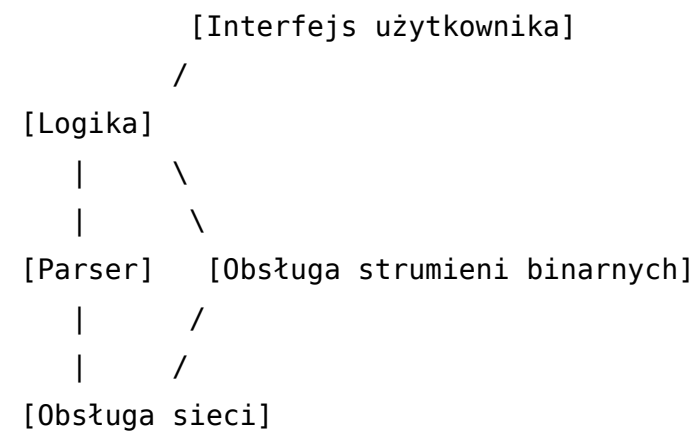
Paulina Motyl
Paweł Pawliński
Michał Przyłuski
Marcin Szewczyk

SHYSHARK

Shyshark jest klientem sieci TOAD, napisanym w języku C++ na systemu zgodne z POSIX (obecnie przetestowano na systemach: GNU/Linux, Solaris i w środowisku Cygwin). Aplikacja implementuje obsługę zasobu 'file', umożliwiającym wymianę plików między dowolnymi węzłami. Architektura pozwala na łatwą rozbudowę funkcjonalności sieciowej i implementację własnego interfejsu użytkownika.

1. Budowa

Schemat budowy aplikacji z podziałem na moduły/warstwy:



a) **Obsługa sieci**

Warstwa odpowiadająca za komunikację sieciową z innymi węzłami. Tłumaczy strumień TCP z postaci binarnej na nagłówek XML i strumień danych binarnych.

Pliki:

Socket.h/cpp

b) **Parser**

- Parsuje nagłówek XML, weryfikując jego poprawność i tłumacząc na postać prostych zdarzeń zrozumiałych przez warstwę logiki – NetEvents (NetEvent.h).

- Tłumaczy zdarzenia z warstwy logicznej - ParserEvents (ParserEvent.h) na pełną wiadomość (nagłówek XML z opcjonalnym strumieniem binarnym).

Pliki:

Parser.h/cpp

ParserEvent.h/cpp

ParsedData.h

c) **Obsługa strumieni binarnych**

Pośredniczy w wymianie danych między warstwami. Podstawowe klasy BinReader i BinWriter zapewniają warstwę abstrakcji pomiędzy odbieranymi/wysyłanymi danymi binarnymi a ich lokalną reprezentacją.

Pliki:

BinStream.h/cpp

d) **Logika**

Kontroluje wszystkie połączenia, podejmuje decyzje o wysyłaniu wiadomości do innych węzłów. Przyjmuje zdarzenia od parsera i interfejsu użytkownika. Zarządza całością wymiany zasobów (plików) na danym kliencie. Implementacja: główna klasa MainContext będąca globalnym interfejsem danych dla zdarzeń przychodzących do logiki.

Pliki:

Logic.h/cpp

NetEvent.h/cpp

File.h/cpp

Request.h/cpp

Time.h/cpp

e) **Interfejs użytkownika**

Wysyła do warstwy logiki żądania użytkownika, takie jak wyszukiwanie lub pobieranie plików - Requests (Request.h). Odbiera informacje o zdarzeniach w aplikacji - UIEvents (UIEvent.h) - otrzymane wyniki wyszukiwania, zmiany

połączeń aplikacji, etc.

2. Rozszerzalność

Pomiędzy różnymi modułami aplikacji istnieje dobrze określone API, w dużej części oparte na zdarzeniach, pozwalające na łatwą modyfikację lub zamianę danego modułu. Aby rozszerzyć funkcjonalność protokołu o nowe funkcje lub zasoby, wystarczy zmodyfikować warstwę logiki (semantyka) i parser (syntaktyka).

3. Wątki

Shyshark jest aplikacją wielowątkową (wątki POSIX). Klasa Thread (Thread.h/cpp) tworzy interfejs C++ dla wywołań pthreads. Wątki komunikują się wyłącznie poprzez kolejki zdarzeń (Queue.h/cpp). Struktura aplikacji z punktu widzenia wielowątkowości:

- Najpierw startowany jest wątek warstwy logicznej.

- Jeden wątek przyjmuje połączenia przychodzące na ustalonym porcie.

- Każde połączenie przychodzące lub wychodzące jest obsługiwane w osobnym wątku. Wątki połączeń wychodzących tworzone są z wątku warstwy logicznej, dla połączeń przychodzących - w wątku nasłuchującym.

- UI działa na dwóch osobnych wątkach - jeden nasłuchuje zdarzeń UIEvent na kolejce, drugi czyta znaki z klawiatury.

Łączna liczba działających wątków to 4 + liczba aktywnych połączeń z innymi węzłami.

4. Komunikacja pomiędzy modułami aplikacji

Cała komunikacja pomiędzy różnymi wątkami opiera się na wysyłanych i odbieranych zdarzeniach. Wykorzystujemy w tym celu kolejki - klasa Queue.

Każde zdarzenie obsługiwane jest przez odpowiednią warstwę aplikacji i właściwym kontekście (warstwa może różnie obsługiwać zdarzenia tego samego typu, zależnie od wybranych kryteriów - adres nadawcy, uzgodnione rozszerzenia protokołu, etc). Klasy *Context zawierają interfejs potrzebny do obsłużenia wszystkich przychodzących zdarzeń. Rozbudowany interfejs kontekstu potrzebny jest w sytuacji kiedy następuje dużo interakcji pomiędzy przychodzącymi zdarzeniami - tak jest w przypadku klasy MainContext reprezentującej warstwę logiczną.

Komunikacja pomiędzy klasami/obiektami w ramach jednego wątku odbywa się poprzez ustalone interfejsy (patrz: pliki nagłówkowe) - nie ma ryzyka związanego z dostępem przez wiele wątków do jednego obiektu, ponieważ każdy z nich jest wykorzystywany w danej chwili przez jeden tylko wątek.

5. Zdarzenie (klasa Event)

Podstawowym elementem klasy Event są funkcje process(...), które biorąc jako argument aktualny kontekst wykonania w pełni obsługują dane zdarzenie.

Przykład: zakładając poniższą hierarchię klas:

```
Event
  ↑
MainEvent (zdarzenia obsługiwane przez logikę aplikacji)
  ↑
NetEvent (przychodzące zdarzenia sieciowe)
  ↑
NetPing (przychodzące żądanie typu ping)
```

Funkcja NetPing::process(MainContext) powinna zweryfikować czy dany węzeł jest uprawniony do wysłania żądania ping, a następnie utworzyć i wysłać zdarzenie-

odpowiedź dla danego połączenia (warstwa parsera).

6. Przepływ zdarzeń

Zdarzenia mogą być przekazywane do innych warstw i wątków o ile są przez nie obsługiwane. Teoretycznie większość zdarzeń NetEvent powinna być obsługiwana przez warstwę Main i UI, niektóre OutEvent również mogą być obsługiwane przez UI, chociaż ich głównym odbiorcą jest parser.

7. Hierarchia klas zdarzeń

```
* = klasa abstrakcyjna

> Event *

  > ParserEvent *     (zdarzenia obsługiwane przez ParserContext)
                 (dotyczą wiadomości wychodzących)
    > OutError   (wysłanie wiadomości o błędzie)
    > OutClose   (zamknięcie połączenia - wykorzystywane przy informacjach)
    > OutPing    (żądanie ping)
    > OutQuery   (informacja query)
    > OutQueryReq     (żądanie query)
    > OutH2H     (żądanie h2h)
    > OutL2H     (żądanie l2h)
    > OutResults (informacja results)
    > OutGet     (żądanie get)
    > OutHubInfo (informacja hub_info)

    > OutReply * (odpowiedzi - analogiczne do występujących w protokole)
      > OutReplyPing
      > OutReplyQuery
      > OutReplyH2H
```

```
    > OutReplyL2H
    > OutReplyNeigh
    > OutReplyGet

> UIEvent *    (zdarzenia obsługiwane przez UIContext)

  > UIQuit     (zakończenie działania UI)
  > UIResults  (odebrano nowe wyniki)
  > UIFileInfo (szczegółowe informacje o lokalnym/ściąganym pliku)
  > UINeighbours    (lista bieżących sąsiadów)
  > UIError    (próba wykonania polecenia od UI zakończona błędem)

> MainEvent *  (zdarzenia obsługiwane przez MainContext)

  > Request *  (polecenia od UI do logiki)
    > ReqQuit  (zakończenie działania aplikacji)
    > ReqConnect    (podłączenie do sieci)
    > ReqDisconnect  (odłączenie od sieci)
    > ReqDirs *     (polecenia dotyczące lokalnego systemu plików)
      > ReqShareAdd  (udostępnienie plików z katalogu)
      > ReqShareRem  (zakończenie udostępniania plików z katalogu)
      > ReqShareList (lista udostępnianych katalogów)
      > ReqDestSet   (ustawienie katalogu do zapisywania ściąganych plików)
      > ReqDestGet   (pobranie katalogu docelowego)
    > ReqQuery (rozpoczęcie wyszukiwania)
    > ReqFileInfo   (uzyskanie informacji o pliku)
    > ReqGetFile    (rozpoczęcie pobierania pliku)
    > ReqTransferAbort    (przerwanie pobierania pliku)
    > ReqHubConnect  (podłączenie do danego huba)
    > ReqHubForce   (zmuszenie węzła do zostania hubem)
    > ReqNeighbours  (uzyskanie listy sąsiadów)

  > NetEvent * (zdarzenia dotyczące połączenia)
               (dotyczą wiadomości przychodzących)
    > NetError *    (błąd w połączeniu)
      > SocketError  (błąd podczas nawiązywania połączenia)
     > ParserError    (błąd podczas parsowania nagłówka XML)
```

```
    > StreamError    (błąd podczas przesyłania danych binarnych)
    > NetPing  (żądanie ping)
    > NetQuery (informacja query)
    > NetQueryReq    (żądanie query)
    > NetH2H   (żądanie h2h)
    > NetL2H   (żądanie l2h)
    > NetResults     (informacja results)
    > NetGet   (żądanie get)
    > NetHubInfo     (informacja hub_info)
    > NetReply *     (odpowiedzi - analogiczne do występujących w protokole)
      > NetReplyPing
      > NetReplyQuery
      > NetReplyH2H
      > NetReplyL2H
      > NetReplyNeigh
      > NetReplyGet

  > ConnEvent *      (dodatkowe zdarzenia)
    > ConnTimer      (nastąpił czas wysłania ping/hub_info do danego węzła)
```

8. Obsługa błędów w transmisji

Aplikacja Shyshark kontroluje transmisję danych. By zapewnić wysoką przepustowość i nie marnotrawić zasobów, niedziałające lub nieaktywne połączenia są zrywane. Realizacja przy pomocy obsługi błędów zwracanych przez sockety BSD, takich jak EPIPE oraz programowej realizacji timeout'ów. W tym drugim przypadku chodzi o to, by nie blokować żadnej ze stron transmisji, jeśli nie są przesyłane dane, mimo istniejącego połączenia TCP, które bez przeszkód jest utrzymywane dzięki niewidocznym w warstwie aplikacji pakietom keepalive.

Po stronie ściągającej dane, kontrola realizowana jest za pomocą funkcji select(). Jeśli w ciągu określonego czasu gniazdo nie doczeka się danych, zgłaszany jest wyjątek StreamFail.

Po stronie wysyłającej wykorzystuje się wywołanie nieblokujące send (z flagą MSG_DONTWAIT). Jeśli zostanie zwrócony błąd EAGAIN, czyli informacja o tym że wywołanie blokujące w tej samej sytuacji zablokowałoby się, aplikacja oczekuje

określony czas, po czym próbuje ponowić transmisję. Zakłada się, że w przypadku braku usterek połączenia oraz aplikacji odbierającej dane, czas ten powinien wystarczyć na opróżnienie buforów sieciowych. Jeśli drugi raz z rzędu zostanie zwrócony błąd EAGAIN, zgłaszany jest wyjątek StreamFail.

Wspomniane czasy aktualnie są rzędu sekund, by umożliwić prezentację ich działania. Są jednak w pełni niesymetryzowane. (SENDIN_TIMEOUT oraz SLEEP_ON_EAGAIN)

Wyjątek StreamFail zgłaszany jest bezwarunkowo w przypadkach błędów innych niż EAGAIN. Obsługa 'broken pipe' jest ograniczona do zwrócenia błędu EPIPE przez funkcje sieciowe - wyłączone jest zgłaszanie sygnału SIGPIPE.

Wyjątek StreamFail jest przechwytywany przez warstwę która aktualnie korzysta z obieku Socket – najczęściej jest to parser - która powoduje wygenerowanie zdarzenia SocketError do logiki i zamknięcie połączenia.

9. Dodatkowe klasy

NodeInfo – identyfikuje węzeł, przede wszystkim zawiera adres IP i numer portu.

File – hierarchia klas dziedziczących po File służy do reprezentacji wszystkich rodzajów plików (ActiveFile – ściągany, NetFile – zdalny, File – lokalny).

FileSet – kolekcja plików, którą używamy do przekazywania informacji o całych zbiorach, tak jak np. w wiadomości Results – gdzie informacje o zdalnych zasobach są trzymane w FileSet jako NetFile.

Chunk + ChunkSet – klasy wykorzystywane do zarządzania ściąganiem plików w kawałkach.