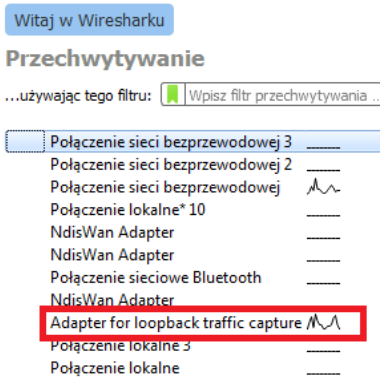
Systemy rozproszone | Technologie middleware, cz. I

Łukasz Czekierda, Katedra Informatyki AGH (luke@agh.edu.pl)

1. Przygotowanie do zajęć i weryfikacja środowiska

Co będzie potrzebne:

- java
- IDE: Eclipse, IntelliJ
- Wireshark z możliwością przechwytywania pakietów przechodzących przez interfejs loopback (windows: potrzebna biblioteka npcap, instalowana automatycznie w czasie instalacji Wiresharka (starsze wersje Wireshark instalowały winpcap)), alternatywnie: dwa komputery z możliwością wzajemnej komunikacji
- Zeroc ICE (wersja 3.7): https://zeroc.com/
- Apache Thrift: (wersja 0.13.0): https://thrift.apache.org/



Weryfikacja czy wszystko jest gotowe na zajęcia:

- console# slice2java
- console# thrift
- wireshark: dla opcji z loopback: rysunek z prawej

2. Wykonanie ćwiczenia

2.1 Wprowadzenie

W miarę możliwości technicznych Prowadzący wprowadzi Studentów w temat ćwiczenia sprawdzając równocześnie ich przygotowanie. W razie braku takich możliwości, Student sam zapoznaje się z udostępnioną prezentacją.

2.2 Zeroc ICE

Na platformie Moodle znajdują się kody źródłowe aplikacji klient-serwer napisanej w języku Java, a struktura projektu jest akceptowana przez IDE Eclipse i IntelliJ. Wykonaj poniższe kroki zastanawiając się nad podanymi pytaniami. Cenne jest także nieznaczne samodzielne rozszerzanie zakresu ćwiczenia (w miarę wolnego czasu).

1. Zaimportuj projekt do IDE.
2. Skompiluj plik z definicją interfejsu: otwórz okno konsolowe (MS-DOS) i z poziomu głównego katalogu projektu wydaj polecenie **slice2java --output-dir *generated slice\calculator.ice***. Jak skompilować plik dla aplikacji w języku Python? A C++?
3. Jeśli IDE nie realizuje automatycznego odświeżania w razie zmian zawartości projektu na dysku, należy wymusić odświeżenie. Występujące wcześniej błędy kompilacji powinny zniknąć. W przypadku korzystania z **IntelliJ** po kompilacji interfejsu należy oznaczyć folder **generated** jako **Generated Sources Root**.
4. **Analiza kodu źródłowego.** Zapoznaj się ze strukturą projektu. Co znajduje się w katalogu **generated**?
5. **Analiza kodu źródłowego.** Przejrzyj kod źródłowy zawarty w katalogu **src** w następującej kolejności: 1. klasa serwanta, 2. klasa serwera, 3. klasa klienta.
6. Uruchom serwer. Jakiego gniazda używa? Jakim poleceniem systemowym można sprawdzić jakie gniazda są powiązane z danym procesem?
7. Uruchom klienta. Korzystając z interaktywnego interfejsu użytkownika (menu) wywołaj operacje **add** oraz **subtract**. Uzupełnij implementację.
8. Odpowiedz na poniższe pytania:
   - Jak nazywa się zmienna serwanta obiektu implementującego kalkulator?
   - Ile sztuk obiektów obecnie udostępnia klientom serwer?
   - Jak nazywa się obiekt udostępniany zdalnie (klientowi)?
   - W jaki sposób klient uzyskał referencję do tego konkretnego obiektu (tj. co zawiera referencja)?

- Co się stanie jeśli klient zrealizuje wywołanie na nieistniejącym obiekcie (przetestuj)?

9. **Strategia: wiele obiektów, wspólny serwant**. Skojarz nowy obiekt o nazwie **calc102**, z dotychczasowym serwantem. Przetestuj działanie aplikacji komunikacji z oboma obiektami naraz. Sprawdź czy serwant może wiedzieć, na rzecz jakiego obiektu realizuje wywołanie (podpowiedź: **__current.id**);
10. **Strategia: wiele obiektów, każdy z dedykowanym serwantem**. Stwórz nowy (dodatkowy) obiekt serwanta i skojarz z nim obiekt o nazwie **calc102**. Przetestuj działanie aplikacji w komunikacji z oboma obiektami naraz.
11. **Analiza komunikacji sieciowej**. Przetestuj działanie komunikacji z użyciem wireshark (warto dodać odpowiedni filtr, np. **tcp.port==10000**). Jakie informacje zawiera żądanie, a jakie odpowiedź? Co trzeba zmienić w kodzie/konfiguracji klienta i/lub serwera, by klient mógł się komunikować z serwerem działającym na innej maszynie? Zastanów się dwa razy zanim udzielisz (niepoprawnej) odpowiedzi…
12. **Sekwencje**. Rozbuduj interfejs (slice) o nową operację wyliczającą średnią z sekwencji N podanych liczb typu **int**. Użyj właściwych typów do reprezentacji sekwencji i wartości zwracanej. Zadbaj również o elegancką obsługę sytuacji, w której długość sekwencji wynosi zero (wyjątek). W razie potrzeby sięgnij do dokumentacji Ice (https://doc.zeroc.com/ice/3.7/the-slice-language). Skompiluj plik, zaimplementuj nową operację, rozbuduj klienta i przetestuj działanie aplikacji.
13. **Wywołanie synchroniczne – symulacja dużego opóźnienia**. Zrealizuj takie wywołanie operacji **add** w kliencie (dodaj kolejną pozycję w menu), by wywołanie metody serwanta (po stronie serwera) trwało długo (zobacz na jej implementację) i przetestuj komunikację.
14. **Wywołania asynchroniczne**. Prześledź istniejące wzorcowe wywołania **add-asyn1** oraz **add-asyn2-req** i **add-asyn2-res**. W jakich przypadkach warto w taki sposób realizować wywołania zdalne?
15. **Pula wątków adaptera**. Domyślna wielkość puli wątków adaptera wynosi 1 (spróbuj przetestować, np. uruchamiając kolejną instancję klienta i wywołując długotrwałą operację) – kiedy to może stanowić problem? Bardziej zaawansowane aspekty działania aplikacji Ice są definiowane w pliku konfiguracyjnym, dla serwera zazwyczaj ma nazwę **config.server**. Jaką wielkość puli wątków tam skonfigurowano? Użyj ten plik do konfiguracji serwera (dotąd cała konfiguracja była w kodzie źródłowym) startując serwer z argumentem **--Ice.Config=config.server**. Zauważ różnicę w działaniu serwera w aspekcie puli wątków adaptera.

2.3 Apache Thrift

Na platformie Moodle znajdują się kody źródłowe aplikacji klient-serwer napisanej w języku Java, a struktura projektu jest akceptowana przez IDE Eclipse i IntelliJ. Wykonaj poniższe kroki zastanawiając się nad podanymi pytaniami. Cenne jest także nieznaczne samodzielne rozszerzanie zakresu ćwiczenia (w miarę wolnego czasu).

1. Zaimportuj projekt do IDE.
2. Skompiluj plik z definicją interfejsu: otwórz okno konsolowe (MS-DOS) i z poziomu głównego katalogu projektu wydaj polecenie **thrift --gen java calculator.thrift**. Jak skompilować ten plik dla aplikacji w języku Python?
3. Jeśli IDE nie realizuje automatycznego odświeżania w razie zmian zawartości projektu na dysku, należy wymusić odświeżenie. Występujące wcześniej błędy kompilacji powinny zniknąć.
4. **Analiza kodu źródłowego.** Zapoznaj się ze strukturą projektu i przejrzyj kod źródłowy, w tym wygenerowane pliki źródłowe.
5. Uruchom klienta i serwer oraz przetestuj poprawność działania aplikacji na jednej maszynie. Wywołuj różne operacje w różnych konfiguracjach serwera (komentując/odkomentowując poszczególne fragmenty jego kodu źródłowego) dla osiągnięcia pełnego zrozumienia działania aplikacji.
6. **Rozbudowa interfejsu.** Rozbuduj interfejs o nową, nie bardzo trywialną operację, wykorzystując dostępne typy języka IDL. Zaimplementuj ją i przetestuj działanie aplikacji. W razie potrzeby posłuż się dokumentacją: https://thrift.apache.org/docs/.
7. **Analiza komunikacji sieciowej.** Prześledź komunikację pomiędzy klientem i serwerem używając wireshark z odpowiednim filtrem. Ile bajtów (na poziomie warstwy czwartej) ma pojedyncze wywołanie? Którego protokołu transportowego używa?
8. **Zmiana serializacji wiadomości w komunikacji sieciowej.** Zmień w kodzie klienta i serwera sposób serializacji, tak, by była wykorzystywana notacja JSON. Prześledź komunikację korzystając z wireshark.
9. **Podejście obiektowe czy usługowe?** Zaobserwuj (testując), w jaki sposób wykorzystując **TBinaryProtocol** jest możliwe udostępnienie dla zdalnych wywołań kilku obiektów (implementujących ten sam lub różne interfejsy IDL) naraz.
10. **Podejście obiektowe czy usługowe?** Zaobserwuj (testując), w jaki sposób wykorzystując **TMultiplexedProcessor** jest możliwe udostępnienie dla zdalnych wywołań kilku obiektów (implementujących ten sam lub różne interfejsy IDL) naraz.