Bezpieczeństwo usług sieciowych

Laboratorium 1: Komunikator z szyfrowaniem

Szymon Bagiński

4 listopada 2018

1 Cel zadania

Celem zadania było przygotowanie komunikatora (czatu) klient-serwer wspierającego bezpieczną wymianę sekretu (przy użyciu protokołu Diffiego-Hellmana). Komunikator musiał wspierać szyfrowanie wiadomości zgodnie z następującym formatem danych opartym o JSON:

Stage	A (client)	B (server)
1	{ “request”: “keys” } ->	
2		<- { “p”: 123, “g”: 123 }
3	{ “a”: 123 } ->	<- { “b”: 123 }
4	{ “encryption”: “none” } ->	
5	{ “msg”: “...”, “from”: “John” } ->	<- { “msg”: “...”, “from”: “Anna” }

1. Po połączeniu do serwera klient prosi o liczby p oraz g.
2. Serwer wysyła do klienta liczby p oraz g.
3. Serwer i klient wymieniają się publicznymi wartościami A oraz B:
   (a) Klient wysyła do serwera obliczoną wartość A.
   (b) Serwer wysyła do klient obliczoną wartość B.

   Kroki a oraz b mogą nastąpić w dowolnej kolejności.
4. *OPCJONALNIE* Klient wysyła do serwera informację o żądanym sposobie szyfrowania wiadomości. Jeżeli klient nie wyśle tej informacji, to strony przyjmują domyślne szyfrowanie ustawione na “none”.
5. Klient oraz serwer wymieniają się szyfrowanymi wiadomościami.

Treść wiadomości powinna być zakodowana za pomocą base64 przed umieszczeniem jej w strukturze JSON.

Wspierane metody szyfrowania:

- none - brak szyfrowania (domyślne)
- xor - szyfrowanie xor jedno-bajtowe (należy użyć najmłodszego bajtu sekretu)
- cezar - szyfr cezara

2 Wykonanie zadania

2.1 Technologia

Do wykonania zadania postanowiono skorzystać z języka programowania **Rust**. Jest to stosunkowo nowy język ukierunkowany na bezpieczeństwo oraz wysoką wydajność tworzonych programów. Zadanie potraktowano jako pewnego rodzaju eksperyment z tym językiem, żeby się z nim zapoznać.

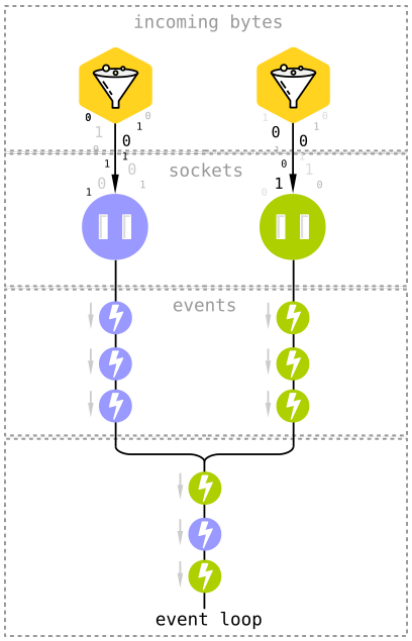
Aby działać poprawnie, komunikator musi wysyłać i odbierać dane przez gniazda sieciowe. Może się wydawać, że jest to proste zadanie, ale istnieje kilka sposobów o różnej złożoności, efektywnej obsługi operacji wejścia i wyjścia. Główna różnica między podejściami polega na traktowaniu blokowania: domyślnie zapobiega się wszelkim operacjom CPU, podczas gdy czekamy, aż dane dotrą do gniazda sieciowego. Ponieważ nie możemy pozwolić jednemu użytkownikowi naszej usługi czatu blokować innych, musimy jakoś je wyizolować.

Najczęstszym sposobem jest utworzenie oddzielnego wątku dla każdego użytkownika, tak aby blokowanie zadziałało tylko w kontekście pojedynczego wątku. Ale chociaż koncepcja jest prosta i taki kod jest łatwy do napisania, każdy wątek wymaga pamięci dla swojego stosu i ma narzut przełączników kontekstu - nowoczesne procesory serwera mają zwykle około 8 lub 16 rdzeni, a pojawianie się wielu wątków wymaga od jądra systemu operacyjnego ciężkiej pracy, aby przełączać między nimi wykonanie z odpowiednią prędkością. Z tego powodu trudno jest skalować wielowątkowość do wielu połączeń.

Zamiast tego użyjemy wydajnych interfejsów systemu, które wykorzystują pętlę zdarzeń (event loop). Działają one w prosty sposób: bajty przychodzą przez sieć, docierają do gniazd, a zamiast czekać, kiedy dane staną się dostępne dla odczytu, mówimy gniazdu, aby nas powiadomić o nadejściu nowych danych.

Powiadomienia przychodzą w formie zdarzeń, które są obsługiwane przez pętle zdarzeń [Rysunek 1]. I tu właśnie dzieje się blokowanie: zamiast okresowych kontroli tysięcy gniazd, czekamy tylko na pojawienie się nowych zdarzeń. To ważne różnica, ponieważ szczególnie w aplikacjach sieciowych bardzo często zdarza się, że wielu bezczynnych klientów czeka na jakąś aktywność. Dzięki asynchronicznym operacjom wejścia / wyjścia będziemy mieli bardzo niewiele narzutów na uchwytach gniazda.

Aby zaimplementować opisane podejście skorzystano z biblioteki mio (Metal IO), która kładzie nacisk na jak najmniejsze obciążenie.



Rysunek 1: Schemat modelu opartego o zdarzenia

2.2 Generowanie parametrów protokołu Diffiego-Hellmana

W programie server zaimplementowano dynamiczne generowanie parametrów wysyłanych do każdego klienta, na podstawie których potem oblicza się numery publiczne oraz sekret.

W pod module serwera `parameters.rs` znajduje się funkcja `generate_parameters`, która tworzy nowy zestaw parametrów. W tym celu najpierw wybierana jest losowa liczba pierwsza p. W tym przypadku jest to liczba pierwsza z zakresu $<9001, 10007>$. Drugi parametr g jest wybierany losowo spośród wszystkich pierwiastków pierwotnych modulo p. Z uwagi na fakt, że znajdowanie pierwiastków pierwotnych jest stosunkowo czasochłonną operacją zdecydowano się na niewielkie liczby. Implementacja rozwiązania pozwala jednak na szybką zmianę możliwych wartości parametrów.

2.3 Ważne funkcjonalności systemu

- Serwer wspiera wielu klientów
- Kolejność wymiany wartości A i B jest dowolna
- Parametry protokołu Diffiego-Hellmana są generowane dynamicznie
- Parametry protokołu Diffiego-Hellmana są różne dla każdego klienta
- Z punktu widzenia serwera wiadomość o szyfrowaniu jest opcjonalna, natomiast klient zawsze ją wysyła nawet gdy nie jest ona ustawiona

2.4 Korzystanie

Jako, że było to zadanie czysto edukacyjne, programy **client** i **server** są skonfigurowane odgórnie, aby komunikować się za pomocą konkretnego portu na localhost-cie.

Aby korzystać z komunikatora należy najpierw uruchomić program **server**, a następnie dowolną ilość programów **client**, które posiadają tylko interfejs konsolowy (sposób uruchomienia został opisany w pliku README). Każda wpisana linia jest interpretowana jako nowa wiadomość i rozsyłana do pozostałych użytkowników. Równolegle są pokazywane wiadomości od innych użytkowników połączonych z tym samym serwerem.

3 Podsumowanie

Wykonanie zadania zajęło dużo więcej czasu niż założono na początku. Spowodowane było to przede wszystkim nieznajomością technologii oraz wyborem bibliotek. Nie mają one w znacznej większości stabilnej wersji, a ich interfejs się znacząca różni między wersjami. Jest to zrozumiałe, ponieważ otoczenie języka **Rust** jest relatywnie młode i się dynamicznie rozwija. Materiały edukacyjne i różnego rodzaju tutoriale nie są w sieci mocno rozpowszechnione w porównaniu do innych technologii, a dokumentacja często pozostawia wiele do życzenia.

Niemniej jednak założenia zadania udało się wypełnić w zdecydowanej większości, przy okazji zdobywając nową wiedzę i doświadczenia z **Rust**-em. Specyficzna składnia tego języka oraz pewien rygor programowania w nim pozwala uniknąć wielu błędów podczas programowania, co w przypadku komunikacji sieciowej jest istotne. Język ten staje się także po pewnym czasie, gdy zrozumie się podstawowe koncepty takie jak zarządzanie pamięcią, wygodny i intuicyjny.