



---

**CIĄG DALSZY NASTĄPIŁ...**

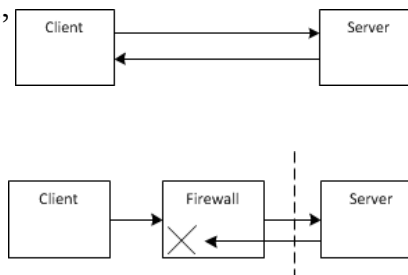


---

**CO NIECO O KOMUNIKACJI  
W INTERNECIE...**

## Komunikacja dwukierunkowa

- Architektura klient-serwer jasno precyzuje role:  
klient: aktywny, serwer: pasywny
- Czasami potrzebujemy więcej...
- Wiemy, że *polling* nie jest efektywny
- Klient nie musi być „czystym” klientem, serwer nie musi być „czystym” serwerem...
- Brzmi dobrze, ale...
- Problemy: NAT, firewall,...  
– może się skończyć tak:

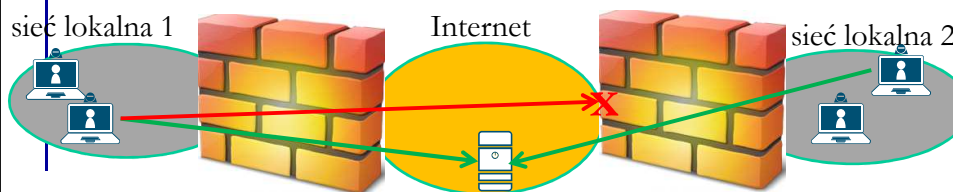


## Translacja adresów

- NAT, a tak naprawdę PAT
- Jak się skomunikować z komputerem za NAT?
- Jak działają aplikacje typu Team Viewer?
- STUN+TURN=ICE (choć nie ten...)

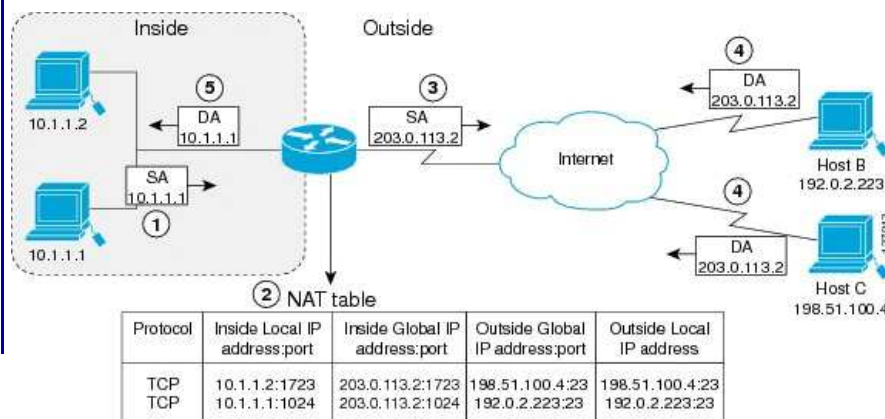
## Czy możliwe jest nawiązanie bezpośredniej łączności?

- Nawiązanie bezpośredniej łączności pomiędzy dwiema aplikacjami działającymi na urządzeniach z adresami prywatnymi jest trudne i może być... niemożliwe



- Może być konieczny pośrednik
- Warto spojrzeć: STUN, TURN

## Translacja adresów – PAT



## Tablica translacji PAT

- Uwzględnia L4
  - UDP, TCP
  - Co z innymi protokołami?
- Co daje połączeniowość protokołu w tym kontekście?
- Czas obecności wpisów przy braku aktywności: Cisco
  - domyślnie 24h dla TCP (chyba, że połączenie zostanie zamknięte lub przerwane: wówczas minuta) i 5 min. dla UDP – te wartości są często znacznie zmniejszane

## O czym warto pamiętać?

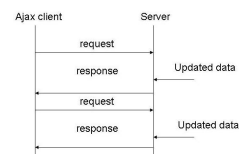
- Urządzenie NAT/PAT zazwyczaj nie podmienia adresów i portów przesyłanych wewnątrz wiadomości
- Zniknięcie wpisu w tablicy translacji: wiele powodów
  - przekroczenie czasu życia wpisu, ale też:
  - restart urządzenia
  - administracyjne usunięcie wpisów
- Zniknięcie wpisu w tablicy translacji: mogą być problemy...

## Komunikacja dwukierunkowa

- Pomysł: wykorzystać istniejący, ustanowiony przez klienta kanał komunikacyjny do komunikacji serwera z klientem – inicjowanej przez serwer
- Czy to będzie działać?
- Czy to będzie działać niezawodnie?
- Konieczne zabiegi:
  - rozsądne podtrzymywanie aktywności w kanale łączności
  - odbudowywanie zerwanego kanału łączności – kiedy? jak? przez kogo?

## Komunikacja dwukierunkowa

- Pomysł ICE:
  - Klient *też* uruchamia Object Adapter i instancjonuje serwanty (też staje się serwerem)
  - Komunikacja z obiektami (serwantami) klienta może się odbywać w ramach jednej asocjacji TCP ustanowionej przez klienta
- Pomysł gRPC:
  - Wywołanie strumieniowe strony serwerowej (*dalej*)
- Pomysł ~HTTP: *long polling*



## Podtrzymywanie aktywności

- **TCP:** keepalive – czasami wyłączony, domyślnie raz na dwie godziny ;)
- **ICE:**
  - tzw. heartbeat
  - dodatkowo mechanizm ACM (Active Connection Management) podtrzymujący lub usuwający połączenia TCP – ważny także ze względów efektywnościowych (wolny start vs. wykorzystanie zasobów)
- **Thrift:** wykorzystanie TCP keepalive
- **Websocket:** ramki kontrolne ping/pong (RFC 6455) lub TCP keepalive

- **gRPC:** HTTP/2 ping

(<https://github.com/grpc/grpc/blob/master/doc/keepalive.md>)

```

2870 261.673504 ::1 ::1 WebSocket 77 WebSocket Text [FIN] [MASKED]
2871 261.673562 ::1 ::1 TCP 64 3100 → 32291 [ACK] Seq=136 Ack=546 Win=2160128 Len=0
2872 261.675224 ::1 ::1 WebSocket 73 WebSocket Text [FIN]
2873 261.675266 ::1 ::1 TCP 64 32291 → 3100 [ACK] Seq=546 Ack=145 Win=327168 Len=0
3498 306.678640 ::1 ::1 TCP 65 [TCP Keep-Alive] 32291 → 3100 [ACK] Seq=545 Ack=145 Win=327
3499 306.678670 ::1 ::1 TCP 76 [TCP Keep-Alive ACK] 3100 → 32291 [ACK] Seq=145 Ack=546 Win
4832 351.682349 ::1 ::1 TCP 65 [TCP Keep-Alive] 32291 → 3100 [ACK] Seq=545 Ack=145 Win=327
4833 351.682383 ::1 ::1 TCP 76 [TCP Keep-Alive ACK] 3100 → 32291 [ACK] Seq=145 Ack=546 Win
4684 396.692598 ::1 ::1 TCP 65 [TCP Keep-Alive] 32291 → 3100 [ACK] Seq=545 Ack=145 Win=327

```

## Nie udało się...

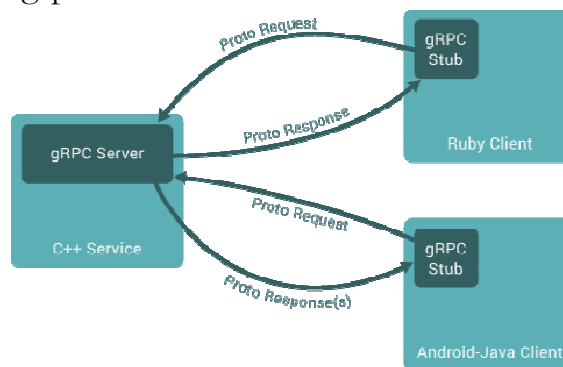
- Co może zrobić serwer w razie stwierdzenia utraty łączności z klientem?
- Czy klient wie, że serwer stracił z nim łączność?
- Jak przywrócić łączność?

# gRPC

## Wprowadzenie



gRPC = grpc Remote Procedure Call



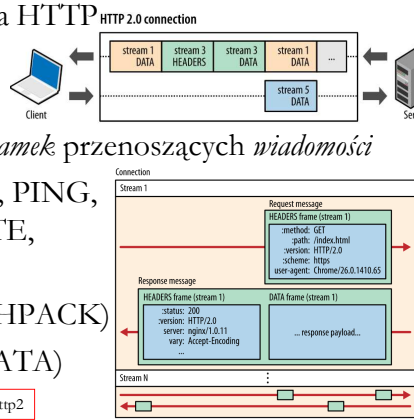
## Istotne cechy

- Usługi – nie obiekty
- gRPC **nie może** być nazwane *OO middleware*
- Komunikacja z wykorzystaniem transferu wiadomości (*message*) - ale to nie MOM!
- Ciekawa i przydatna funkcjonalność: strumieniowanie
- Serializacja: Protocol Buffers
- Komunikacja: HTTP/2 (metoda POST) (+opcjonalnie TLS)
  - Wstępne prace nad gRPC over HTTP/3 (G2)
- Obsługa wielu języków programowania
- Szeroko wykorzystywany: Google, Netflix, IBM, Cisco, Juniper, Spotify, Dropbox, Docker, Akka, Kubernetes...

## HTTP/2

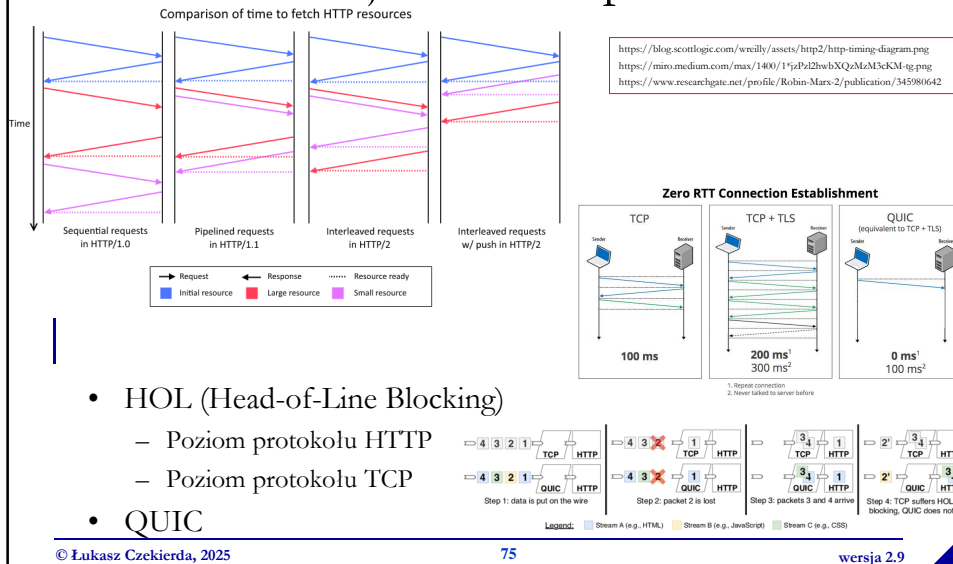
- Początki: SPDY (Google), standard od 2015 (RFC 7540)
- Pozostawiono kluczowe założenia HTTP
- Logiczne strumienie danych w pojedynczym połączeniu TCP
- Komunikacja z wykorzystaniem *ramek* przenoszących *wiadomości*
- Typy ramek: HEADERS, DATA, PING, GOAWAY, WINDOW\_UPDATE, PRIORITY, ...
- Binarne kodowanie nagłówków (HPACK)
- Kontrola przepływu (tylko dla DATA)

grafika: <https://developers.google.com/web/fundamentals/performance/http2>





# Komunikacja HTTP – porównanie



## Serializacja: Protocol Buffers

- Serializacja (ogólnie) – cechy
  - Tekstowa lub binarna
  - Zawierająca metadane lub nie
  - Opisana schematem lub nie
  - Ograniczona do języka, platformy itp. lub nie
- Jej realizacje: XML, JSON, Ice, Thrift, **Protocol Buffers**
- Jakie cechy ma serializacja Protocol Buffers?
- Jakie ma zalety? Jakiej ma wady?
- Gdzie jest wykorzystywana?

## proto – podstawowe informacje

- Wersja (np. syntax = "proto2"), istotniejsze różnice:
  - Proto2:
    - pola muszą być otagowane: optional/required
    - możliwość określenia domyślnej wartości pola
  - Proto3:
    - wszystkie pola są opcjonalne (optional)
    - pola nie mogą mieć deklarowanej domyślnej wartości
- Podstawowy element: wiadomość (*message*) → ~struktura
- Typy: int32, int64, float, double, string, bytes, bool, enum, sekwencje (repeated), map, message, ...

więcej na: <https://developers.google.com/protocol-buffers/docs/proto>

## proto – przykładowe wiadomości

```
message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3 [default = 10];
  enum Corpus {
    UNIVERSAL = 0;
    WEB = 1;
    IMAGES = 2;
    LOCAL = 3;
    NEWS = 4;
    PRODUCTS = 5;
    VIDEO = 6;
  }
  optional Corpus corpus = 4 [default = UNIVERSAL];
}
```

```
message SearchResponse {
  message Result {
    required string url = 1;
    optional string title = 2;
    repeated string snippets = 3;
  }
  repeated Result result = 1;
}
```

- message → struct

```
message Outer { // Level 0
  message MiddleAA { // Level 1
    message Inner { // Level 2
      required int64 ival = 1;
      optional bool  booly = 2;
    }
  }
}
```

więcej na: <https://developers.google.com/protocol-buffers/docs/proto>

## IDL gRPC: wykorzystanie i rozszerzenie definicji proto

```

message ArithmeticOpArguments {
    int32 arg1 = 1;
    int32 arg2 = 2;
}
message ArithmeticOpResult {
    int32 res = 1;
}
service Calculator {
    rpc Add (ArithmeticOpArguments) returns (ArithmeticOpResult) {}
}
message ComplexArithmeticOpArguments {
    OperationType optype = 1;
    repeated double args = 2;
}
service AdvancedCalculator {
    rpc ComplexOperation (ComplexArithmeticOpArguments) returns
        (ComplexArithmeticOpResult) {}
}

```

## Interfejs usługi gRPC – kilka uwag

- Brak możliwości rozszerzania definicji usług przez dziedziczenie
- Brak wyjątków
- Obsługa błędów – statusy wywołań, m.in.:
  - GRPC\_STATUS\_UNIMPLEMENTED
  - GRPC\_STATUS\_UNAVAILABLE
  - GRPC\_STATUS\_DEADLINE\_EXCEEDED
  - GRPC\_STATUS\_INVALID\_ARGUMENT

## Komunikacja strumieniowa

- Sposoby komunikacji w gRPC
  - Simple (unary) RPC
  - Server-side streaming RPC
  - Client-side streaming RPC
  - Bidirectional streaming RPC

```
service StreamTester {
  rpc GeneratePrimeNumbers(Task) returns (stream Number) {}
  rpc CountPrimeNumbers(stream Number) returns (Report) {}
}
```

- Strumieniowanie – dostarczanie wielu osobnych wiadomości przed zakończeniem wywołania
- Strumieniowanie jest **zawsze** inicjowane przez klienta

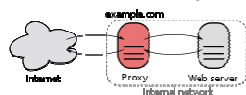
## Komunikacja strumieniowa – przykład

```
@Override
public void generatePrimeNumbers(Task request, StreamObserver<Number> responseObserver)
{
    System.out.println("generatePrimeNumbers");
    for (int i = 0; i < request.getMax(); i++) {
        if (isPrime(i)) { //zwłoka czasowa - dla obserwacji procesu strumieniowania
            Number number = Number.newBuilder().setValue(i).build();
            responseObserver.onNext(number);
        }
    }
    responseObserver.onCompleted();
}
```

```
Task request = Task.newBuilder().setMax(15).build();
Iterator<Number> numbers;
try {
    numbers = streamTesterBlockingStub.generatePrimeNumbers(request);
    while (numbers.hasNext())
    {
        Number num = numbers.next();
        System.out.println("Number: " + num.getValue());
    }
} catch (StatusRuntimeException ex) {
    Logger.log(Level.WARNING, "RPC failed: {0}", ex.getStatus());
    return;
}
```

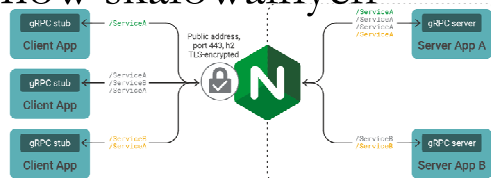
## Budowa systemów skalowalnych

### • Koncepcja reverse proxy



- Równoważenie obciążenia
- Routing wiadomości
- Terminowanie zabezpieczeń
- Wykorzystanie np. nginx, envoy

<https://www.nginx.com/blog/nginx-1-13-10-grpc/>



```
upstream grpcservers {
    server 192.168.20.11:50051;
    server 192.168.20.12:50051;
}

location /helloworld.Greeter {
    grpc_pass grpc://192.168.20.11:50051;
}

location /helloworld.Dispatcher {
    grpc_pass grpc://192.168.20.11:50052;
}

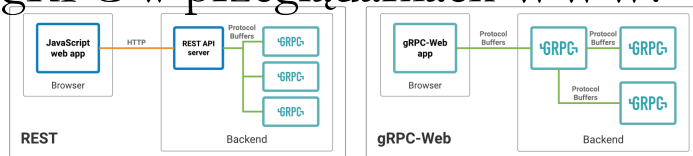
location / {
    root html;
    index index.html index.htm;
}
```

## Kontrola przepływu

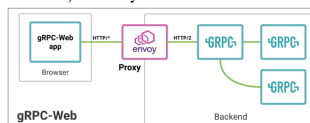
- Konsument może ograniczać tempo „produkcji” wiadomości
- Wykorzystanie kontroli przepływu HTTP/2
  - Każde żądanie HTTP/2 jest przesyłane w którymś ze strumieni
  - Tempo komunikacji może być ograniczane zarówno na poziomie pojedynczego strumienia jak i całego połączenia
- Tylko przy wywołaniach strumieniowych (unary: nie)
- Reactive gRPC (github: reactive-grpc)
  - Wykorzystanie kontroli przepływu (*back-pressure*) gRPC
  - Wskazania kontroli przepływu mogą się propagować do Reactive Streams, wsparcie także w akka-grpc

<https://github.com/salesforce/reactive-grpc>

## gRPC w przeglądarkach WWW?



- gRPC nie może być wprost używany w aplikacjach webowych (m.in. brak dostępu przeglądarki do surowych danych HTTP) → gRPC-Web
- Zaleta podejścia z prawej strony – jednolita reprezentacja danych: albo natywna (proto), albo tekstowa (Base-64, JSON)
- gRPC-Web: biblioteka JavaScript
- Nie jest wymagane HTTP/2
- Konieczna obecność reverse-proxy (konieczna translacja wiadomości)
- Nie jest dostępna pełna funkcjonalność gRPC
- Roadmap (github):
  - The binary protobuf encoding format is not most CPU efficient for browser clients
  - We plan to leverage WebTransport for bi-directional streaming.



## Zastosowania gRPC

- Aplikacje klient-serwer (klient: desktop lub mobile)
- Integracja w backendzie: łączenie (mikro)usług
- Ekspozycja API
  - konkurencyjny wobec REST i GraphQL
  - REST adresuje dane (zasoby), gRPC: procedury ich przetwarzania
- Ważne cechy
  - Bardzo dobra wydajność
  - Wykorzystanie najpopularniejszego protokołu Internetu
  - Efektywna komunikacja obustronna

## Dodatek: G2 – gRPC over HTTP/3

- Pewne różnice pomiędzy HTTP/2 i HTTP/3
  - HTTP/3 has different error codes from HTTP/2. HTTP/3 errors are sent via QUIC frames instead of an RST\_STREAM HTTP frame.
  - When an RPC has exceeded its deadline, the server will reset the stream. In HTTP/2, a stream is reset using the RST\_STREAM frame. The RST\_STREAM frame doesn't exist in HTTP/3. Instead, this action is performed using a QUIC frame, called RESET\_STREAM.
  - PING frames do not exist in HTTP/3, as QUIC provides equivalent functionality.
- Różnice w oferowanej funkcjonalności pomiędzy TCP i QUIC
- A trial implementation in grpc-dotnet is underway. .NET 6 is adding preview support for HTTP/3 to the .NET server and client. grpc-dotnet leverages that underlying HTTP/3 support.
- Ostatnie zmiany sprzed 4 lat...

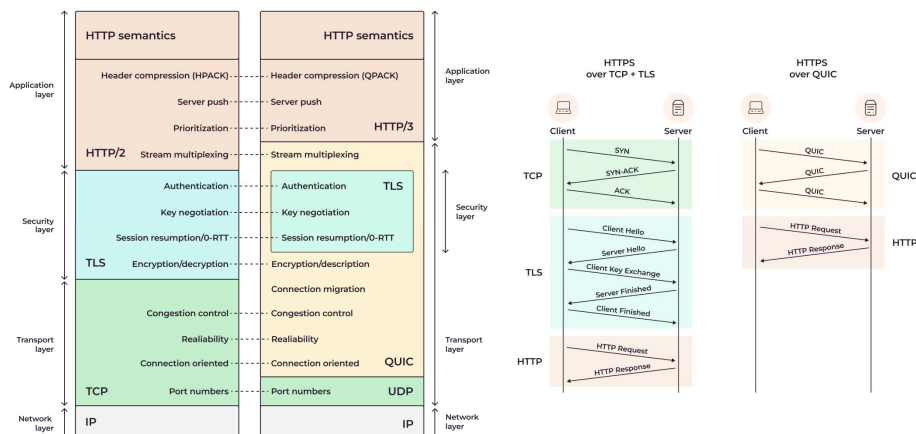
## Dodatek: gRPC directly over QUIC?

- The Go language implementation of gRPC over QUIC
  - <https://github.com/sssgun/grpc-quic> → ostatnie zmiany sprzed 4 lat...
- *The biggest consequence of using QUIC directly rather than HTTP/3 is that it would be a "custom protocol" and not understood by things like proxies, should users need to rely on them.*

## Dodatek: QUIC + HTTP/3

- Multiplexing of requests [in HTTP/3] is performed using the QUIC stream abstraction
- Each request-response pair consumes a single QUIC stream. Streams are independent of each other, so one stream that is blocked or suffers packet loss does not prevent progress on other streams.
- To support independent streams, QUIC performs flow control on a per-stream basis. It controls the bandwidth consumption of stream data at two levels:
  - on each stream individually, by setting the maximum amount of data that can be allocated to one stream
  - across the entire connection, by setting the maximum cumulative number of active streams

## Dodatek: HTTP/3






## Dodatek: WebTransport

- WebTransport is a web API that enables bidirectional and multiplexed communication between a web client and an HTTP/3 server. It supports both reliable and unreliable data transmission via streams and datagrams, respectively. It provides low latency, high throughput, and out-of-order delivery. It's intended for scenarios where WebSockets or WebRTC are not suitable or optimal, such as gaming, live streaming, or machine learning.
- Bidirectional: You can send and receive data in both directions between the client and the server.
- Multiplexed: You can open multiple logical channels (streams or datagrams) over a single connection.
- Streams: You can use streams to send and receive reliable, ordered, and flow-controlled data. Streams are analogous to TCP connections, but they can be opened and closed independently.
- Datagrams: You can use datagrams to send and receive unreliable, unordered, and congestion-controlled data. Datagrams are analogous to UDP packets, but they are encrypted and congestion-controlled by HTTP/3.
- Unreliable transport: You can use datagrams to send data that does not require reliability, such as real-time audio or video frames.
- Out-of-order delivery: You can use streams or datagrams to send data that does not require in-order delivery, such as multiplexed media chunks.

```
:method = CONNECT
:scheme = https           :status = 200
:authority = example.com:443 :protocol = webtransport
:path = /webtransport
:protocol = webtransport
origin = https://example.com
```

## Subiektywne porównanie technologii

- Efektywność komunikacji: Ice, Thrift, gRPC
- Ładne, bogate interfejsy, wyjątki: Ice, Thrift
- Podejście obiektowe: Ice
- Bogata funkcjonalność: Ice + icerpc 
- Możliwość użycia w aplikacjach Web: gRPC, Ice
- Łatwość integracji z „nowymi” technologiami: gRPC
- Popularność: gRPC
- Licencjonowanie: uwaga na Ice! (GPLv2 i komercyjna)
- A może warto spojrzeć na inne: DRPC (Go), Twirp, kRPC...?

## Podsumowanie zajęć

- Czy wiem, co to jest to middleware?
- Czy wiem, na czym polega specyfika i wartość dodana technologii middleware w stosunku do omawianych wcześniej rozwiązań?
- Czy znam architekturę technologii middleware?
- Czy znam obszary ich zastosowań oraz ich ograniczenia?
- Czy umiem poprawnie definiować interfejsy komunikacji zdalnej?
- Czy znam zaawansowane mechanizmy tych technologii?
- Czy umiem stworzyć efektywny i niezawodny system rozproszony wykorzystujący te technologie?

# KONIEC