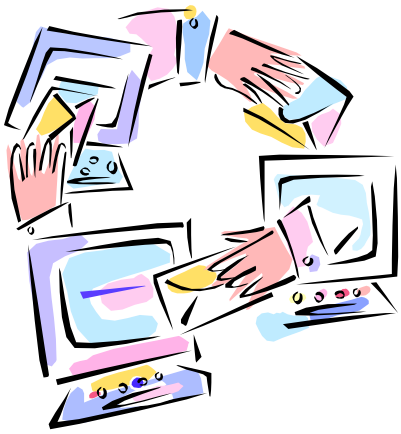
Politechnika Warszawska

Wydział Elektroniki i Technik Informacyjnych



Techniki Internetowe

Biblioteka komunikacyjna - TINYLIB

Prowadzący:

dr inż. Piotr Gawkowski

Autorzy:

Krzysztof Jastrzębski, Wojciech Kaczmar, Michał Krawczak, Tomasz Mrozek

Spis treści

Wstęp

Celem projektu była budowa biblioteki, która za pomocą prostych funkcji umożliwiała komunikację między zdalnymi hostami. Zaletą biblioteki jest jej prostota obsługi - do poprawnego użytkowania nie jest wymagane instalowanie żadnych dodatkowych bibliotek ani oprogramowania oraz bezpieczeństwo - wiadomości będą szyfrowane algorytmem hybrydowym (szyfrowanie asymetryczne do przesyłania klucza oraz szyfrowanie symetryczne do przesyłania wiadomości).
Projekt został zaimplementowany w języku C++ (z elementami C++11).

Opis funkcjonalny

Poniżej przedstawiona jest zrealizowana funkcjonalność systemu:

- Bezpieczne przysyłanie danych pomiędzy użytkownikami, którzy w danej chwili korzystają z biblioteki.
- Klient podaje adres i port serwera, z którym chce się połączyć.
- Serwer może określić, którym klientom (adresom) wolno się z nim połączyć.
- Serwer może określić, którym klientom (adresom) nie wolno się z nim połączyć.
- Dane mogą być odbierane w jednym z dwóch trybów: synchronicznym i asynchronicznym.
    - Tryb synchroniczny powoduje zawieszenie wywołującego wątku do momentu zakończenia operacji.
    - Tryb asynchroniczny nie zawiesza wołającego, za to w momencie zakończenia transmisji (niekoniecznie pomyślnego) wywołuje odpowiednie zdarzenie, które może zostać obsłużone w osobnym wątku.

- W trybie asynchronicznym trwająca operacja może zostać przerwana.
- Testowanie biblioteki będzie umożliwione przez prostą aplikację demonstrującą jej działanie oraz poprzez przykładowe programy testowe, w których będą testowane różnego rodzaju sytuacje krytyczne oraz niespodziewane.
- Biblioteka umożliwia komunikację używającą algorytmu szyfrowania hybrydowego gdzie role algorytmu asymetrycznego pełni zaimplementowany algorytm RSA.

Testy i analiza sytuacji krytycznych

Utrata połączenia – w przypadku utraty połączenia dbamy o zakończenie wszystkich wątków, których dane połączenie dotyczyło.

Scenariusz testowania:

1. .Na jednym z komputerów stawiamy serwer, gotowy do odbierania danych na porcie x.
2. Na drugiej maszynie tworzymy klienta, którego podłączamy do serwera na dany port.
3. Rozpoczynamy wymianę danych.
4. W trakcie tej wymiany kończymy jeden z programów i sprawdzamy czy na żadnym z komputerów nie zostały jakieś śmieci.
5. Powtarzamy wszystkie kroki, zmieniając stronę kończącą połączenie.

3.2 Zbyt duża liczba połączeń do jednego serwera (zapchanie serwera).

Scenariusz testowania:

Ten przypadek pozostawiamy zdrowemu rozsądkowi użytkownika biblioteki, ponieważ nie będzie możliwości podłączenia się większej ilości klientów niż jest zadeklarowanych portów do nasłuchiwania. W razie jakby jeden lub więcej klientów chciał uniemożliwić prace serwera ciągłym podłączaniem się, serwer ma do dyspozycji czarną oraz białą listę użytkowników wrogich lub zaprzyjaźnionych.

3.3 Niepowodzenie przy uwierzytelnianiu klienta

Scenariusz testowania:

1. Na jednym z komputerów stawiamy serwer, gotowy do odbierania danych na porcie x.
2. . Na drugiej maszynie tworzymy klienta.
3. Serwer definiuje listę dozwolonych adresów IP, z którymi może się połączyć.
4. Serwer sprawdza adres na liście dozwolonych adresów (whitelist)
5. Serwer odpowiada “ACCEPT”, (jeśli klient może się z nim połączyć) lub “DENY”

3.4 Klient po nawiązaniu połączenia z serwerem zrywa połączenia

Opis testowania systemu

Testować będziemy oczywiście podstawowe funkcjonalności biblioteki

- nawiązanie połączenia

- wysyłanie i odbieranie plików (synchroniczne i asynchroniczne)

- poprawność szyfrowania i deszyfrowania danych

- poprawność odebranych danych

Koniecznością jest sprawdzenie czy biblioteka błędnie nie zezwoli na połączenie klientowi, którego nie ma na liście akceptowalnych maszyn.

Badanie przeciążenia serwera.

Testy muszą sprawdzić radzenie sobie programu z obsługą sytuacji krytycznych, np. utraty połączenia, oraz z sytuacjami narażonymi na potencjalne błędy, np. wysyłanie bardzo dużego pliku.

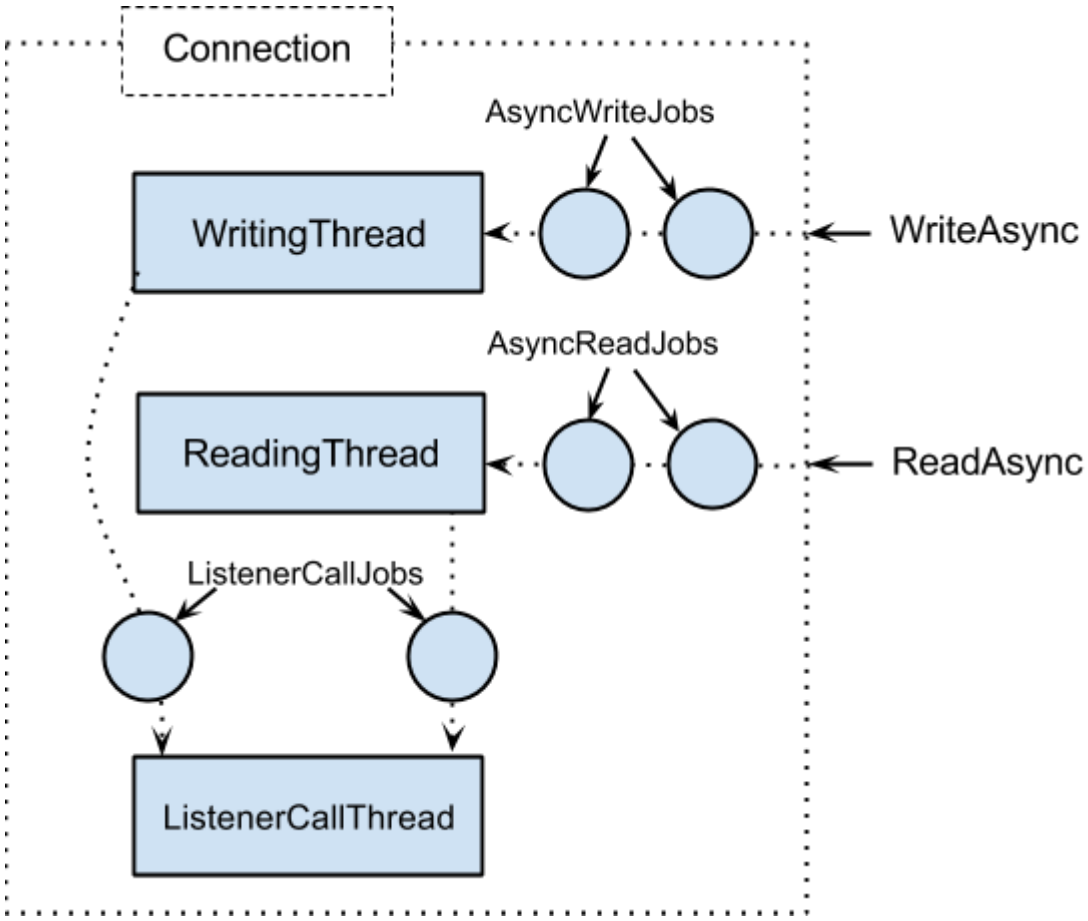
Realizacja operacji asynchronicznych

W projekcie wykorzystamy wątki z biblioteki standardowej c++11 (przeniesione z biblioteki boost::thread). Biblioteka standardowa c++11 dostarcza też odpowiednich mechanizmów do ich synchronizacji (my wykorzystamy semafory).

Schemat komunikacji między wątkami, nieuwzględniający możliwości przerwania zapisu / odczytu asynchronicznego przez aplikację kliencką (o tym dalej).

Z każdym połączeniem związane są trzy wątki, tworzone w momencie nawiązywania połączenia i likwidowane po jego przerwaniu:

- WritingThread - wykonuje operacje nadawania asynchronicznego
- ReadingThread - wykonuje operacje odbierania asynchronicznego
- ListenerCallThread - wykonuje dostarczone przez użytkownika funkcje “nasłuchujące”. Istnienie takiego wątku, co prawda nie jest konieczne, gdyż funkcje te mogłyby być wykonywane przez wątek piszący lub czytający. Jednak istnienie ListenerCallThread zapewnia większą elastyczność - dzięki temu odbieranie i nadawanie danych będzie działać dalej, nawet, jeśli listener z poprzedniej operacji będzie wykonywał jakąś dłuższą operację (np. złożone obliczeniowo przetwarzanie odebranych właśnie danych).



Działanie wątku bibliotecznego

Aby zapewnić prawidłową synchronizację operacji wysyłania (odpowiednio: odbierania), wątki dla operacji asynchronicznych będą korzystać z kolekcji działającej jak BlockingQueue znanej z Javy:

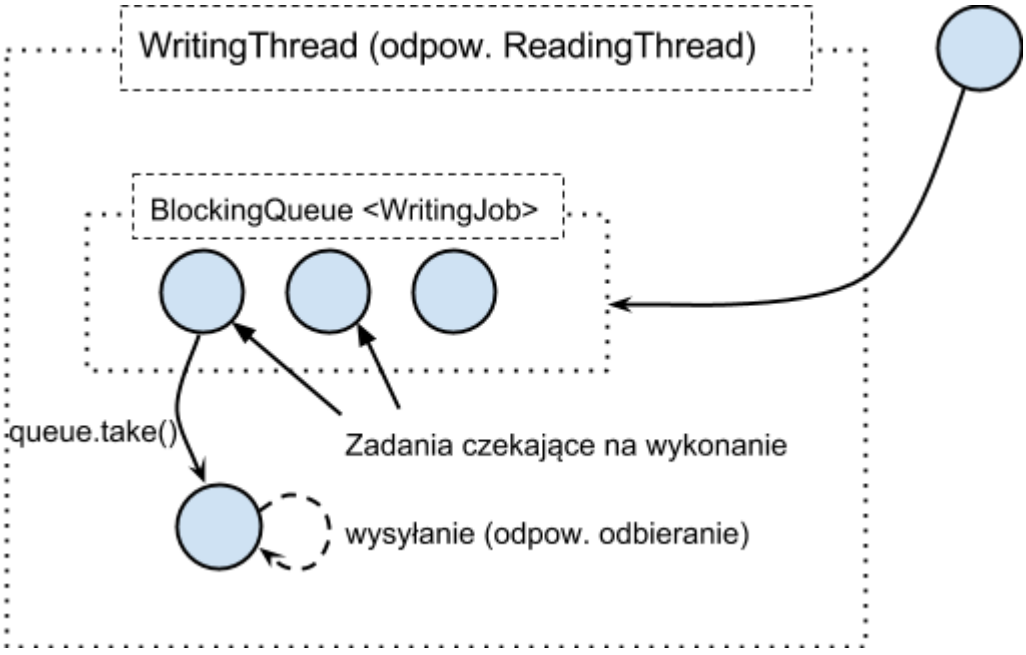
- Zlecenie nowego zadania jest realizowane, jako dodanie obiektu WritingJob (odpowiednio ReadingJob) do kolejki. Obiekt taki reprezentuje jedną operację, (czyli pojedyncze wywołanie AsyncRead/AsyncWrite). Jest on powiązany z obiektem AsyncIdentifier zwróconym do użytkownika w relacji "jeden do jednego". Wywołanie funkcji stop() na obiekcie AsyncIdentifier powoduje unieważnienie zadanej operacji (szczegóły niżej).

- Wątek w nieskończonej pętli pobiera zadania z kolejki (metoda take() kolejki). Jeżeli w kolejce znajduje się jakieś zadanie - kolejka po prostu je zwraca (jak std::queue). Jeżeli zaś w kolejce nie ma żadnych zadań, wtedy wątek zostaje uśpiony.

- Uśpiony wątek zostaje obudzony w momencie, gdy do pustej kolejki zostaje dodane zadanie.

- Usypianie wątków zostanie zrealizowane w wewnętrznej strukturze klasy BlockingQueue - za pomocą semaforów.

- Ponadto BlockingQueue zapewnia sekcje krytyczne (również za pomocą semafora), co chroni przed hazardami, które mogłyby wystąpić na granicy wątków.



Schemat działania wątku, nieuwzględniający możliwości przerwania operacji z zewnątrz

W ten sposób zapewniamy elegancką (BlockingQueue ma interfejs podobny do queue) i wydajną (brak ryzyka aktywnego oczekiwania) obsługę wątków.

Analogicznie działa ListenerCallThread - również posiada kolejkę zadań (listenerów) do wykonania, przy czym do niej zdarzenia są dodawane przez dwa pozostałe wątki: piszący i czytający.

Dzięki takiej architekturze zapewniamy, że zlecenia od użytkownika są zawsze wykonywane w takiej kolejności, w jakiej zostały zlecone (w ramach swojego "typu"), a także, że listenery zdarzeń są wykonywane w takiej kolejności, w jakiej zaszły odpowiadające im zdarzenia.

Szyfrowanie

W bibliotece do bezpiecznego przesyłania danych zastosowaliśmy szyfrowanie hybrydowe - polegające na tym, że nadawca najpierw wysyła klucz do szyfrowania symetrycznego. Klucz jest zaszyfrowany kluczem publicznym odbiorcy algorytmu asymetrycznego (RSA). Dalsza komunikacja odbywa się poprzez kodowanie algorytmem symetrycznym - ze względów wydajnościowych (algorytmy symetryczne są z reguły szybsze od algorytmów asymetrycznych).

Obsługa biblioteki

Aby nawiązać połączenie i w konsekwencji przesłać dane, musi zostać spełnione kilka warunków. Należy stworzyć dwie instancje klasy Connection. Jeden obiekt będzie serwerem (klasa Server dziedzicząca po klasie ServerConnection) a drugi klientem (klasa ClientConnection) Podczas tworzenia obiektu serwer należy podać port, na którym, będzie łączył się z klientem. W konstruktorze serwera ustawi się również tryb działania listy adresów IP. Domyślny tryb akceptuje wszystkie adresy, oprócz tych znajdujących się na czarnej liście. Po stworzeniu obiektu ClientConnection należy dla tego obiektu wywołać metodę connect(), która przyjmuje dwa argumenty: adres IP serwera (w postaci stringu) oraz port serwera(4-cyfrowy int). W przypadku pomyślnego połączenia z serwerem (nie zostanie rzucony żaden wyjątek) obiekt klienta bądź serwera może wywołać metody czytania i pisania w

dwóch trybach: asynchronicznym i synchronicznym za pomocą metod readSync()/writeSync() lub readAsync()/writeAsync(). Klasa serwer udostępnia również prostą metodę listenForOneCllient()->readSync() w której serwer odczytuje dane tylko od jednego klienta. Metoda ta przyjmuje dwa argumenty: tablice do której będą odczytywane dane oraz ilość danych do przeczytania.