ProtoGen 1.0 — Dokumentacja generatora protokołów sieciowych

Piotr Tabor (pt214569@students.mimuw.edu.pl)

4 czerwca 2008

Historia

Data	Wersja	Autor	Zmiany
2008-05-22	0.1	Piotr Tabor	Pierwsza wersja dokumentu
2008-05-25	0.2	Piotr Tabor	Skończona dokumentacja formatu XML opisującego protokół
2008-05-26	0.3	Piotr Tabor	Spisane pozostałe rozdziały
2008-05-27	0.4	Piotr Tabor	Poprawienie literówek
2008-05-27	0.5	Piotr Tabor	Poprawienie według uwaga dr hab. Krzysztofa Stencla, dopisanie przykładów użycia
2008-05-27	0.6	Piotr Tabor	Przejrzenie i poprawki całości
2008-05-28	0.7	Piotr Tabor	Dodanie informacji o typie „bytes” (2.3.6)
2008-06-01	0.8	Piotr Tabor	Dodanie schematu XML deskryptora w dodatku: 2.2.12

Spis treści

1 Wprowadzenie **2**
1.1 Wstęp . . . 2
1.2 Licencjonowanie . . . 2
1.3 Korzyści płynące ze stosowania tego rozwiązania . . . 2
1.4 Ograniczenia rozwiązania . . . 2
1.4.1 Budowa paczki . . . 2
1.4.2 Big/endian - little/endian . . . 2
1.4.3 Format napisów . . . 3
1.5 Złożona logika . . . 3

2 Dokumentacja użytkownika **4**
2.1 Uruchomienie . . . 4
2.1.1 Przykład . . . 4
2.2 Format deskryptora protokołu . . . 4
2.2.1 Przykładowy deskryptor . . . 4
2.2.2 <protocol> . . . 6
2.2.3 <metadata> . . . 6
2.2.4 <lang id=”java”> . . . 7
2.2.5 <lang id=”cpp”> . . . 7
2.2.6 <enums> . . . 7
2.2.7 <item> . . . 8
2.2.8 <packet-groups> . . . 8
2.2.9 <packets> . . . 9
2.2.10 <field> . . . 9
2.2.11 <description> . . . 10
2.2.12 Schemat „XML Schema” deskryptora protokołu . . . 10
2.3 Proste typy danych . . . 13
2.3.1 Postanowienia ogólne . . . 13
2.3.2 Całkowitoliczbowe: uint8, sint8, uint16, sint16, int32, uint32, uint64, sint64 . . . 13
2.3.3 varuint - Całkowitoliczbowy z kompresją (1,3,5 lub 9 bajtów) . . . 13

2.3.4 string - Łańcuchy tekstu . . . 14
2.3.5 sstring - Krótki łańcuch tekstu . . . 14
2.3.6 bytes - Dane binarne . . . 14

3 Dokumentacja wygenerowanego kodu 15
3.1 Wygenerowany kod dla języka Java . . . 15
3.1.1 pom.xml . . . 15
3.1.2 Typy wyliczeniowe - enumy . . . 15
3.1.3 Grupy paczek . . . 15
3.1.4 Paczki . . . 15
3.1.5 Testy . . . 16
3.1.6 java_protolib . . . 17
3.2 Wygenerowany kod dla języka C++ . . . 17
3.2.1 ./make.defs . . . 17
3.2.2 Autentykacja - ./protocol/auth . . . 17
3.2.3 Strumienie paczek - ./protocol/pstreams . . . 18
3.2.4 Typy wyliczeniowe - ./protocol/enums . . . 18
3.2.5 Grupy pakietów . . . 18
3.2.6 Paczki . . . 18
3.2.7 Testy - ./protocol/tests . . . 18
3.2.8 Gniazda - ./protocol/sockets . . . 18
3.2.9 Strumienie ./protocol/streams . . . 19
3.2.10 Warstwy protokołu - ./protocol/layers/ProtocolLayer0 . . . 19
3.2.11 Klasy pomocnicze . . . 19
3.2.12 Przykładowy kod używający protokół po stronie serwera . . . 19
3.2.13 Przykładowy kod używający protokół po stronie klienta . . . 20

4 Dokumentacja programistyczna 22
4.1 Architektura rozwiązania . . . 22
4.1.1 Java . . . 22
4.1.2 Plexus - zorientowana na komponenty . . . 22
4.2 Proces budowania aplikacji . . . 22
4.3 Dalszy rozwój . . . 23
4.3.1 Złożona logika . . . 23
4.3.2 Więcej języków programowania . . . 23
4.3.3 Generowanie dokumentacji . . . 23
4.3.4 Wsparcie dla wielu wersji pobocznych . . . 23

1 Wprowadzenie

1.1 Wstęp

Generator protokołów jest narzędziem służącym do generowania implementacji protokołu sieciowego w różnych językach programowania w oparciu o zadany opis. Danymi wejściowymi generatora jest specyfikacja protokołu zapisana w pliku XML (patrz: 2.2) oraz wybrany język programowania. Danymi wyjściowymi jest kod źródłowy w wybranym języku programowania umożliwiający wygodne posługiwanie się tym protokołem.

1.2 Licencjonowanie

Projekt „ProtoGen 1.0" jak i wygenerowany przez niego kod jest licencjonowany na zasadach licencji „Apache Software License 2.0" (http://www.apache.org/licenses/LICENSE-2.0).

1.3 Korzyści płynące ze stosowania tego rozwiązania

Zastosowanie tego narzędzia daje następujące korzyści:

- Gwarantuje zgodność na poziomie danych przysyłanych siecią dla implementacji w różnych językach programowania
- Pozwala uniknąć programiście „mechanicznego" tworzenia dużych ilości kodu a zaraz błędów z tym związanych
- Gwarantuje, że cały kod protokołu jest napisany w sposób spójny
- Gwarantuje, że wygenerowany kod jest odporny na podstawowe zagrożenia w komunikacji pomiędzy różnymi architekturami: problem kolejności bajtów w słowie (big-endian/little-endian) oraz problem rozmiaru podstawowych typów danych (architektury od 8 do 64 bitów).
- Tworzy zestawy testów, które umożliwiają sprawdzanie komunikacji także pomiędzy kodem wygenerowanym dla różnych języków programowania.
- Umożliwia aktualizację całego modułu poprzez aktualizację generatora i ponowne wygenerowanie kodu.

1.4 Ograniczenia rozwiązania

ProtoGen nie jest narzędziem, które umożliwia wygenerowanie modułu obsługującego każdy protokół. W obecnej wersji zostały przyjęte poniższe ograniczenia.

1.4.1 Budowa paczki

Każda paczka ma następujący schemat budowy:

Od - do	**Typ/Wartość**	**Zawartość**
$0 \rightarrow a$	'zależy od konfiguracji'	Stała mówiąca o typie paczki, a tym samym określająca format zawartych w nich danych
$a+1 \rightarrow a+4$	uint32	n- Stała określająca ilość danych właściwych zawartych w paczce - wyrażona w bajtach
$a+5 \rightarrow a+5+n$	patrz opis zależny od typu paczki	Dane właściwe paczki zgodne z formatem określonym poprzez typ paczki

Formalnie będziemy mówili, że paczka się składa z dwu-polowego nagłówka oraz ciała. Nagłówek wyznacza typ paczki i rozmiar danych właściwych w niej zawartych, a ciało - to dane interpretowane zależnie od typu paczki.

1.4.2 Big/endian - little/endian

Przyjęto, że wszystkie dane są przesyłane przez sieć w formacie big-endian.

1.4.3 Format napisów

Przyjęto, że wszystkie napisy są przesyłane przez sieć w formacie UTF-8.

1.5 Złożona logika

Obecnie ProtoGen nie wspiera złożonej logiki w konstrukcji pakietów. Niemożliwe jest warunkowanie istnienia pola w zależności od wartości innego pola, a także tworzenia tablicy pól. Obecnie ProtoGen przewiduje możliwość nadpisania wygenerowanego kodu. W ten sposób - nadpisując kod obsługi odpowiedniej paczki - można uzyskać obsługę dowolnie złożonej logiki.

O planowanym rozwiązaniu tego problemu w przyszłych wersjach możesz przeczytać w rozdziale „Dalszy rozwój" (patrz: 4.3)

2 Dokumentacja użytkownika

2.1 Uruchomienie

ProtoGen uruchamiamy będąc w katalogu w którym znajdują się jego pliku uruchomieniowe. W zależności od systemu operacyjnego uruchamiamy protoGen.sh (UNIXy), bądź protoGen.bat (MS Windows) podając następujące parametry:

ścieżka do deskryptora - ścieżka do deskryptora xml opisującego protokół (może być względem bieżącego katalogu). Wskazany plik powinien być w formacie opisanym w rozdziale 2.2.

ścieżka do katalogu docelowego - ścieżka do katalogu w którym będą umieszczane wygenerowane pliki.

język docelowy - język do którego generujemy kod. Obecnie obsługiwane wartości tego parametru to: „cpp" dla C++ oraz „java".

ścieżka do katalogu z plikami nadpisującymi (parametr opcjonalny) - parametr służy do wskazania katalogu w którym powinny się znajdować pliki, którymi chcemy nadpisać działanie generatora protokołu. Może się zdarzyć sytuacja w której protoGen generowałby plik „ABC.xxx". Jeżeli we wskazanym katalogu będzie się znajdował plik „ABC.xxx" to on właśnie zostanie użyty zamiast wygenerowanego pliku. Pliki umieszczamy w tym katalogu bezpośrednio (bez podkatalogów).

Konieczność nadpisania pliku pojawia się najczęściej w sytuacji, gdy mamy paczkę zawierającą skomplikowaną logikę. Wtedy piszemy kod obsługi takiej paczki ręcznie, a następnie zmuszamy protoGen, by go wykorzystał (umieścił w kodzie wynikowym).

2.1.1 Przykład

Polecenie:

```
./protoGen.sh ./conf/loxim.xml ./cpp_proto cpp ./conf/overwritten/cpp
```

Spowoduje wygenerowanie kodu w języku C++ w podkatalogu cpp_proto, w oparciu o deskryptor ./conf/loxim.xml - sprawdzając, czy istnieją nadpisujące pliki w katalogu ./conf/overwritten/cpp.

2.2 Format deskryptora protokołu

Poniżej znajduje się przykładowy deskryptor z opisem poszczególnych znaczników.

2.2.1 Przykładowy deskryptor

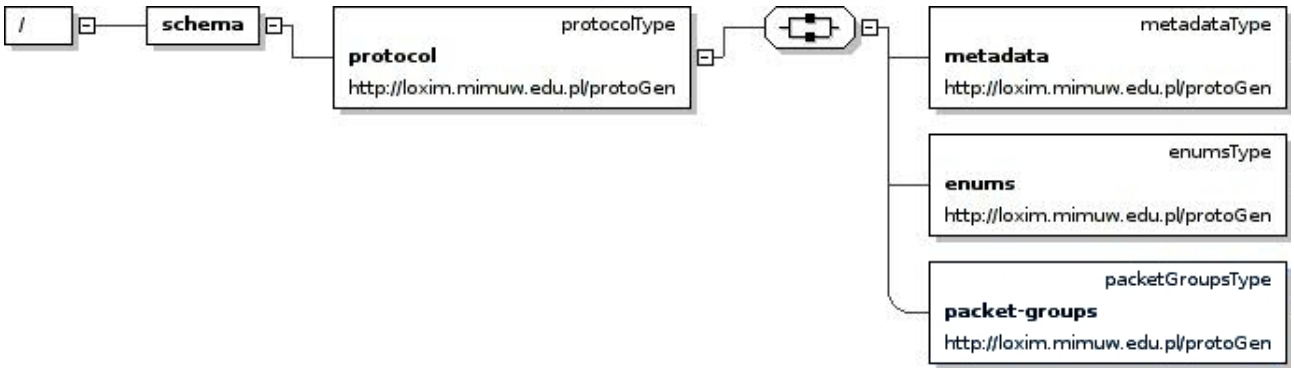
```
<?xml version="1.0" encoding="iso-8859-2"?>
<protocol xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns='http://piotr.tabor.waw.pl/soft/protogen'
    xsi:schemaLocation='http://piotr.tabor.waw.pl/soft/protogen
    http://piotr.tabor.waw.pl/soft/protogen/schema.xsd'>

    <metadata>
        <major-version>2</major-version>
        <minor-version>0</minor-version>
        <languages>
            <lang id="java">
                <packageName>pl.edu.mimuw.loxim.protocol</packageName>
                <artifactId>loxim_protocol</artifactId>
                <version>1.0-SNAPSHOT</version>
            </lang>
        </languages>
    </metadata>
    <enums>
        <enum name="features" as-type="uint64">
            <item name="f_ssl" value="0x0001">
```

```
            <description lang="pl">
                Połaczenie może być szyfrowane metoda SSL
            </description>
        </item>
        ...
    </enum>
    ...
</enums>
<packet-groups>
    <packet-group name="data" id-type="uint32">
        <packets>
            <packet name="uint8"     id-value="1">
                <field name="value" type="uint8"/>
            </packet>
            <packet name="date"    id-value="10">
                <field name="year"  type="sint16"/>
                <field name="month" type="uint8"/>
                <field name="day"   type="uint8"/>
            </packet>
            ...
            <packet name="datetime" id-value="12">
                <field name="date"   type="package" object-ref="data" object
                    -ref-id="date"/>
                <field name="time"   type="package" object-ref="data" object
                    -ref-id="time"/>
            </packet>
            <packet name="binding"     id-value="130">
                <field name="bindingName" type="sstring" />
                <field name="type" type="varuint"/>
                <field name="value" type="package-map" object-ref="data"
                    object-ref-id="type"/>
            </packet>
            ...
        </packets>
    </packet-group>
    <packet-group id-type="uint8">
        <packets>
            <packet name="w_s_hello" id-value="11">
                <field name="protocol_major" type="uint8">
                    <description lang="pl">Numer główny (major) wersji
                        protokołu</description>
                </field>
                <field name="features" type="enum-map" object-ref="features"
                    >
                    <description lang="pl">Mapa bitowa dostępnych cech
                        serwera</description>
                </field>
                <field name="salt" type="sstring" size="20">
                    <description lang="pl">160 bitowy ciąg losowy - używany
                        przez niektóre
                        metody autoryzacji </description>
                </field>
                ...
            </packet>
            <packet name="v_sc_sendvalue" id-value="33">
                <field name="value_id" type="varuint" />
                <field name="flags" type="enum-map" object-ref="
                    send_value_flags" />
```

```
                <field name="value_type" type="varuint" />
                <field name="data" type="package-map" object-ref="data"
                    object-ref-id="value_type"/>
            </packet>
            ...
        </packets>
    </packet-group>
</packet-groups>
</protocol>
```

2.2.2 <protocol>



Rysunek 1: Konstrukcja znacznika <protocol>

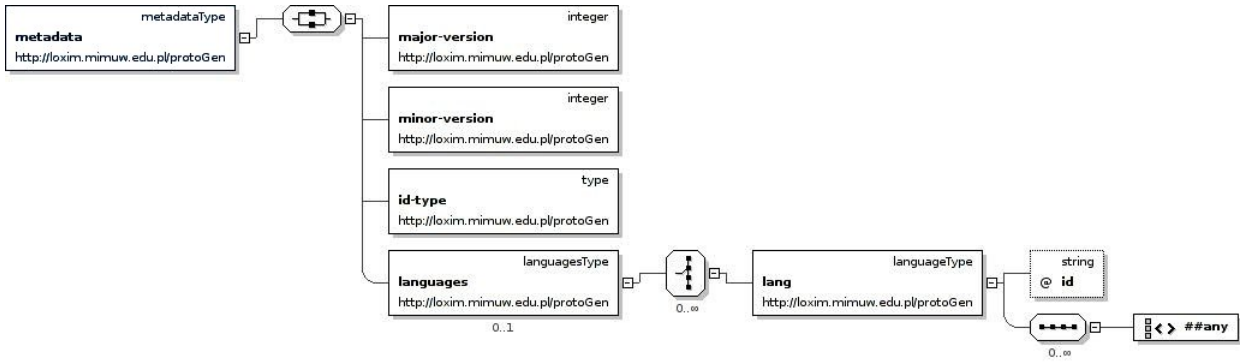
Element <protocol> grupuje znaczniki:

<**metadata**> - opisujące ogólne cechy wygenerowanego protokołu, a także specyficzne cechy zależne od docelowego języka programowania.

<**enums**> - opisuje typy wyliczeniowe, które mogą zostać zastosowane w protokole.

<**packet-groups**> - opisuje grupy paczek (a zatem też paczki), które, bądź budują protokół bezpośrednio, bądź mogą być elementami składowymi paczek.

2.2.3 <metadata>



Rysunek 2: Konstrukcja znacznika <metadata>

Element <metedata> służy opisaniu pewnych ogólnych własności protokołu.

major-version Główna wersja protokołu opisanego w tym deskryptorze. Przyjmuje się, że protokoły o tym samym głównym numerze wersji są ze sobą kompatybilne (co nie znaczy, że można za pomocą nich przekazać te same informacje, ale to że obie implementację będą rozmawiały tak, jakby implementowały tę samą - starszą - wersję protokołu).

minor-version Poboczny numer wersji protokołu.

languages Definicja cech protokołu zależnych od docelowego języka programowania w którym będziemy generowali kod. Szczegóły opisano poniżej.

2.2.4 <lang id="java">

Znacznik ten opisuje dodatkowe atrybuty potrzebne do wygenerowania protokołu dla języka Java.

packageName
Nazwa pakietu (jako grupy klas w języku Java) zawierającego kod protokołu. Należy oddzielić wszystkie elementy ścieżki przy pomocy symbolu '.'.
Wpisana tu nazwa będzie także <groupId> w utworzonym projekcie Maven2 (http://maven.apache.org).

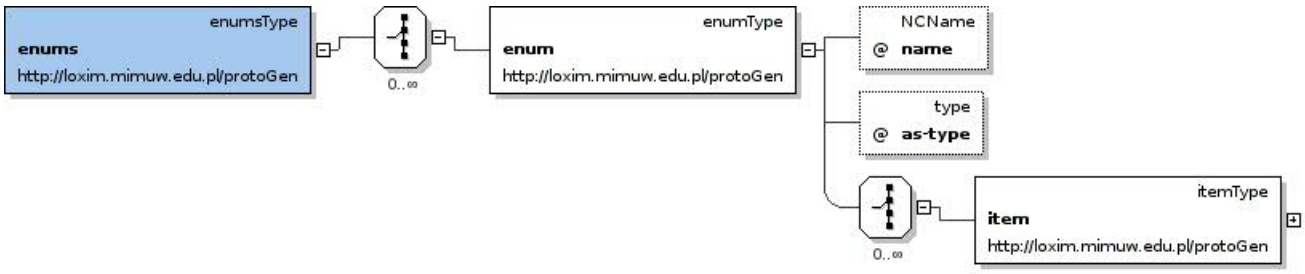
artifactId
Nazwa utworzonego modułu - głównie na potrzeby Maven2.

version
Wersja utworzonego modułu - głównie na potrzeby Maven2.

2.2.5 <lang id="cpp">

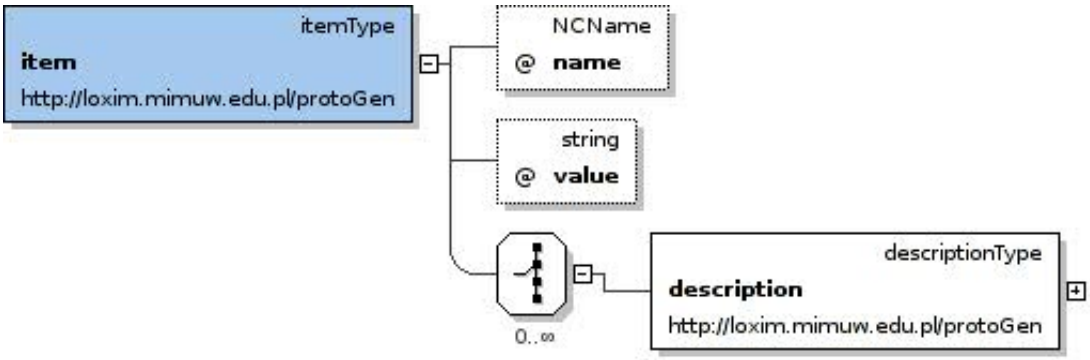
Język C++ nie posiada w obecnej wersji ProtoGen dodatkowych atrybutów.

2.2.6 <enums>



Rysunek 3: Konstrukcja znacznika <enums>

Znacznik enums opisuje typy wyliczeniowe, które mogą zostać wykorzystane w protokole. Typy wyliczeniowe można wykorzystać bezpośrednio - jako pole zawierające jedną wartość z wielu, a także do stworzenia „mapy wartości", czyli do przechowania zbioru wartości (patrz: 2.2.10).



Rysunek 4: Konstrukcja znacznika <item>

Każdy element (<item>) danego typu wyliczeniowego ma przypisaną wartość typu całkowitoliczbowego. Jeśli chcemy dany typ wyliczeniowy wykorzystać jako „mapę" to musimy nadać poszczególnym elementom wartości o bitach zapalonych rozłącznie.

Znacznik <enum> opisuje pojedynczy typ wyliczeniowy.

Jego atrybuty to:

name Nazwa typu wyliczeniowego. Przy pomocy tej nazwy będzie można się odwoływać do tego typu. Od tej nazwy zależy też nazwa wygenerowanej klasy przechowującej ten typ.

as-type Nazwa typu numerycznego (patrz: 2.3), na który będą odwzorcowywane poszczególne enumy.

Ponadto znacznik <enum> budują elementy <item>.

2.2.7 <item>

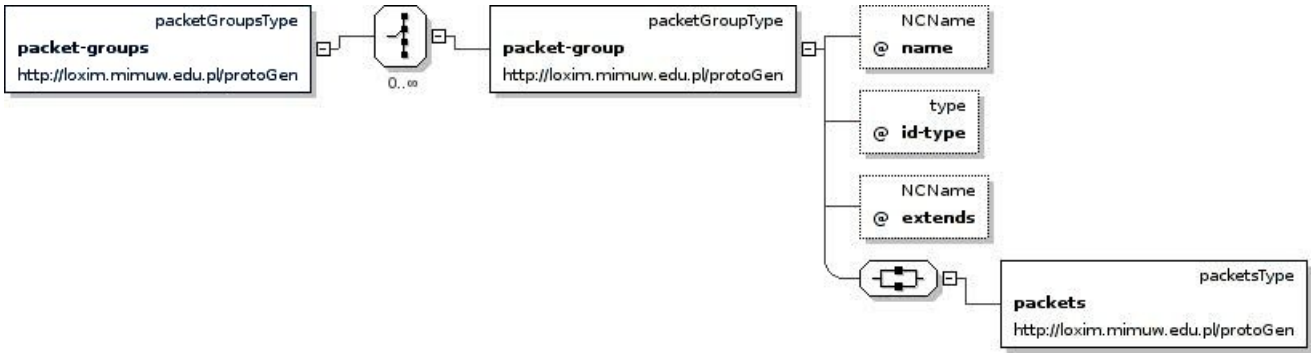
Znacznik item opisuje pojedynczą wartość, którą może przyjąć typ wyliczeniowy. Pojedyncza wartość posiada następujące atrybuty:

name Nazwa elementu typu wyliczeniowego.

value Wartość na którą ten element wyliczeniowy będzie odwzorowywany. Powinna mieścić się w zakresie typu wskazanego w atrybucie „as-type" w znacznika <enum> definiującym ten typ wyliczeniowy. Wartość może być podana w systemie dziesiętnym lub 16-tkowym poprzez poprzedzenie jej symbolami „0x", np. 0x000f4 - co jest szczególnie przydatne przy konstrukcji typów używanych do budowania mapy wartości (patrz: 2.2.10).

Ponadto znacznik <item> może posiadać elementy typu <description> zawierające dane do budowania dokumentacji (patrz: 2.2.11).

2.2.8 <packet-groups>



Rysunek 5: Konstrukcja znacznika <packet-groups>

Znacznik <packet-groups> grupuje znaczniki <packet-group>, które stanowią grupę paczek. Grupa paczek to taki zbiór definicji paczek, która ma różne identyfikatory i w podobny sposób jest wykorzystana. W szczególności dla każdej grupy paczek zostanie wygenerowana fabryka paczek, która pozwala powołać do życia paczkę na podstawie zadanego jej „id".

Zawsze powinna istnieć grupa „główna" (bez nazwy). Jej elementy będą stanowiły podstawowe paczki protokołu. Pozostałe (nazwane) grupy są pomocnicze i mogę służyć do budowania pól innych paczek.

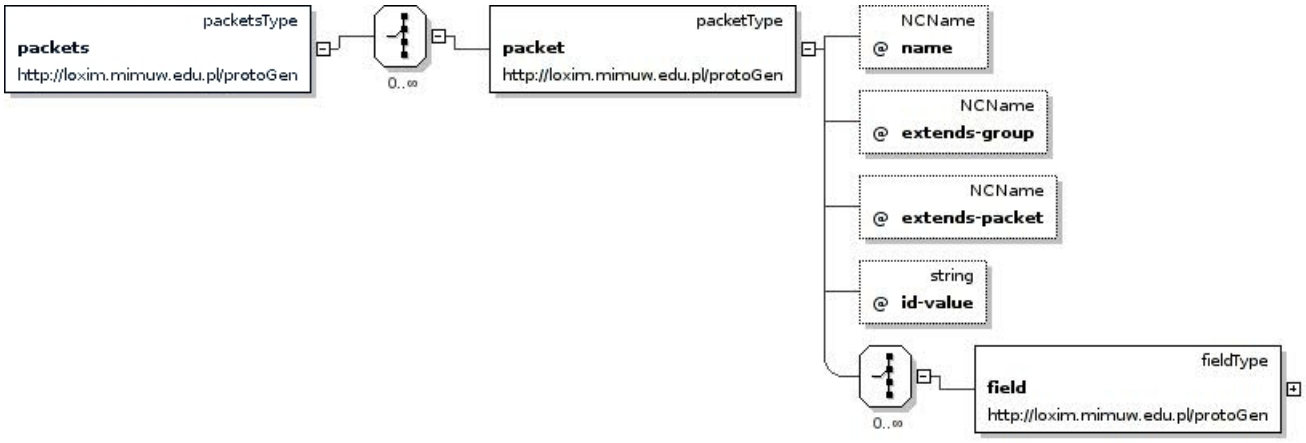
Grupę charakteryzują następujące atrybuty:

name - nazwa grupy. Musi istnieć jedna grupa bez nazwy (grupa „główna")

id-type - nazwa typu całkowitoliczbowego(patrz: 2.3), którego typu będą identyfikatory paczek.

extend (atrybut opcjonalny) - nazwa paczki, która stanowi bazę dla wszystkich paczek tej grupy (o ile nie nadpisano atrybutem extend paczki). Jeśli nie zdefiniowano to paczki będą dziedziczyły domyślnie z głównego typu „Packet" nie zawierającego żadnych pól.

Znacznik <packet-group> zawiera podelementy <packets>.



Rysunek 6: Konstrukcja znacznika <packet>

2.2.9 <packets>

Element <packets> grupuje znaczniki <packet>, które opisują pojedynczą paczkę danych. Paczka danych składa się z pól (patrz: 2.2.10), a ponadto jest opisana następującymi atrybutami:

name - Nazwa paczki

id-value (atrybut opcjonalny) - Wartość identyfikująca paczkę (typu „id-type" z definicji grupy paczek). Jeśli paczka nie posiada id-value - to jest traktowana jako „abstrakcyjna", co oznacza, że może być wykorzystana tylko jako paczka bazowa dla innych paczek.

extends-group (atrybut opcjonalny) - nazwa grupy paczek, w której się znajduje się paczka zdefiniowana w polu „extends-packet".

extends-packet (atrybut opcjonalny) - nazwa paczki z grupy "extends-group", którą deklarowana właśnie paczka rozszerza (o dodatkowe pola).

2.2.10 <field>

Znacznik <field> opisuje pojedyncze pole paczki. Podelementem znacznika <field> może być znacznik <description> (patrz: 2.2.11). Atrybutami znacznika field niezależnie od typu tego pola są:

name - Nazwa pola

type - Typ pola (patrz: 2.3)

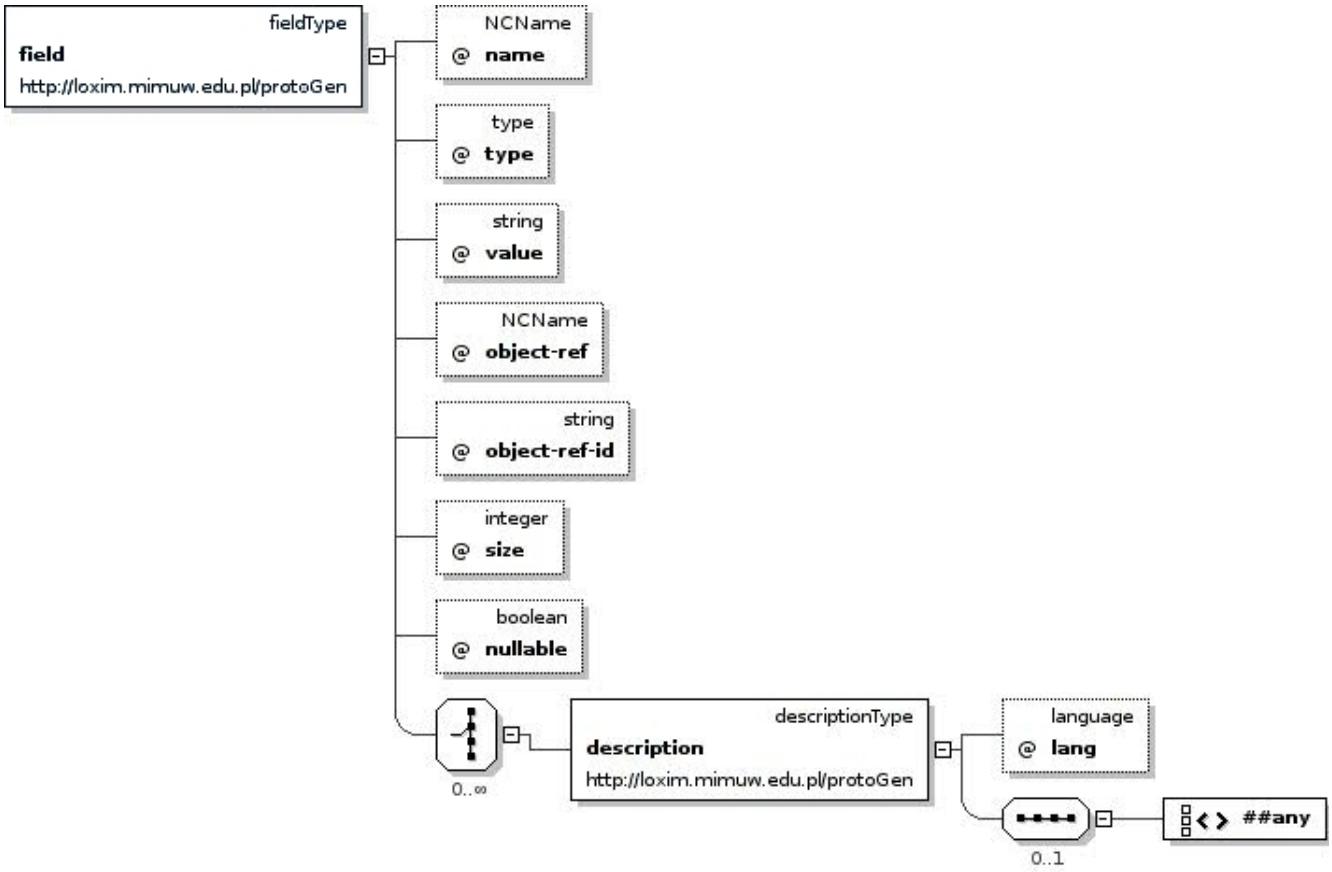
Pozostałe atrybuty zależą od typu pola:

- Dla typów prostych dostępny jest dodatkowo atrybut „value" mogący zawierać domyślną zawartość pola.
- Dla typów: string, sstring, varuint dostępny jest atrybut "nullable" (przyjmujący wartości true lub false), świadczący o tym, czy dopuszczalna jest zawartość NULL dla tego pola.
- Dla typów: string i sstring dostępny jest atrybut size zawierający maksymalną dopuszczalną długość napisu.

Ponadto następuję typy złożone wymagają szczególnego omówienia:

enum - Pole zawiera jedną z wartości z listy wyboru. Atrybut „object-ref" wskazuję na nazwę odpowiedniego „enuma" (patrz: 2.2.6). Pole będzie serializowane jako typ całkowitoliczbowy wskazany w definicji „enuma" o wartości odpowiadającej „id" wybranego elementu z tej listy wyboru.

enum-map - Pole zawiera bitową mapę wartości w listy wyboru. Atrybut „object-ref" wskazuję na nazwę odpowiedniego „enuma" (patrz: 2.2.6). Pole będzie serializowane jako typ całkowitoliczbowy wskazany w definicji „enuma" o wartości odpowiadającej sumie „id" wybranych elementu z tej listy wyboru.



Rysunek 7: Konstrukcja znacznika <fields>

package - Pole zawiera ciało (bez nagłówka) całej innej paczki. Atrybut „object-ref" wskazuję na nazwę grupy paczek, a „object-ref-id" na nazwę konkretnej paczki należącej do tej grupy.

package-map - Pole zawiera ciało całej innej paczki należącej do wskazanej grupy paczek, ale konkretny rodzaj paczki jest wyznaczony przez wartość innego - uprzednio wymienionego pola. Zatem „object-ref" wskazuję na nazwę grupy paczek, a „object-ref-id" na nazwę wcześniej zdefiniowanego pola w bieżącej paczce - które będzie zawierało identyfikator paczki, który ma zostać w bieżącym polu wczytany/zapisany. O polu „package-map" należy myśleć jak o polu „package" z dynamicznie wyznaczanym konkretnym rodzajem paczki.

2.2.11 <description>

Znacznik <description> może występować w wielu miejscach specyfikacji protokołu. Służy on do wygenerowania dokumentacji dotyczącej wskazanego elementu protokołu. Znacznik ten zawiera atrybut „lang", którego wartością powinien być kod języka (kraju) - dwuliterowy, opisujący język w którym został przeprowadzony opis. Zawartością znacznika może być dowolny tekstem także wzbogacony o znaczniki języka HTML.

2.2.12 Schemat (XML Schema) deskryptora protokołu

```
<?xml version="1.0" encoding="iso8859-2" ?>
<xsd:schema xmlns:pg="http://loxim.mimuw.edu.pl/protoGen" xmlns:xsd="http://www.
    w3.org/2001/XMLSchema" targetNamespace="http://loxim.mimuw.edu.pl/protoGen"
    elementFormDefault="qualified" attributeFormDefault="unqualified">

    <!-- ========================= SIMPLE TYPES ========================== -->

    <xsd:simpleType name="type">
        <xsd:restriction base="xsd:string">
```

```
                    <xsd:enumeration value="uint8"/>
                    <xsd:enumeration value="uint16"/>
                    <xsd:enumeration value="uint32"/>
                    <xsd:enumeration value="uint64"/>
                    <xsd:enumeration value="sint8"/>
                    <xsd:enumeration value="sint16"/>
                    <xsd:enumeration value="sint32"/>
                    <xsd:enumeration value="sint64"/>
                    <xsd:enumeration value="varuint"/>
                    <xsd:enumeration value="bool"/>
                    <xsd:enumeration value="double"/>
                    <xsd:enumeration value="bytes"/>

                    <xsd:enumeration value="sstring"/>
                    <xsd:enumeration value="string"/>
                    <xsd:enumeration value="enum"/>
                    <xsd:enumeration value="enum-map"/>

                    <xsd:enumeration value="package"/>
                    <xsd:enumeration value="package-map"/>
    </xsd:restriction>
</xsd:simpleType>

<!-- ================== COMPLEX TYPES ================================ -->
<!-- packets-->
<xsd:complexType name="descriptionType" mixed="true" >
    <xsd:sequence minOccurs="0" maxOccurs="1">
        <xsd:any processContents="lax"/>
    </xsd:sequence>
    <xsd:attribute name="lang" type="xsd:language"/>
</xsd:complexType>

<xsd:complexType name="fieldType">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="description" type="pg:descriptionType"/>
    </xsd:choice>
    <xsd:attribute name="name" type="xsd:NCName"/>
    <xsd:attribute name="type" type="pg:type"/>
    <xsd:attribute name="value" type="xsd:string"/>
    <xsd:attribute name="object-ref" type="xsd:NCName"/>
    <xsd:attribute name="object-ref-id" type="xsd:string"/>
    <xsd:attribute name="size" type="xsd:integer"/>
    <xsd:attribute name="nullable" type="xsd:boolean" use="optional" default
       ="false"/>
 </xsd:complexType>

<xsd:complexType name="packetType">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="field" type="pg:fieldType"/>
        <!--xsd:element name="data-set-field" type="pg:dataSetFieldType"/-->
    </xsd:choice>
    <xsd:attribute name="name" type="xsd:NCName"/>
    <xsd:attribute name="extends-group"  type="xsd:NCName" use="optional" />
    <xsd:attribute name="extends-packet" type="xsd:NCName" use="optional" />
    <xsd:attribute name="id-value" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="packetsType">
```

```
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="packet" type="pg:packetType"/>
    </xsd:choice>
    <!--xsd:attribute name="id-type" type="pg:type"/-->
</xsd:complexType>
<!-- enums -->

<xsd:complexType name="itemType">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="description" type="pg:descriptionType"/>
    </xsd:choice>
    <xsd:attribute name="name" type="xsd:NCName"/>
    <xsd:attribute name="value" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="enumType">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="item" type="pg:itemType"/>
    </xsd:choice>
    <xsd:attribute name="name" type="xsd:NCName"/>
    <xsd:attribute name="as-type" type="pg:type"/>
</xsd:complexType>

<xsd:complexType name="enumsType">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="enum" type="pg:enumType"/>
    </xsd:choice>
</xsd:complexType>

<!-- metadata -->

<xsd:complexType name="languagesType">
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="lang" type="pg:languageType"/>
  </xsd:choice>
</xsd:complexType>

<xsd:complexType name="languageType">
  <xsd:sequence minOccurs="0" maxOccurs="unbounded">
    <xsd:any processContents="skip"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="metadataType">
    <xsd:all>
        <xsd:element name="major-version" type="xsd:integer"/>
        <xsd:element name="minor-version" type="xsd:integer"/>
        <!--xsd:element name="id-type" type="pg:type"/-->
        <xsd:element name="languages" type="pg:languagesType" minOccurs="0"
            maxOccurs="1"/>
    </xsd:all>
    <!--xsd:element name="size-header"-->
</xsd:complexType>

<xsd:complexType name="packetGroupType">
    <xsd:all minOccurs="1" maxOccurs="1">
        <xsd:element name="packets" type="pg:packetsType"/>
```

```
        </xsd:all>
        <xsd:attribute name="name" type="xsd:NCName"/>
        <xsd:attribute name="id-type" type="pg:type" use="optional" default="
            uint8"/>
        <xsd:attribute name="extends" type="xsd:NCName"/>
    </xsd:complexType>

    <xsd:complexType name="packetGroupsType">
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="packet-group" type="pg:packetGroupType"/>
        </xsd:choice>
    </xsd:complexType>

    <!-- protocol -->

    <xsd:complexType name="protocolType">
        <xsd:all>
          <xsd:element name="metadata" type="pg:metadataType" />
          <xsd:element name="enums" type="pg:enumsType" />
          <xsd:element name="packet-groups" type="pg:packetGroupsType" />
          <!-- xsd:element name="data-set-groups" type="pg:dataSetGroupsType"/--
              >
        </xsd:all>
    </xsd:complexType>

    <!-- ================== ROOT ======================== -->

    <xsd:element name="protocol" type="pg:protocolType"></xsd:element>
</xsd:schema>
```

2.3 Proste typy danych

2.3.1 Postanowienia ogólne

- Jeśli w sposób szczególny nie zaznaczono inaczej (a raczej nigdzie nie zaznaczono) wszystkie wartości zapisywane są w formacie Big-endian. W szczególności obejmuje to typy całkowitoliczbowe, rzeczywiste, oraz napisy w kodowaniu UTF-8 także stosują kolejność Big-endian.

2.3.2 Całkowitoliczbowe: uint8, sint8, uint16, sint16, int32, uint32, uint64, sint64

Pierwsza litera determinuje, czy mamy do czynienia z typem ze znakiem (u - unsigned), czy z typem bez znaku (s - signed). Liczba na końcu wyraża długość typu wyrażoną w bitach.

Typy są kodowane oczywiście w kolejności Big-endian.

2.3.3 varuint - Całkowitoliczbowy z kompresją (1,3,5 lub 9 bajtów)

Będzie to typ używany głównie do oznaczania długości stringów i paczek.

Rozwiązanie techniczne zostało zaczerpnięte z protokołu serwera MySQL [1].

Idea jest taka, że krótkie stringi (< 250znaków) będą się pojawiały najczęściej i chcemy mieć najmniejszy narzut na zapisanie długości (jedno bajtowy).

Zatem semantyka pierwszego bajtu jest następująca (w zależności od jego wartości)

0-249 Wartość ta jest jednocześnie wartością wynikową

250 Wartość jest null'em (zostawiamy dla umożliwienia stosowanie tego rozwiązania z systemami relacyjnymi)

251 Wartość ta poprzedza 2-bajtowe (uint16) pole w wartością właściwą

252 Wartość ta poprzedza 4-bajtowe (uint32) pole z wartością właściwą

253 Wartość ta poprzedza 8-bajtowe (uint64) pole z wartością właściwą. Nie należy jednak korzystać z wartości większych niż $2^{63} - 1$ (MAX_SINT64), ze względu na to, że w Javie nie są obsługiwane 8 bajtowe typy bez znaku.

2.3.4 string - Łańcuchy tekstu

Służy do przesyłania tekstów. Teksty te muszą być kodowane za pomocą UTF-8.

Format wartości typu string jest następujący: Najpierw idzie pole typu varuint - opisujące długość w bajtach następującego potem ciągu w UTF-8 (Big-endian).

2.3.5 sstring - Krótki łańcuch tekstu

Jest to tak naprawdę wariant typu string, ale ograniczony do łańcuchów nie dłuższych niż 249 bajtów. Czyli wtedy pole oznaczające długość ma jeden bajt. Jego wewnętrzna reprezentacja zupełnie się nie różni od typu string - został on wyróżniony formalnie - ze względu na uproszczenie notacji używanej przy opisach formatów poszczególnych paczek, a także ze względów bezpieczeństwa - gdyż pozwala zmniejszyć ryzyko typu DOS poprzez zawężenie wielkości danych, które mogą być zinterpretowane przez serwer.

2.3.6 bytes - Dane binarne

Służy do przesyłania danych binarnych (tablic bajtów).

Format wartości typu bytes jest następujący: Najpierw jest wysyłane pole typu varuint - opisujące długość w bajtach następującego potem ciągu bajtów.

3 Dokumentacja wygenerowanego kodu

3.1 Wygenerowany kod dla języka Java

Generator protokołów generuje kod w języku Java w wersji nie starszej niż Java 1.5, ponieważ korzysta z typów wyliczeniowych (enums) i szablonów (generics).

Zakładając, że w metadanych opisu protokołu dla języka java zadeklarowano packageName jako 'taki.sobie.protokol' (patrz: 2.2.4) to zostanie wygenerowany następujący projekt:

3.1.1 pom.xml

Pom.xml (Project Object Model) jest to plik opisujący jak kompilować ten moduł za pomocą narzędzia „Maven2". Wskazuje on w szczególności biblioteki od których projekt zależy.

Moduł protokołu wygenerowany przez ProtoGen zależy od biblioteki: pl.edu.mimuw.loxim.protogen.lang.java.protolib:java_protolib, która zawiera definicję odpowiednich strumieni i klas bazowych (patrz: 3.1.6).

Aby zbudować moduł wystarczy (zakładając, że użytkownik posiada zainstalowanego Maven'a 2 [1]) wykonać polecenie „mvn package" w katalogu zawierającym plik pom.xml. Wyniki procesu budowania zostaną umieszczone w katalogu target.

3.1.2 Typy wyliczeniowe - enumy

Każdy „enum" (typ wyliczeniowy) zostanie zapisany do katalogu ./src/main/java/taki/sobie/protokol/enums jako „Enum" języka Java (wersja 1.5+).

Typ „enum-map" jest odwzorcowywany na odpowiedni „java.util.EnumSet<typ bazowy>" (patrz: 2.2.10).

3.1.3 Grupy paczek

Paczki grupy głównej zostaną zapisane w folderze ./src/main/java/taki/sobie/protokol/packages, a paczki z pozostałych grup w folderze ./src/main/java/taki/sobie/protokol/packages_[nazwa_grupy].

W każdym takim folderze powstanie także klasa będąca fabryką paczek, która udostępnia metodę:

public Package createPackage(long package_id) throws ProtocolException

Metoda ta tworzy paczkę w zależności od danego identyfikatora paczki.

3.1.4 Paczki

Dla każdej paczki zadeklarowanej w opisie protokołu jest tworzona odpowiednia klasa paczki. Wszystkie paczki rozszerzają następującą klasę:

```
public abstract class Package {

  /**
   * Metoda wypełnia pola paczki w oparciu o zserializowane dane zawarte w
     tablicy danej parametrem b.
   * Tablica "b" powinna zawierać pełny opis paczki - włącznie z jej nagłówkiem.
   * */
  public void deserialize(byte[] b) throws ProtocolException;

  /**
   * Metoda sprawdza, czy dana paczka jest równa paczce na której wywołujemy tę
     metodę.
   * Za równe uznajemy te paczki, które są tego samego typu i których pola są
     równe co do zawartości.
   *
   * Metoda powinna być nadpisana przez deklarację każdej paczki.
   */
  public boolean equals(Object obj) {};
```

[1] http://maven.apache.org/download.html

```
    /**
     * Metoda zapisuje do danego obiektu piszącego do strumienia wszystkie pola
         paczki (bez nagłówka).
     */
    public void deserializeContent(PackageInputStreamReader reader);

    /**
     * Metoda wczytuje paczkę z danego obiektu czytającego strumień.
     * Wczytywane są wszystkie pola paczki (bez nagłówka).
     */
    public void serializeContent(PackageOutputStreamWriter writer);

    /**
     * Metoda zapisuje do danego obiektu piszącego do strumienia wszystkie pola
         paczki (bez nagłówka).
     *
     * Metoda powinna być nadpisana przez deklarację każdej paczki.
     */
    protected abstract void deserializeW(PackageInputStreamReader reader)
        throws ProtocolException;

    /**
     * Metoda wczytuje paczkę z danego obiektu czytającego strumień.
     * Wczytywane są wszystkie pola paczki (bez nagłówka).
     *
     * Metoda powinna być nadpisana przez deklarację każdej paczki.
     */
    protected abstract void serializeW(PackageOutputStreamWriter writer)
        throws ProtocolException;
}
```

Ponadto każda paczka zawiera następujące elementy:

- Publiczną stałą ID zawierającą identyfikator paczki
- Bezparametrowy konstruktor paczki
- Konstruktor paczki, który przyjmuje wszystkie pola paczki jako swoje parametry
- Gettery i settery dla wszystkich pól paczki

3.1.5 Testy

Generator wygeneruje także testy:

./src/test/java/taki/sobie/protokol/tests/TestRunnerRec.java -
Klasa języka java dająca się uruchomić (zawiera metodę main), która uruchamia proces, który otwiera gniazdo TCPIP na wskazanym porcie i sprawdza, czy otrzymane paczki są zgodne z oczekiwanymi. Jedynym parametrem jaki można przekazać uruchamiając tę klasę jest numer portu na którym chcemy otworzyć gniazdo serwera.

Program ten można wykorzystywać do testów komunikacji z implementacjami protokołu dla innych języków programowania.

./src/test/java/taki/sobie/protokol/tests/TestRunnerSender.java -
Klasa języka java dająca się uruchomić (zawiera metodę main), która uruchamia proces, który wysyła na wskazany port na wskazanym serwerze paczki testowe. Uruchamiając musimy przekazać dwa parametry: nazwę hosta docelowego (lub adres IP), a także port do którego chcemy się podłączyć.

Program ten można wykorzystywać do testów komunikacji z implementacjami protokołu dla innych języków programowania.

./src/test/java/taki/sobie/protokol/tests/TestPackagesFactory.java -
Zawiera konstruktor wygenerowanych paczek testowych. Stanowi bazę dla innych testów.

./src/test/java/taki/sobie/protokol/tests/TestPackages.java -
Jest JUnitowym testem, który zapisuje wszystkie paczki testowe do strumienia, a następnie je wszystkie wczytuje i sprawdza, czy paczki są równe. Ten test jest używany w fazie testów w procesie budowania modułu przez Maven 2.

3.1.6 java_protolib

Biblioteka java_protolib.jar jest plikiem, który zawiera niezależną od konkretnego protokołu definicję podstawowych klas i interfejsów.

Poniżej listujemy kluczowe klasy znajdujące się w tej bibliotece:

pl.edu.mimuw.loxim.protogen.lang.java.template.layers.ProtocolLayer0
Podstawowa klasa używana w aplikacji do obsługi protokołu. Zawiera metody wysyłające przekazaną paczkę, a także czekającą i odczytującą paczkę. Klasa ponadto automatycznie odpowiada na paczkę „Ping" paczką „Pong".

pl.edu.mimuw.loxim.protogen.lang.java.template.exception.ProtocolException
Wyjątek rzucany gdy coś w protokole pójdzie nie tak jak powinno.

pl.edu.mimuw.loxim.protogen.lang.java.template.pstreams.PackageOutputStream
Strumień zakładany na OutputStream'ie, który udostępnia metodę writePackage(Package p).

pl.edu.mimuw.loxim.protogen.lang.java.template.pstreams.PackageInputStream
Strumień zakładany na InputStreamie, który udostępnia metodę readPackage().

pl.edu.mimuw.loxim.protogen.lang.java.template.streams.PackageOutputStreamWriter
Strumień zakładany na OutputStream'ie, który udostępnia metody pozwalające zapisać proste typy danych (patrz 2.3).

pl.edu.mimuw.loxim.protogen.lang.java.template.streams.PackageInputStreamReader
Strumień zakładany na InputStream'ie, który udostępnia metody pozwalające odczytać proste typy danych (patrz 2.3).

pl.edu.mimuw.loxim.protogen.lang.java.template.auth.AuthPassMySQL
Klasa pomocnicza oferująca wsparcie dla przeprowadzania autoryzacji taką metodą jak robi to serwer bazy danych MySQL („Password Algorithm" [**?**])

pl.edu.mimuw.loxim.protogen.lang.java.template.pstreams.PackagesFactory
Interfejs, który implementują wszystkie fabryki pakietów (patrz: 3.1.3)

pl.edu.mimuw.loxim.protogen.lang.java.template.ptools.Package
Klasa bazowa dla wszystkich paczek.

3.2 Wygenerowany kod dla języka C++

Kod dla języka C++ jest samodzielny (jedyną zależnością jest OpenSSL potrzeby do działającej autoryzacji metodą zastosowaną w serwerze MySQL). Buduje się go za pomocą polecenia make wydanego w katalogu „protocol".

3.2.1 ./make.defs

Zawiera definicję narzędzi i bibliotek używanych przez make.

3.2.2 Autentykacja - ./protocol/auth

W tym folderze zawarte są klasy pomocnicze potrzebne do przeprowadzenia autentykacji. Obecnie zaimplementowana jest tylko klasa AuthPassMySQL wspierająca autentykacją metodą zastosowaną w serwerze MySQL i wykorzystująca sumy kontrolne SHA1.

3.2.3 Strumienie paczek - ./protocol/pstreams

PackageOutputStream - Strumień zakładany na OutputStream;ie, który udostępnia metodę writePackage(Package* p).

PackageInputStream - Strumień zakładany na InputStream'ie, który udostępnia metodę readPackage() wczytującą dany pakiet ze strumienia.

3.2.4 Typy wyliczeniowe - ./protocol/enums

Każdy typ wyliczeniowy definiuje odpowiedni „enum" języka C++. Tworzony jest także typ mapy (zawierający pole bitowe dla każdej wartości typu wyliczeniowego).

Ponadto tworzona jest także fabryka umożliwiająca stworzenie instancji enuma lub mapy enumów na podstawie przekazanej wartości całkowitej, a także zapisanie enuma lub mapy enumów jako liczbę.

3.2.5 Grupy pakietów

Paczki grupy głównej zostaną zapisane w katalogu: ./protocol/packages, a grup nazwanych w ./protocol/packages_[nazwa_grupy].

Dla każdej grupy zostanie wygenerowana klasa, będąca fabryką paczek w zależności od przekazanego identyfikatora paczki.

3.2.6 Paczki

Dla każdej paczki zostaje zaimplementowana klasa analogiczna do tej opisanej dla języka Java (patrz: 3.1.4).

3.2.7 Testy - ./protocol/tests

./protocol/tests/TestRunnerRec Uruchamia proces, który otwiera gniazdo TCPIP na wskazanym porcie i sprawdza, czy otrzymane pakiety są zgodne z oczekiwanymi. Jedynym parametrem jaki można przekazać uruchamiając tę klasę jest numer portu na którym chcemy otworzyć gniazdo serwera.

Program ten można wykorzystywać do testów komunikacji z implementacjami protokołu w innych językach programowania.

./protocol/tests/TestRunnerSender Uruchamia proces, który wysyła na wskazany port na wskazanym serwerze pakiety testowe. Uruchamiając musimy przekazać dwa parametry: nazwę hosta docelowego (lub adres IP), a także port do którego chcemy się podłączyć.

Program ten można wykorzystywać do testów komunikacji z implementacjami protokołu w innych językach programowania.

./protocol/tests/TestPackagesFactory Jest to fabryka wygenerowanych pakietów testowych. Stanowi bazę dla innych testów.

3.2.8 Gniazda - ./protocol/sockets

Katalog zawiera prostą, obiektową implementację gniazd. Jest ona wzorowana na tej dostępnej w języku Java i umożliwia łatwe używanie gniazd sieciowych (zarówno serwerowych jak i klienckich) i otwieranie strumieni na nich.

- ./protocol/sockets/TCPIPServerSocket
- ./protocol/sockets/TCPIPServerSingleSocket
- ./protocol/sockets/AbstractSocket
- ./protocol/sockets/TCPIPClientSocket

3.2.9 Strumienie ./protocol/streams

Katalog zawiera elementarne, obiektowe implementacje strumieni - wzorowane na języku Java.

./protocol/streams/AbstractOutputStream,AbstractInputStream - abstrakcyjne klasy bazowe dla strumieni

./protocol/streams/DescriptorInputStream,DescriptorOutputStream - prosta implementacja strumieni oparta o zapis i odczyt ze wskazanego deskryptora systemu operacyjnego (pliku, urządzenia, gniazda sieciowego, łącza).

./protocol/streams/PriorityOutputStream Klasa okalająca (wrapper) strumienia wyjściowego, który udostępnia dodatkową metodę writePriority(char*), która przy wielu odwołaniach do tego obiektu (z wielu wątków) zostanie wykonana przed wykonaniami zapisu w zwykłych metodach write(char*) (odpowiednia synchronizacja na semaforach).

3.2.10 Warstwy protokołu - ./protocol/layers/ProtocolLayer0

Podstawowa klasa używana w aplikacji do obsługi protokołu. Zawiera metody wysyłające przekazaną paczkę, a także metodę czekającą na i odczytującą paczkę. Klasa ponadto automatycznie odpowiada na paczki „Ping" paczkami „Pong".

3.2.11 Klasy pomocnicze

./protocol/ptools/Endianness -
Klasa zawiera metody ułatwiające konwersję Little/Big Endian dla typów prostych

./protocol/ptools/StringBufferStack -
Klasa umożliwia szybką konstrukcje napisów (buforów znaków) poprzez sklejanie napisów (tablic znaków).

./protocol/ptools/Package -
Klasa bazowa dla wszystkich paczek

./protocol/ptools/CharArray -
Tablica znaków z określonym rozmiarem

./protocol/ptools/PackageBufferWriter -
Klasa zawierająca metody zapisujące do danego strumienia typy proste.

./protocol/ptools/PackageBufferReader -
Klasa zawierająca metody odczytujące z danego strumienia typy proste.

3.2.12 Przykładowy kod używający protokół po stronie serwera

```
#include <stdio.h>
#include <stdlib.h>

#include "./ptools/Package.h"
#include "./sockets/TCPIPServerSocket.h"
#include "./layers/ProtocolLayer0.h"
#include "./packages/W_c_helloPackage.h"
#include "./packages/A_sc_byePackage.h"

using namespace protocol;

int main(int argc,char** argv)
{
  printf("Opening server socket on port: %d\n",atoi(argv[1]));

  TCPIPServerSocket* serversocket=new TCPIPServerSocket(NULL,atoi(argv[1]));
  serversocket->bind();
```

```
  AbstractSocket *socket=serversocket->accept();

  ProtocolLayer0 *pl0=new ProtocolLayer0(socket);

  Package* p=pl0->readPackage();
  if((p)&&(p->getPackageType()==ID_W_c_helloPackage))
  {
    printf("Received package\n");
    delete p;

    printf("Sending response...\n");
    p=new A_sc_byePackage();
    pl0->writePackage(p);
    delete p;
  }else{
    printf("Unexpected package\n");
    if (p)
      delete p;
  }

  delete pl0;
  socket->close();
  delete socket;
  serversocket->close();
  delete serversocket;
  printf("Finished");
}
```

3.2.13 Przykładowy kod używający protokół po stronie klienta

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

#include "sockets/TCPIPClientSocket.h"
#include "layers/ProtocolLayer0.h"
#include "./packages/W_c_helloPackage.h"
#include "./packages/A_sc_byePackage.h"
#include "./enums/CollationsFactory.h"

using namespace protocol;

int main(int argc, char** argv) {
  printf("Opening server socket on: %s:%d\n", argv[1], atoi(argv[2]));

  TCPIPClientSocket* clientsocket=new TCPIPClientSocket(argv[1],atoi(argv[2]));
  clientsocket->connect();

  ProtocolLayer0 *pl0=new ProtocolLayer0(clientsocket);

  printf("Sending invitation...\n");
  Package * p=new W_c_helloPackage(getpid(),new CharArray("testClient"),
      NULL,NULL,NULL, coll_unicode_collation_alghorithm_uts10,2);
  pl0->writePackage(p);
  delete pl0;

  printf("Waiting for package...");
  p=pl0->readPackage();
```

```
  printf("... got it");
  if ((p)&&(p->getPackageType()==ID_A_sc_byePackage)) {
    printf("Received package\n");
    delete p;
  } else {
    printf("Unexpected package\n");
    if (p)
      delete p;
  }

  delete pl0;
  clientsocket->close();
  delete clientsocket;
  printf("Finished");
}
```

4 Dokumentacja programistyczna

4.1 Architektura rozwiązania

4.1.1 Java

ProtoGen został napisany w języku Java co umożliwia używanie go na większości platform sprzętowych i z wykorzystaniem wielu systemów operacyjnych.

4.1.2 Plexus - zorientowana na komponenty

ProtoGen został napisany w architekturze zorientowanej na komponenty. Do zarządzania komponentami używamy biblioteki Plexus (http://plexus.codehaus.org). Z punktu widzenia użytkownika i programisty, zastosowanie tej architektury umożliwia stosunkowo łatwe dodanie obsługi nowego języka generowanego kodu. Wymaga ono jedynie stworzenia implementacji interfejsu: „pl.edu.mimuw.loxim.protogen.api.PartialLanguageGenerator", a następnie zadeklarowanie jej w lokalnym deskryptorze components.xml jar'a zawierającego tę implementację jako:

```
<?xml version="1.0" encoding="iso8859-2"?>
<component-set>
    <components>
        <component>
            <role>pl.edu.mimuw.loxim.protogen.api.PartialLanguageGenerator</role
                >
            <role-hint>NowyJęzyk</role-hint>
            <implementation>pl.edu.mimuw.loxim.protogen.lang.nl.
                NewLanguageGenerator</implementation>
        </component>
        <component>
            <role>pl.edu.mimuw.loxim.protogen.api.LanguageGenerator</role>
            <role-hint>NowyJęzyk</role-hint>
            <implementation>pl.edu.mimuw.loxim.protogen.api.
                BasicLanguageGenerator</implementation>
            <requirements>
                <requirement>
                    <role>pl.edu.mimuw.loxim.protogen.api.
                        PartialLanguageGenerator</role>
                    <role-hint>NowyJęzyk</role-hint>
                </requirement>
            </requirements>
        </component>
    </components>
</component-set>
```

W momencie, gdy taki jar znajdzie się w classpath'ie uruchamianego ProtoGen'a, to język „NowyJęzyk" będzie dostępny jako jeden z docelowych język dla generowanego kodu.

Reasumując, główną korzyścią zastosowania architektury zorientowanej na komponenty, jest umożliwienie zewnętrznym dostawcom dodawanie implementacji nowych języków bez żadnej ingerencji w dostępny dotychczas kod.

4.2 Proces budowania aplikacji

1. Zainstalować Maven2 (http://maven.apache.org/download.html)
2. Wejść do katalogu głównego ProtoGen'a i wydać polecenie „mvn clean package assembly:assembly"
3. W podkatalogu „target/protogen-x.xx.xx-bin.dir" będzie się znajdowała wersja dystrybucyjna projektu.

4.3 Dalszy rozwój

4.3.1 Złożona logika

Obecnie ProtoGen nie wspiera złożonej logiki w konstrukcji pakietów. Niemożliwe jest warunkowanie istnienia pola w zależności od wartości innego pola, a także tworzenia tablicy pól. Obecnie ProtoGen przewiduje możliwość nadpisania wygenerowanego kodu. W ten sposób można - nadpisując kod obsługi odpowiedniej paczki - uzyskać obsługę dowolnie złożonej logiki.

Żeby zminimalizować konieczność wykorzystania takiego obejścia, a tym samym uzyskać „pełność" rozwiązania przewidujemy w przyszłych wersjach możliwość wprowadzenia następujących atrybutów do pola field (patrz: 2.2.10):

if-field — Zawiera nazwę pola z bieżącej paczki (wcześniej zadeklarowanego), którego wartość będziemy porównywali do „if-value".

if-value — Jeżeli zawartość pola o nazwie zadanej atrybutem „if-field" jest równa wartości „if-value" to opisywane pole występuje w tej paczce. W przeciwnym przypadku nie występuje.

repeat — Zawiera nazwę pola całkowitoliczbowego, które zawiera informację ile razy bieżące pole jest powtórzone.

4.3.2 Więcej języków programowania

System jest przystosowany do rozszerzania o generatory do innych języków programowania bez modyfikacji kodu źródłowego projektu bazowego. Dodanie obsługi nowego języka programowania wymaga napisania komponentu systemu Plexus implementującego interfejs „pl.edu.mimuw.loxim.protogen.api.PartialLanguageGenerator".

Oczywistym jest, że dodanie generatorów do języków programowania takich jak, C#, Python, Ruby, PHP, Ocaml istotnie zwiększy użyteczność tego projektu.

4.3.3 Generowanie dokumentacji

Podobnie jak z dodatkowymi językami programowania, wydaje się istotną z punktu widzenia użytkownika możliwość uzyskania aktualnej dokumentacji protokołu. Cechą pożądaną jest by ProtoGen był wstanie wygenerować dokumentację - najlepiej do wielu najpopularniejszych formatów (HTML, LaTeX, PDF, ODF).

By uniknąć ręcznej obsługi tych wszystkich formatów - wydaje się uzasadnionym użycie narzędzia typu Doxia (http://maven.apache.org/doxia), które umożliwia konwersje pomiędzy wieloma formatami „bogatego" tekstu.

4.3.4 Wsparcie dla wielu wersji pobocznych

Zarówno definicję paczki, jak i pola można rozszerzyć o atrybut „since-minor-version", który będzie specyfikował od której wersji pobocznej protokołu dane pole występuje.

Dodatkowo wymagamy, by kolejność wartości „since-minor-version" zgadzała się z kolejnością pól w paczce i by każde pole posiadające ten atrybut posiadało także zadeklarowaną wartość domyślną (pole „value"). Warunki te są wystarczające, by zachować zgodność protokołu pomiędzy różnymi wersjami pobocznymi.

Literatura

[1] MySQL Protocol - version 10. Ian Redfern, [03.12.2006], (http://www.redferni.uklinux.net/mysql/MySQL-Protocol.html)