benefity v podobě velmi úsporného řešení přenášených informací. Tato vlastnost je v embedded systémech výhodou. Díky rozmanitosti nároků různých distribuovaných systému by mohla sloužit jako kritický faktor pro volbu za předpokladu, že by byla rozšířena o plnohodnotnou RPC nadstavbu.

Knihovna eRPC nabízí široký výběr nástrojů pro kódování zpráv. Pomocí toho nabízí flexibilní možnosti pro vývoj specializovaných aplikací v konkrétních scénářích. Rozšíření tohoto výběru o nástroj Nanopb by mohlo znamenat benefity pro už tak dost širokou základnu, stojí to tedy za zvážení.

ZÁVĚR

Cílem bakalářské práce bylo zjistit, do jaké míry lze použít knihovnu Nanopb jako RPC nástroj a porovnat ji s knihovnou eRPC, která byla k tomuto účelu určena. Bylo tedy nutné přistupovat k oběma knihovnám podobně i přes jejich rozdíly.

K pochopení celé problematiky sloužila teoretická část, ve které byly rozebrány koncepty RPC s přiblížením k prostředí, ve kterém byly obě knihovny testovány. Poznatky získané z této části byly využity při návrhu prostředí a vyhodnocení toho, které vlastnosti knihoven by měly být pro relevantní výsledky porovnány. V druhé polovině teoretické části byly podrobně rozebrány obě knihovny, specificky jejich možnosti použití. Tato část umožnila autorovi zorientovat se v používání obou knihoven a poznatky z ní byly použity pro realizaci praktické části.

Praktická část měla za úkol porovnat implementace obou knihoven. Bylo tedy nutné najít ideální scénář, který by vyzdvihl rozdíly obou knihoven. Jakožto reference pro návrh aplikace byla zvolena testovací aplikace matrix multiply, která slouží jako demonstrativní příklad při prezentování použitelnosti eRPC. Na základě toho byla vytvořena podobná aplikace realizovaná pomocí knihovny Nanopb s důrazem na co nejúspornější způsob použití pro prezentaci jejich možností. Obě aplikace byly rozšířeny o prvky, které umožňovaly sběr dat pro následné zhodnocení.

Výstupem práce je RPC aplikace implantovaná pomocí Nanopb a data získaná z provozu obou aplikací. Tato data byla vyhodnocena a pomocí nich byla určena odpověď na prvotně kladenou otázku. Ze zhodnocení vyšlo najevo, že je knihovna Nanopb sama o sobě téměř nepoužitelná jakožto RPC nástroj, nicméně se v určitých ohledech osvědčila a předčila knihovnu eRPC. Díky tomu lze říct, že za určitých okolností dokáže nabídnout pro vzdálené volání procedur benefity.

SEZNAM POUŽITÉ LITERATURY

[1] TANENBAUM, Andrew S. a VAN STEEN, Maarten. Distributed systems: principles and paradigms. 2nd ed. Upper Saddle River: Pearson Education, 2007. ISBN 978-0132392273.

[2] WEI, Yi-Hsiu; STOYENKO, Alexander D. a GOLDSZMIDT, German S. The design of a stub generator for heterogeneous RPC systems. Online. Journal of Parallel and Distributed Computing. 1991, roč. 11, č. 3, s. 188-197. ISSN 07437315. Dostupné z: https://doi.org/10.1016/0743-7315(91)90043-9. [cit. 2024-05-07].

[3] The Interface Definition Language. OrbixWeb Programming Guide [online]. [cit. 2024-05-08]. Dostupné z: https://www.cs.cmu.edu/afs/cs/project/classes-bob/link/tool/OrbixWeb2.0/doc/pguide/idl.html

[4] CURRIER, Chris. Protocol Buffers. In: HUMMERT, Christian a Dirk PAWLASZCZYK, ed. Mobile Forensics – The File Format Handbook. Springer International Publishing, 2022, s. 223-260. ISBN 978-3-030-98467-0.

[5] BESCHOKOV, Mukhadin. What Is GRPC? Meaning, Architecture, Advantages. Wallarm [online]. ©2024 [cit. 2024-05-07]. Dostupné z: https://www.wallarm.com/what/the-concept-of-grpcc

[6] What is gRPC? GRPC [online]. ©2024 [cit. 2024-05-08]. Dostupné z: https://grpc.io/docs/what-is-grpc/

[7] SIRHAN, Najem N. Multi-core processors: concepts and implementations. International Journal of Computer Science & Information Technology (IJCSIT). 2018, 10(1), 1-10.

[8] GOYAL, Abhilash. Multi-Core Processors and Systems: State-ofthe-Art and Study of Performance Increase. San Jose, CA 95192, 2014. Dostupné také z: https://api.semanticscholar.org/CorpusID:53516059. San Jose State University.

[9] ABDALLAH, Abderazek Ben. Multicore Systems On-Chip: Practical Software/Hardware Design: Practical Software/Hardware Design. 2. Second edition. Springer, 2013. ISBN 978-94-91216-91-6.

[10] NXP. EmbeddedRPC / erpc. [online]. c2016-2023 [cit. 2024-05-08]. Dostupné z: https://github.com/EmbeddedRPC/erpc

[11] Introducing eRPC. NXP Community [online]. ©2006-2024 [cit. 2024-05-08]. Dostupné z: https://community.nxp.com/t5/MCUXpresso-SDK-Knowledge-Base/Introducing-eRPC/ta-p/1099139

[12] EmbeddedRPC / erpc Wiki. GitHub [online]. ©2024 [cit. 2024-05-08]. Dostupné z: https://github.com/EmbeddedRPC/erpc/wiki

[13] The Java™ Tutorials - Lesson: Annotations. Oracle Java™ Documentation [online]. ©1994-2024 [cit. 2024-05-08]. Dostupné z: https://docs.oracle.com/javase/tutorial/java/annotations/index.html

[14] ERPC Generator (erpcgen). ERPC Infrastructure [online]. ©2016-2020 [cit. 2024-05-08]. Dostupné z: https://embeddedrpc.github.io/erpcgen/index.html

[15] NXP SEMICONDUCTORS. Dual MCU Sync Solution User Guide. Rev 1.0. 2020. Dostupné také z: https://community.nxp.com/t5/S32K-Knowledge-Base/eRPC-based-Dual-MCU-Sync-Solution-for-S32K1xx-Family-MCU/ta-p/1151939

[16] ERPC README. GitHub [online]. ©2024 [cit. 2024-05-08]. Dostupné z: https://github.com/andeya/erpc/blob/master/README.md

[17] AIMONEN, Petteri. Nanopb - protocol buffers with small code size. [online]. c2011 [citováno 2023-11-10]. Dostupné z: https://jpa.kapsi.fi/nanopb/

[18] Language Guide (proto 3). Protocol Buffers Documentation [online]. ©2024 [cit. 2024-05-08]. Dostupné z: https://protobuf.dev/programming-guides/proto3/

[19] Nanopb - Protocol Buffers for Embedded Systems. GitHub [online]. ©2024 [cit. 2024-05-08]. Dostupné z: https://github.com/nanopb/nanopb/blob/master/README.md

[20] NXP. LPC55S6x, Product data sheet [online]. Rev. 2.4. 2022 [citováno 2023-11-10]. Dostupné z: www.nxp.com

[21] NXP. LPCXpresso55S69/55S28 Development Boards, User manual [online]. Rev. 1.6. 2022 [citováno 2023-05-08]. Dostupné z: www.nxp.com

[22] Cortex-M3 Devices Generic User Guide. Https://developer.arm.com [online]. ©1995-2024 [cit. 2024-05-08]. Dostupné z: https://developer.arm.com/documentation/dui0552/a/cortex-m3-peripherals/system-timer--systick

[23] Protocol buffers. In: Martin Kleppmann [online]. [cit. 2024-05-08]. Dostupné z: https://martin.kleppmann.com/2012/12/protobuf.png

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API	Application Programming Interface
CRC	Cyclic Redundancy Check
eRPC	Embedded Remote Procedure Call
gRPC	Google Remote Procedure Call
HTTP	Hypertext Transfer Protocol
ID	Identification
IDL	Interface Description Language
JSON	Javascript Object Notation
PC	Personal computer
Protobuf	Protocol Buffers
QUIC	Quick UDP Internet Connections
REST	Representational State Transfer
RPC	Remote Procedure Call
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
XML	Extensible Markup Language

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

- CD

PŘÍLOHA P I: CD

Přiložené CD obsahuje:

- PDF soubor obsahující bakalářskou práci
- Zdrojový kód testovací aplikace Nanopb