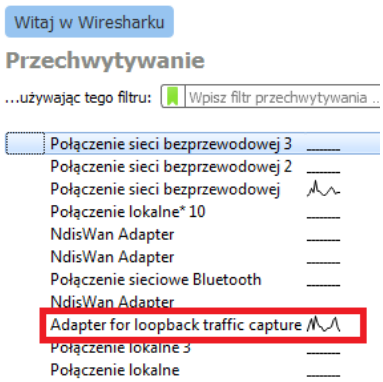
Systemy rozproszone | Technologie middleware, cz. I

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

1. Przygotowanie do zajęć i weryfikacja środowiska

Co będzie potrzebne:

- java
- IDE: Eclipse lub 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 3.7.5): https://zeroc.com/
- Apache Thrift: (wersja 0.14.1): 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

Prowadzący wprowadzi Studentów w temat ćwiczenia sprawdzając równocześnie ich przygotowanie do zajęć.

2.2 Zeroc ICE

W platformie Moodle znajdują się kody źródłowe aplikacji klient-serwer napisanej w języku Java. 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. Zapoznaj się z zawartością pliku **slice/calculator.ice**.
3. 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 tworzonych w języku Python? A C++? Sprawdź.
4. Jeśli IDE nie realizuje automatycznego odświeżania w razie zmian zawartości projektu na dysku, należy wymusić to odświeżenie. Występujące wcześniej błędy kompilacji powinny zniknąć. W przypadku korzystania z **IntelliJ** po kompilacji interfejsu należy oznaczyć folder **generated** jako **Generated Sources Root**.
5. **Analiza kodu źródłowego.** Zapoznaj się ze strukturą projektu. Co znajduje się w poszczególnych katalogach, w tym w katalogu **generated**?
6. **Analiza kodu źródłowego.** Przejrzyj kod źródłowy zawarty w katalogu **src** w następującej kolejności: 1. klasa serwanta, 2. klasa serwera, 3. klasa klienta.
7. Uruchom serwer. Jakiego/jakich gniazd(a) używa? Jakim poleceniem systemowym można sprawdzić jakie gniazda są powiązane z danym procesem? Sprawdź.
8. Uruchom klienta. Korzystając z interaktywnego interfejsu użytkownika (menu) wywołaj operacje **add** oraz **subtract**. Uzupełnij implementację.
9. Odpowiedz na poniższe pytania:
   - Jak nazywają się zmienne serwanta obiektu implementującego kalkulator?
   - Ile sztuk obiektów obecnie udostępnia klientom serwer?
   - Jak nazywają się te obiekty?
   - Jak nazywa się obiekt, z którym komunikował się klient?
   - 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)?

10. **Strategia: wiele obiektów, wspólny serwant**. W kodzie serwera skojarz nowy obiekt o nazwie **calc33** i kategorii **calc** z dotychczasowym serwantem. Przetestuj działanie komunikacji z oboma obiektami naraz (np. w kodzie klienta utwórz drugą zmienną będącą referencją do drugiego zdalnego obiektu). Sprawdź czy serwant może wiedzieć, na rzecz jakiego obiektu realizuje konkretne wywołanie (podpowiedź: **__current.id**);
11. **Strategia: wiele obiektów, każdy z dedykowanym serwantem**. Stwórz nowy (dodatkowy) obiekt serwanta i skojarz z nim obiekt o nazwie **calc33** i kategorii **calc**. Przetestuj działanie aplikacji w komunikacji z oboma obiektami naraz.
12. **Analiza komunikacji sieciowej**. Przetestuj działanie komunikacji z użyciem wireshark (warto dodać odpowiedni filtr, np. **ip.addr==127.0.0.2**). Jakie informacje zawiera żądanie, a jakie odpowiedź? Co trzeba zmienić w kodzie/konfiguracji klienta i/lub serwera, by mogła zachodzić komunikacja w scenariuszu, w którym klient i serwer są na różnych maszynach? Zastanów się dwa razy zanim udzielisz (niepoprawnej) odpowiedzi…
13. **Sekwencje**. Rozbuduj interfejs (***slice***) o nową operację **avg** wyliczającą średnią z sekwencji N podanych liczb typu **long**. Użyj właściwych typów do reprezentacji sekwencji i wartości zwracanej. Zadbaj również o elegancką obsługę sytuacji, w której długość sekwencji wynosi zero (tj. zadeklaruj i obsłuż wyjątek). W razie potrzeby sięgnij do dokumentacji Ice (https://doc.zeroc.com/ice/3.7/the-slice-language). Skompiluj plik **.slice**, zaimplementuj nową operację w klasie serwanta, rozbuduj klienta o dodatkowe wywołanie i przetestuj działanie aplikacji.
14. **Operacje idempotentne**. Ice pozwala zadeklarować operację jako idempotentną (w interfejsie slice). Co to daje? Jakie operacje mogą być tak oznaczone? Które z operacji interfejsu **Calc** mogą być tak oznaczone?

Komunikacja sieciowa w porównaniu do komunikacji w ramach jednego procesu charakteryzuje się znacznie większym opóźnieniem. Do wzrostu opóźnienia może się także przyczynić niewłaściwie zaimplementowany lub wyskalowany serwer. Warto dołożyć starań, by osiągnąć jak najlepszą efektywność i ergonomię aplikacji rozproszonej.

15. **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 aktualną implementację) i przetestuj komunikację.
16. **Wywołania asynchroniczne**. Prześledź i przetestuj istniejące wzorcowe wywołania **add-asyn1** oraz **add-asyn2-req** i **add-asyn2-res**. Odpowiedz na poniższe pytania:
    - Co składa się na opóźnienie tak realizowanego zlecenia wywołania asynchronicznego z perspektywy klienta (tj. dokąd trafia to wywołanie) i skąd jest odbierany rezultat?
    - W jakich przypadkach warto w taki sposób realizować wywołania zdalne?
    - Czy każde zdalne wywołanie powinno być realizowane asynchronicznie?
17. **Pula wątków adaptera**. Domyślna wielkość puli wątków adaptera w Ice wynosi **1** (spróbuj przetestować, np. wywołując asynchronicznie więcej niż jedno wywołanie naraz (np. „**op-asyn1b 100**”) lub uruchamiając kolejną instancję klienta i wywołując długotrwałą operację), co oczywiście w systemie produkcyjnym nie jest zazwyczaj akceptowalne. Bardziej zaawansowane aspekty działania aplikacji Ice są definiowane w pliku konfiguracyjnym, dla serwera zazwyczaj ma nazwę **config.server**. Jaką wielkość puli wątków jest tam ustawiona? Użyj ten plik do konfiguracji serwera (dotąd cała konfiguracja była w kodzie źródłowym) zmieniając w kodzie serwera sposób inicjalizacji adaptera **ObjectAdapter** i startując serwer z argumentem linii poleceń **--Ice.Config=config.server**. Zauważ różnicę w działaniu serwera w aspekcie puli wątków adaptera i w działaniu klienta (czas oczekiwania).

W wielu scenariuszach minimalizacja ilości danych w komunikacji rozproszonej jest krytyczna. Ice przewiduje kilka mechanizmów mogących ją poprawić: są to m.in. kompresja wiadomości, wywołania oneway, wywołania datagramowe i agregacja żądań.

18. **Efektywność komunikacji – wywołania oneway**. Na czym polega wywołanie **oneway**? Jakie są wymogi dla takiej realizacji? Które z operacji interfejsu **Calculator** mogą być tak zrealizowane? Przetestuj: wywołaj **add** i **op** zmieniając wcześniej tryb pracy *proxy* na ***oneway*** (*dolna część instrukcji w menu klienta*). Na czym polega różnica w komunikacji sieciowej (dialog klienta z serwerem) w stosunku do tradycyjnego wywołania (tj. **twoway**)?
19. **Efektywność komunikacji – wywołania datagramowe**. Na czym polega wywołanie **datagram**? Jakie są wymogi dla takiej realizacji? Które z operacji interfejsu **Calculator** mogą być tak zrealizowane? Przetestuj: wywołaj **add** i **op** zmieniając wcześniej tryb pracy *proxy* na ***datagram***. Na czym polega różnica w komunikacji sieciowej (dialog klienta z serwerem) w stosunku do tradycyjnego wywołania (tj. **twoway**)?
20. **Efektywność komunikacji – agregacja wywołań**. Na czym polega korzyść z agregacji wywołań? Jakie są wymogi dla takiej realizacji? Które z operacji interfejsu **Calculator** mogą być tak zrealizowane? Przetestuj: wywołaj

kilku(nasto)krotnie **add** i **op** zmieniając wcześniej tryb pracy *proxy*. Pamiętaj o „przepchaniu" żądań (***flush***). Na czym polega różnica w komunikacji sieciowej (dialog klienta z serwerem) w stosunku do tradycyjnego wywołania (tj. **twoway**)?

21. **Efektywność komunikacji – kompresja wiadomości**. Kompresja wiadomości jest wykonywana przez zewnętrzne biblioteki – do projektu zostały pliki ***jar*** odpowiedzialne za jej przeprowadzanie (**bzip2-1.0** i **commons-compress-1.20** znajdujące się w katalogu **lib**). W konfiguracji adaptera serwera i *proxy* klienta została dodana opcja **-z** aktywująca ją. Przetestuj komunikację pod tym kątem realizując wywołania operacji **op** oraz **op2** – użyj Wireshark. Drugie z nich powinno być skompresowane – dlaczego tylko drugie? A co z odpowiedziami serwera? Kiedy aktywacja kompresji jest pożądana? Kompresję możesz wyłączyć wołając z menu klienta **compress off** (jeśli dezaktywacja nie działa, usuń opcję **-z** z konfiguracji proxy dla danego protokołu).
22. **Efektywność komunikacji – czas podtrzymywania połączenia TCP**. Kiedy kończyć ustanowione na potrzeby wywołania połączenie TCP? Szybko – kolejne wywołanie będzie musiało ustanowić nowe (opóźnienie, wolny start…). Późno – zużywamy zasoby, zamiast je zwolnić. Za czas utrzymywania połączenia TCP odpowiada m.in. zmienna **Ice.ACM.Timeout** (czas w sekundach). Odkomentuj w pliku konfiguracyjnym klienta (tj. **config.client**) linię zawierającą konfigurację **Ice.Trace.Network=2** i zaobserwuj zdarzenia otwierania i zamykania połączeń TCP (konieczne jest uruchomienie klienta z opcją **--Ice.Config=config.client**). Więcej tu: https://doc.zeroc.com/ice/3.7/property-reference/ice-acm
23. **Efektywność komunikacji – analiza ruchu sieciowego.** Plik **ice.pcapng** zawiera zapis przykładowej komunikacji. Prześledź interesujące Cię aspekty komunikacji. Ciekawsze rzeczy to:
    - ustanowienie połączenia Ice i wywołanie operacji (#4-#12);
    - wiadomość skompresowana (#14);
    - wiadomości w trybie **batched** dla TCP (#33) i UDP (#35);
    - wiadomości **oneway** dla UDP i TCP (#54 i #108);
    - analiza opóźnień wywołań realizowanych w trybie **oneway** (#54-#63, #104-#123) i **synchronicznie** (#64-#103) – serwant wprowadza opóźnienie 500 ms.

2.3 Apache Thrift

W 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 **simple** (komentując/odkomentowując poszczególne fragmenty jego kodu źródłowego) dla osiągnięcia zrozumienia działania aplikacji.
6. **Rozbudowa interfejsu.** Rozbuduj interfejs o nową, nie bardzo trywialną operację (niekoniecznie arytmetyczną), wykorzystując dostępne typy języka IDL. Zaimplementuj ją i przetestuj działanie aplikacji wywołując nową operację. 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 pola danych warstwy czwartej) ma pojedyncze wywołanie? Którego protokołu transportowego (L4) używa Thrift?
8. **Zmiana serializacji wiadomości w komunikacji sieciowej.** Zmień w kodzie klienta i serwera sposób serializacji na pozostałe dostępne i zanotuj ile bajtów ma przykładowe wywołanie w każdej z nich. Do pozyskania tych informacji użyj Wireshark. Porównaj z zapisem komunikacji zawartym w pliku **thrift.pcapng** – znajdź te wywołania (były zrealizowane wywoływane w kolejności **binary-compact-json**).
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.