Systemy wbudowane dla automatyki W12

dr inż. Krzysztof Urbański

Dodatkowe materiały do kursu
na prawach rękopisu

Źródła: materiały własne, publicznie dostępne dokumenty w dostępie otwartym oraz noty katalogowe i oficjalna dokumentacja prezentowanych rozwiązań.

Push a HTTP

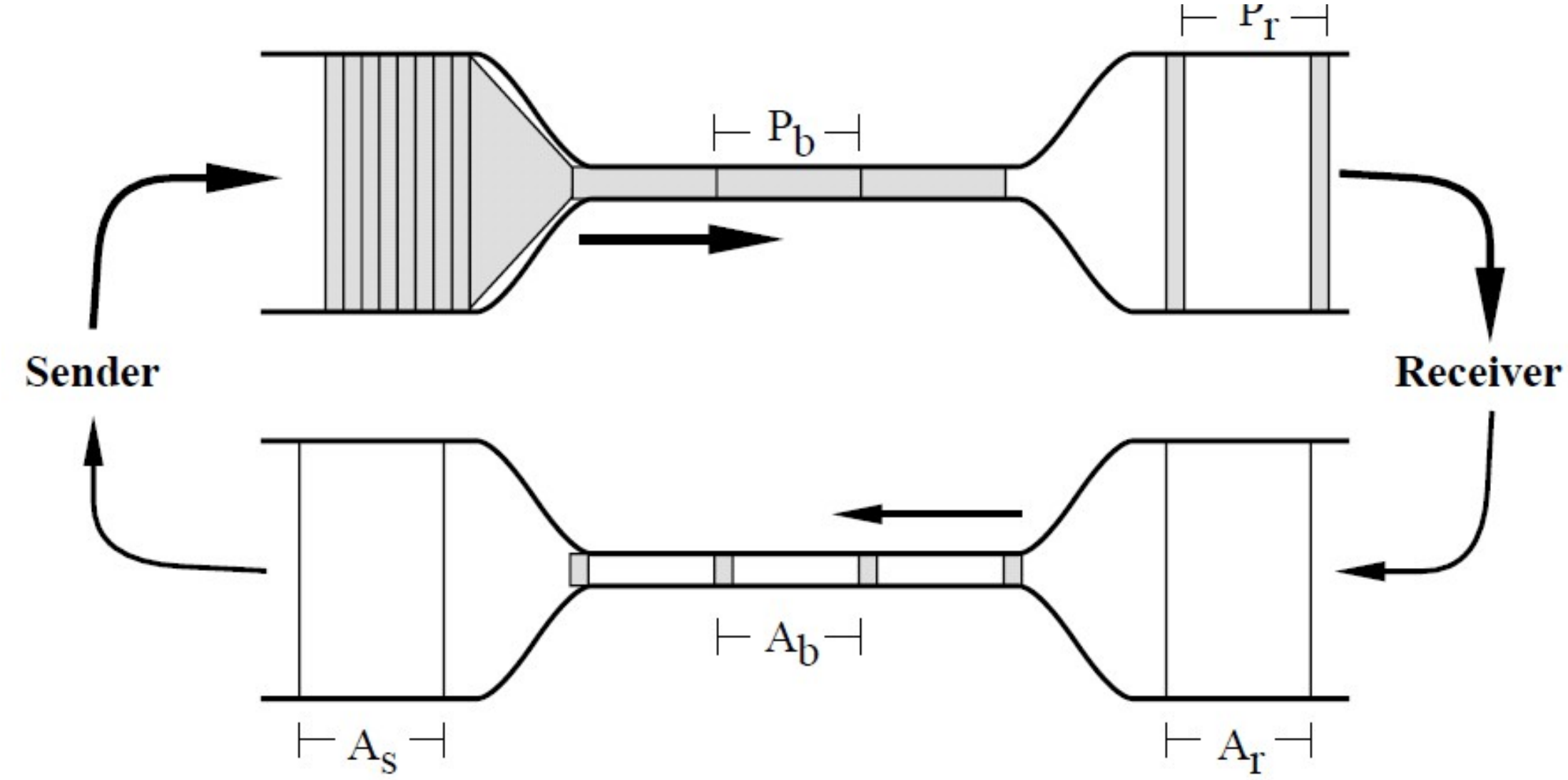
Serwer 8053 został wykonany trochę inaczej – symuluje zachowanie mechanizmu push.

Typowo schemat odpytywania (HTTP/1.0, 1.1) → nieefektywne np. w IoT lub ogólnie kiedy chcemy mieć dostęp do informacji w czasie rzeczywistym

1. HTTP/2 Server Push →stopniowa rezygnacja na rzecz innych mechanizmów, np. z HTTP/3
2. WebSockets (RFC 6455) – zestawiane i utrzymywane jest połączenie TCP fullduplex.
3. Server-Sent Events (SSE, HTML5) – serwer może wysyłać klientowi dane przez standardowe połączenie HTTP. Tylko w jedną stronę, łątwiejsze w użyciu niż WebSockets, mechanizm subskrypcji.
4. Long Polling: odpytywanie, ale z bardzo długim czasem odpowiedzi. Mniej efektywne niż WebSockets lub SSE. Obecnie przestarzałe, ale nadal stosowane.
5. Powiadomienia Web Push (Web Push Notifications) – docierają nawet gdy strona www jest zamknięta, pod warunkiem że się na to zgodzimy. Wymaga pewnych dodatkowych usług działających w tle.

Slow start

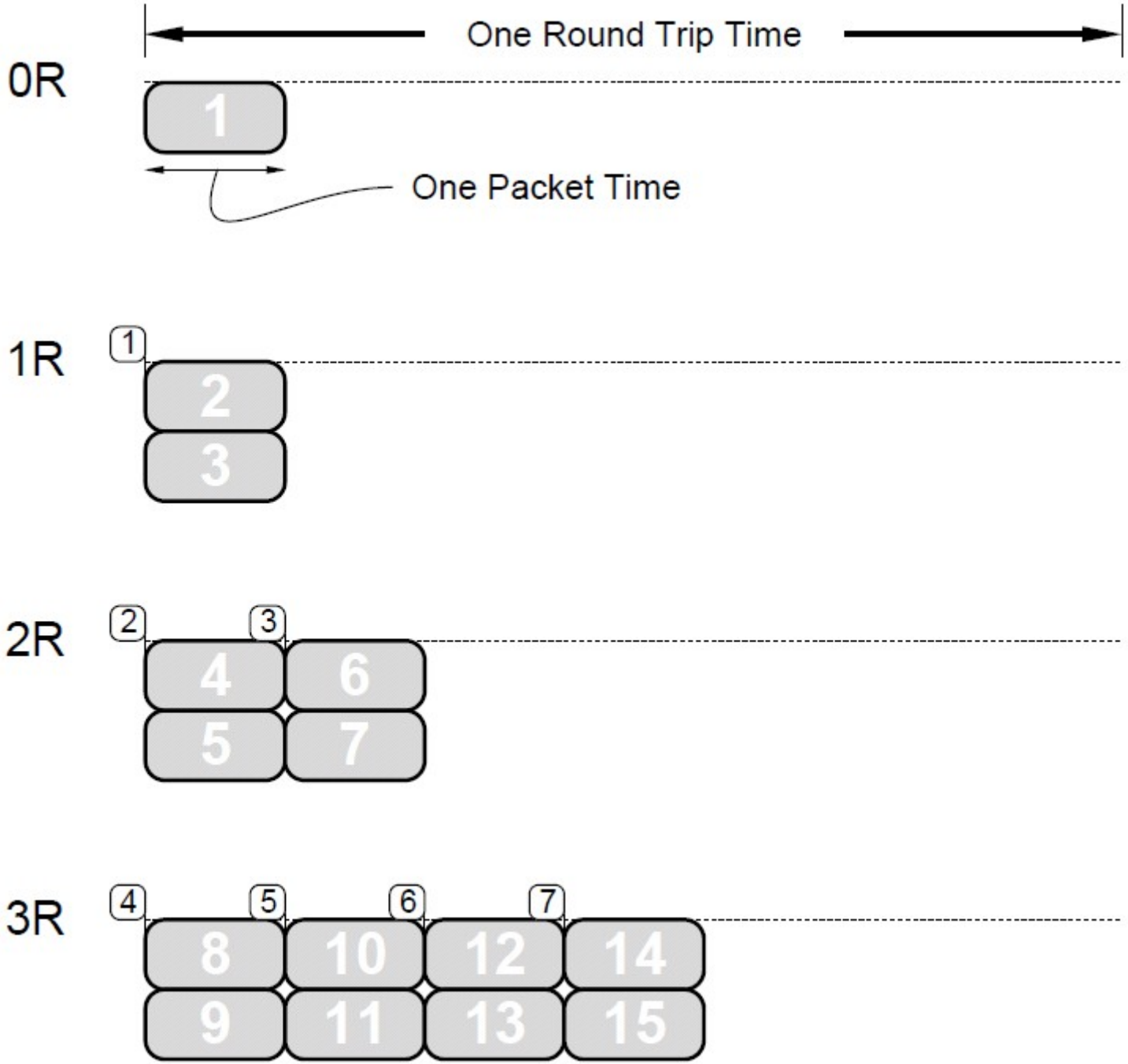
Jest jednym z rozwiązań zwiększających wydajność TCP (inne to FastRetransmit, FastRecovery, **algorytm Nagle'a**).
Mechanizm „wolnego startu" → TCP posiada wbudowany mechanizm dostrajania szybkości wrzucanych do sieci pakietów w zależności od parametrów sieci na drodze klient-serwer. Algorytm ten zakłada początkowo, że parametry sieci są kiepskie.
Dopóki się da, następuje zwiększanie ilości i szybkości generowania pakietów, aż do wystąpienia nasycenia (szczegóły – dalsza część wykładu).
Uboczny skutek – „świeże" połączenie TCP jest powolne.



Źródło: [2]

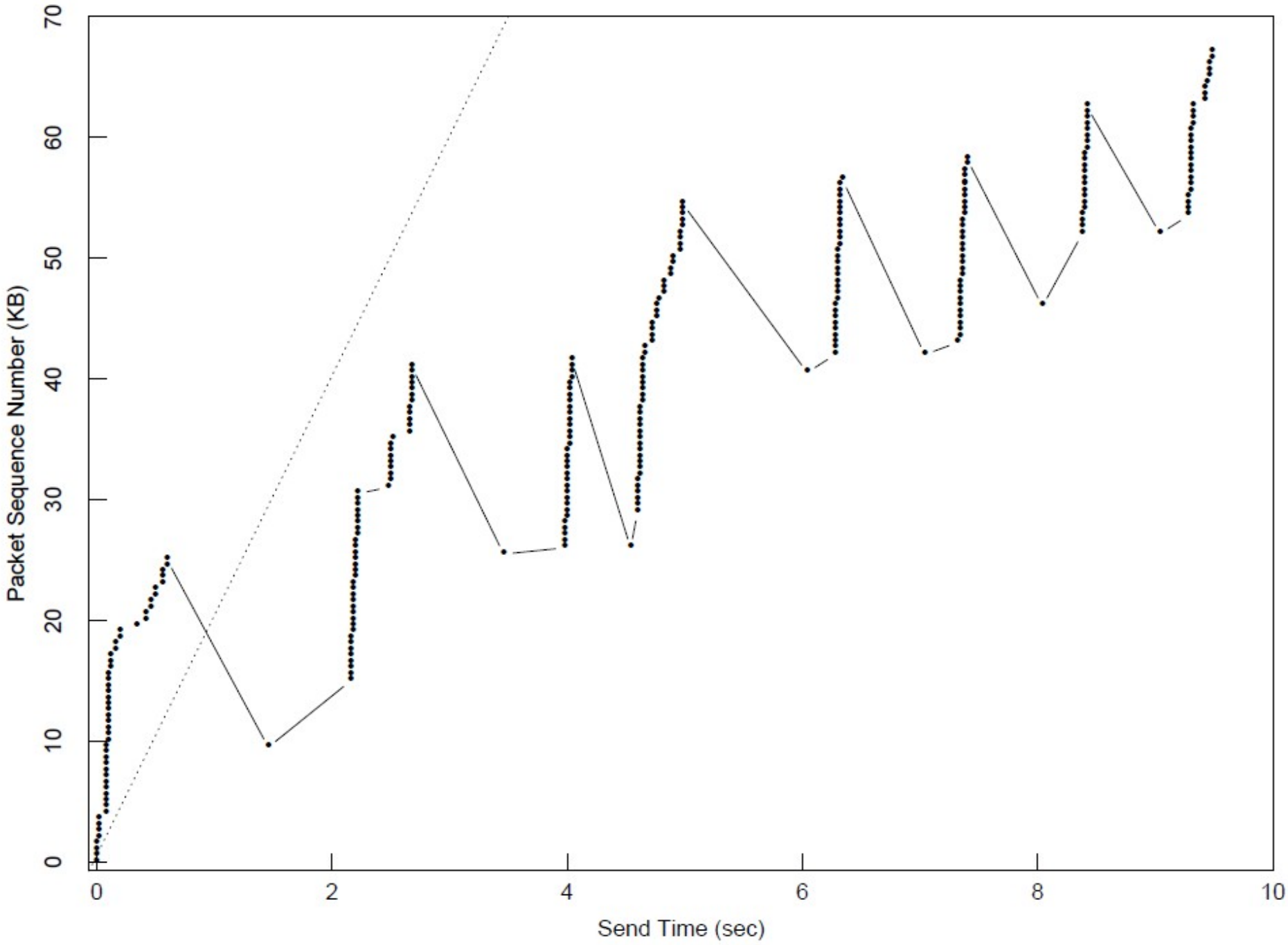
Dostrajanie wielkości okna TCP.

Efekt przejścia danych z szybkiej sieci lokalnej do powolnego łącza z Internetem.

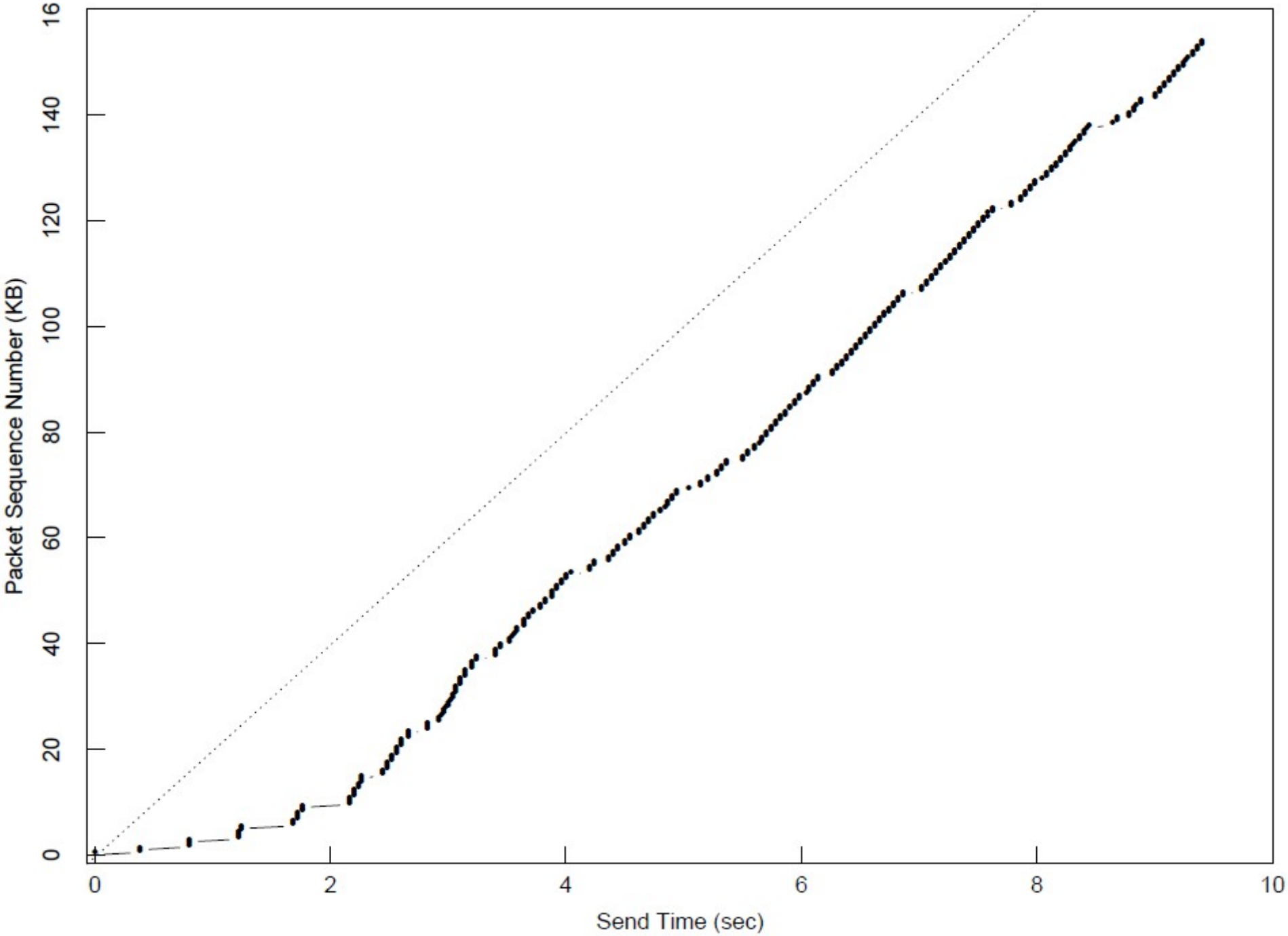


Źródło: [2]

Rozwój wydarzeń w *slow-start.*



źródło: [2]. Przebieg transmisji bez *slow-start*.



źródło: [2]. Efekt użycia *slow-start*.

Skutek: kiepska wydajność przy transmisji niewielkich plików (a z takimi najczęściej mamy do czynienia).

→**to dlatego FTP nie nadaje się do przesłania dużej liczby niewielkiech plików**

Efekt towarzyszący: zestawienie połączenia TCP wymaga przynajmniej trzech pakietów tam-i-z-powrotem → dodatkowe opóźnienia na dalekich trasach.

Wniosek: HTTP 1.0 będzie **zwykle** wolniejszy niż HTTP 1.1 (uwaga – nie zawsze!)

Network Performance Effects of HTTP/1.1, CSS1, and PNG
http://www.w3.org/TR/NOTE-pipelining-970624

W artykule porównane zostały wydajności kilku konfiguracji:

- HTTP/1.0
- HTTP/1.1 z trwałymi połączeniami
- HTTP/1.1 z potokowym przetwarzaniem żądań
- HTTP/1.1 potokowym przetwarzaniem żądań i kompresją danych
- Wpływ *slow-start* na wydajność HTTP/1.1

Przypomnienie:

1.0 – każda transakcja = osobne połączenie TCP

1.1 – w jednym połączeniu TCP można przesłać wiele obiektów (html, obrazki, css, js itp.)

Typowa strona WWW: wiele (-naście, -dziesiąt obiektów).

Kilkanaście połączeń nawiązanych równocześnie w celu załadowania pojedynczej strony? Popularny, często odwiedzany serwer nie ma szans!

„Cienkie” (odchudzone) implementacje stosu TCP/IP ograniczają liczbę połączeń TCP, np. DS80C400 – maksymalnie 32 połączenia.

Potokowe przetwarzanie żądań HTTP 1.1

Typowy schemat „żądaj dokumentu A → odbierz dokument A → żądaj dokumentu B → odbierz dokument B → …” – nieefektywny na dalekich trasach oraz w innych przypadkach występowania opóźnień (spowodowanych działaniem klienta, serwera lub sieci).

HTTP 1.1 przewiduje przetwarzanie potokowe → możliwe wysłanie wielu żądań bez zakończenia obsługi poprzednich.

Buforowanie zawartości

HTTP 1.1 ma ulepszone (właściwie: pojawiły się) mechanizmy wspomagające obsługę *cache* (buforowanie zawartości)

Oznaczanie zakresów danych do pobrania (1.1)

1. Ładowanie strony z wieloma obrazkami: aby właściwie wyświetlić stronę, należy znać rozmiary wszystkich jej obiektów → załadujmy tylko „początki” plików GIF/PNG/JPEG
2. Wznawianie przerwanej transmisji

Wybrane wyniki

	First Time Retrieval				Cache Validation			
	Pa	**Bytes**	**Sec**	%ov	Pa	**Bytes**	**Sec**	%ov
HTTP/1.0	489.4	**215536**	**0.72**	8.3	365.4	**60605**	**0.41**	19.4
HTTP/1.1	244.2	**189023**	**0.81**	4.9	98.4	**14009**	**0.40**	21.9
HTTP/1.1 Pipelined	175.8	**189607**	**0.49**	3.6	29.2	**14009**	**0.23**	7.7
HTTP/1.1 Pipelined w. compression	139.8	**156834**	**0.41**	3.4	28.4	**14002**	**0.23**	7.5

Table 5 - Apache - High Bandwidth, Low Latency

	First Time Retrieval				Cache Validation			
	Pa	**Bytes**	**Sec**	%ov	Pa	**Bytes**	**Sec**	%ov
HTTP/1.0	559.6	**248655.2**	**4.09**	8.3	370.0	**61887**	**2.64**	19.3
HTTP/1.1	309.4	**191436.0**	**6.14**	6.1	104.2	**14255**	**4.43**	22.6
HTTP/1.1 Pipelined	221.4	**191180.6**	**2.23**	4.4	29.8	**15352**	**0.86**	7.2
HTTP/1.1 Pipelined w. compression	182.0	**159170.0**	**2.11**	4.4	29.0	**15088**	**0.83**	7.2

Table 7 - Apache - High Bandwidth, High Latency

	First Time Retrieval				Cache Validation			
	Pa	**Bytes**	**Sec**	%ov	Pa	**Bytes**	**Sec**	%ov
HTTP/1.1	308.6	**187869**	**65.6**	6.2	89.0	**13843**	**11.1**	20.5
HTTP/1.1 Pipelined	281.4	**187918**	**53.4**	5.7	26.0	**13912**	**3.4**	7.0
HTTP/1.1 Pipelined w. compression	233.0	**157214**	**47.2**	5.6	26.0	**13905**	**3.4**	7.0

Table 9 - Apache - Low Bandwidth, High Latency

Dalsze zwiększanie wydajności HTTP (a przy okazji TCP)

Okno TCP – optymalny rozmiar zależy od parametrów sieci.
sensowny rozmiar jest rzędu opóźnienie × przepustowość.

Domyślny rozmiar okna można zmienić (Windows, Linux), ponadto aplikacja może sterować tą wartością. Rozmiar okna jest dostrajany dynamicznie (ale to już zależy od konkretnej implementacji stosu TCP)

Zbyt małe okno – przede wszystkim straty czasu (przestoje) spowodowane oczekiwaniem na nadejście potwierdzenia

Zbyt duże okno – większe zużycie pamięci oraz straty spowodowane retransmisjami w razie utraty pakietów (złej jakości łącza).

Gospodarka pasmem – współdzielenie połączeń

Mitologia stosowana: „mając łącze 100 Mbps możemy podłączyć 100 osób, a każdy uzyska 1 Mbps”.
W większości przypadków jest to zdanie nieprawdziwe.

1. Przy niezbyt intensywnym obciążeniu sieci (użytkownicy odwiedzają strony WWW) można odnieść wrażenie, że każdy ma do dyspozycji łącze dużo szybsze niż średnio 1 Mbps.
2. W odwrotnym przypadku (intensywnie używane P2P) następuje gwałtowne załamanie i ograniczenie pasma – poniżej 1 Mbps (nawet do 0 – dlaczego?)

→ zjawisko i tematyka znane od wielu lat, ale do tej pory jest to wiedza tajemna dla zwykłych użytkowników domowych „routerów”.

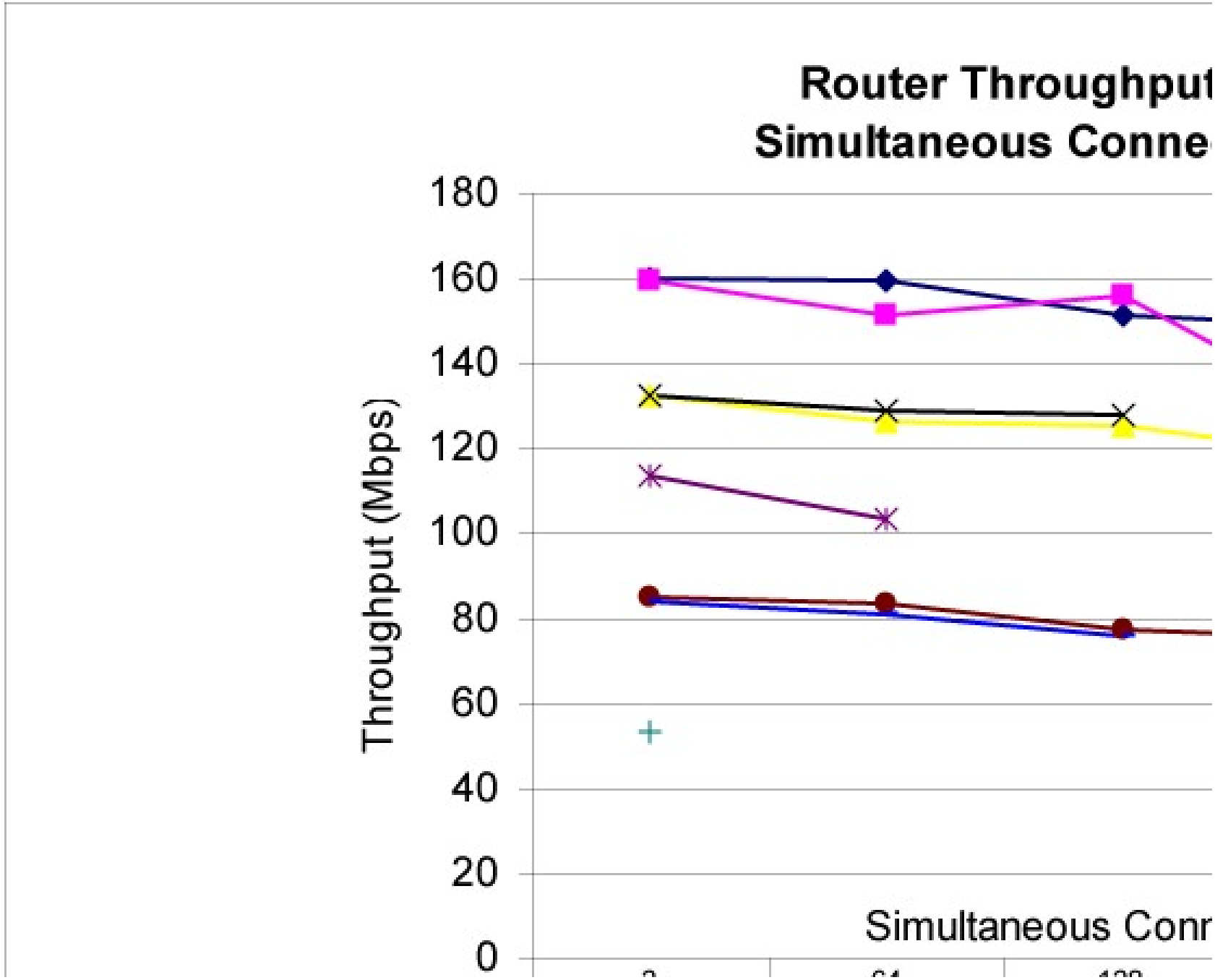
Źródło: Congestion Avoidance and Control

Van Jacobson, Lawrence Berkeley Laboratory
Michael J. Karels, University of California at Berkeley
November, **1988**

Tendencyjny przykład: leciwy „domowy” NAT/router: D-Link DSL-504

RAM: 16+2 Mb (OS + dodatki) – za mało ;)
limity czasowe:

- ICMP query: 10 seconds;
- UDP: 30 seconds
- TCP/IP (established): 300 seconds
- TCP/IP (other): 15 seconds
- GRE: 120 seconds



źródło: www.tomsnetworking.com

Sieciowa wymiana danych bitowych między C i C#

Zdarza się, że pewne aplikacje (szczególnie serwery) już powstały i świetnie działają, używając protokołów wymiany danych o strukturze bitowej (nowszymi rozwiązaniami byłby format tekstowy, a jeszcze lepszym — coraz chętniej stosowany język XML).

Jak to wszystko pogodzić z bardzo wygodnym w użyciu językiem **C#** ? (patrz: zarządzanie pamięcią w C#)

Zacznijmy od definicji pewnej struktury w języku C:

```
#pragma pack (push, store_pack)
#pragma pack (1)
struct t_net_data
{
    char prod[4]; //np. ID1\0
    int app_version;
    char name[101];
    short flags;
    char date[11]; //2000-00-00\0
};
#pragma pack (pop, store_pack)
```

Coś takiego jest przekazywane siecią Internet z użyciem protokołu TCP lub UDP. Chcemy, aby po drugiej stronie odebrała te dane aplikacja w języku C#

Typowym podejściem byłoby zdefiniowanie klasy w C#:

```
public class network_data
{
    public String prod;
    public int app_version;
    public String name;
    public short flags;
    public String date;
}
```

Gdyby jednak przyszło do odczytu danych z gniazda sieciowego, należałoby krok po kroku odkodować odebrane dane i wypełnić pola obiektu powyższej klasy. Kiedy pól jest bardzo dużo, a co więcej — zmienia się struktura danych w trakcie rozwoju aplikacji, ten sposób będzie zgubny dla programisty (wiele miejsc na popełnienie błędów, ogromne fragmenty nieczytelnego kodu).

Dlaczego w powyższym przykładzie tak trudno ustalić, ile zajmują w pamięci i jak są ułożone poszczególne pola? W przeciwieństwie do języka C++, pola tej klasy nie tylko nie są układane po kolei, ale w ogóle nie muszą stanowić spójnego obszaru pamięci. Pamięcią zarządza mechanizm podobny jak ten obecny w JVM. W dodatku obiekty klasy String (napisy) mogą zmieniać swoją długość w trakcie działania aplikacji. Aby było trudniej, są one kodowane jako napisy UNICODE.

C# na szczęście ma wbudowane mechanizmy, które na poziomie samego języka ułatwiają wymianę danych o strukturze bitowej. Oto przykład struktury zgodnej z przedstawionym wcześniej przykładem. Dane odbierane z sieci (np. pakiety UDP) mogą być bezpośrednio ładowane do tej struktury, podobnie jak wypełniając jej pola można przygotować paczkę danych do wysłania.

```
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential, Pack=1, CharSet=CharSet.Ansi)]
public struct network_data
{
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 4)]
    public String prod;
    public int app_version;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 101)]
    public String name;
    public short flags;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 11)]
    public String date;
}
```

Demonstracja jak może wyglądać wysłanie tych danych przez gniazdo UDP:

```
public network_data data;
int rawsize = Marshal.SizeOf(data);
IntPtr buffer = Marshal.AllocHGlobal(rawsize);
byte[] rawbytes = new byte[rawsize];

int sendto(UdpClient udp, string rhost, int rport, network_data data)
{
   //konwersja struktury do tablicy bajtów
   Marshal.StructureToPtr(data, buffer, false);
   Marshal.Copy(buffer, rawbytes, 0, rawsize);

   //wysłanie pakietu UDP
   int len = 0;
   try
   {
      len = udp.Send(rawbytes, rawsize, rhost, rport);
   }
   catch (SocketException) { return -1; }
   return len;
}
```

...oraz odbiór (tym razem gniazdo TCP):

```
public network_data data;
int rawsize = Marshal.SizeOf(data);
IntPtr buffer = Marshal.AllocHGlobal(rawsize);
byte[] rawbytes = new byte[rawsize];

int recv(Socket s)
{
    int len = 0;
    do {
        try
        {
            len = s.Receive(rawbytes, rawsize, SocketFlags.Peek);
        }
        catch (SocketException) { return -1; }
    } while (len != rawsize);
    try
    {
        len = s.Receive(rawbytes, rawsize, 0);
    }
    catch (SocketException) { return -1; }
    Marshal.Copy(rawbytes, 0, buffer, rawsize);
    data = (network_data)Marshal.PtrToStructure(buffer, typeof(network _data));
    return 0;
}
```

XML (eXtensible Markup Language)

- Narodziny XML: 1997
- Pierwszy standard (1.0, edycja pierwsza): 1998 r.
- Bieżąca edycja ma numer 4 i jest dość świeża (sierpień 2006 r.)

Główne założenia XML

- podstawowa jednostka: znak
- znaki tworzą elementy języka XML (entities)
- drzewiasta struktura
- informacje zapisane w pliku tekstowym
- wsparcie dla UNICODE
- ściśle ustalone reguły →

Te reguły to poprawność składniowa (*well-formedness*)

Każdy dokument XML:

- zawiera dokładnie jeden główny element (wszystkie inne elementy są potomkami głównego)
- wszystkie tagi są domknięte, przy czym tagi puste mogą być „samodomykające się”
- tagi otwierajace i zamykajace sa takie same (nie występuje przecinanie tagów)
- atrybuty występują tylko raz w elemencie
- spełnia dodatkowe reguły narzucone przez uzytkownika (np. element musi być liczbą całkowitą)

DTD (Document Type Definition)

Pozwala zdefiniować ograniczenia określające **formalną strukturę** dokumentu XML. W ten sposób użytkownik może dodatkowo narzucić pewne reguły, dzięki którym dostosuje dokument do swoich wymagań.

Elementy:

<! ELEMENT *nazwa_elementu opis* >

Dane:

(#PCDATA) tag bedzie parsowany i pokazywany
(#CDATA) tag nie bedzie parsowany ani pokazywany

DTD jest uważany za przestarzały spadek po wersji 1.0 XML, nie pozwala w pełni wykorzystać możliwości języka XML.

Przykład:

```
<!ELEMENT studenci (student*)>
<!ELEMENT student (imie+, nazwisko, semestr?, nralbumu)>
<!ELEMENT imie (#PCDATA)>
<!ELEMENT nazwisko (#PCDATA)>
<!ELEMENT semestr (#PCDATA)>
<!ELEMENT nralbumu (#PCDATA)>
```

- element **studenci** może składać się z 0 lub dowolnej liczby elementów **student** (znak '*')
- element **student** musi składać się z elementów **imie**, **nazwisko**, **nralbumu** oraz opcjonalnie ('?') **semestr**
- **imie** musi być przynajmniej jedno ('+'), ale może być więcej
- **imie**, **nazwisko**, **semestr** i **nralbumu** to dane znakowe

pasuje do schematu:

```
<?xml version="1.0" encoding="ISO-8859-2" ?>
<!DOCTYPE studenci (View Source for full doctype...)>
- <studenci>
  - <student>
      <imie>Jasiu</imie>
      <nazwisko>Kowalski</nazwisko>
      <semestr>urlop dziekański</semestr>
      <nralb>123456</nralb>
    </student>
  </studenci>
```

nie pasuje do powyższego schematu:

```
<?xml version="1.0" encoding="ISO-8859-2" ?>
<!DOCTYPE studenci (View Source for full doctype...)>
- <studenci>
  - <student>
      <imie>Jasiu</imie>
      <nazwisko>Kowalski</nazwisko>
      <pesel>80010112345</pesel>
      <semestr>urlop dziekański</semestr>
    </student>
  </studenci>
```

XSD (XML Schema Definition)

Przykładowy fragment schematu XSD:

```
<xs:element name="DOEParams">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="ID" msdata:ReadOnly="true" msdata:AutoIncrement="true"
type="xs:int"/>
         <xs:element name="Name" type="xs:string" minOccurs="0" />
         <xs:element name="Min" type="xs:double" minOccurs="0" />
         <xs:element name="Max" type="xs:double" minOccurs="0" />
         <xs:element name="Unit" type="xs:string" minOccurs="0" />
         <xs:element name="Log" type="xs:boolean" />
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Entities (jednostki)

Predefiniowane: **& (&) < (<) > (>) ' (') " (")**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE przyklad [
<!ENTITY uzytkownik "Krzysztof.Urbanski" >
<!ENTITY serwer "pwr.wroc.pl" >
<!ENTITY email "&uzytkownik;@&serwer;" >
]>
<glowny>Pisz na: &email;</glowny>
```

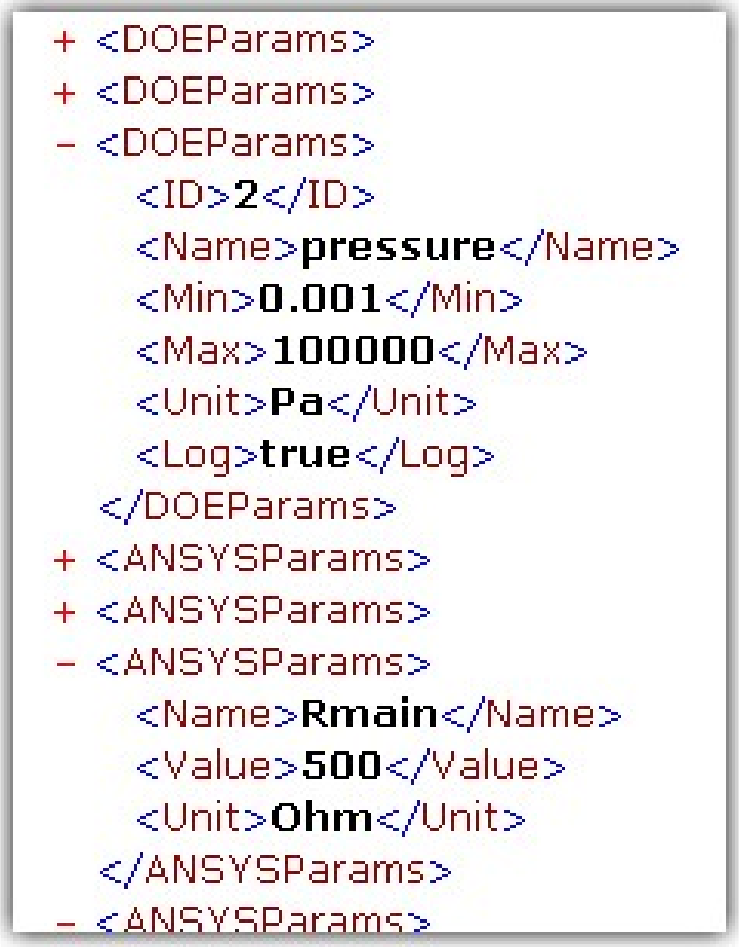
```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE przyklad (View Source for full doctype...)>
<glowny>Pisz na: Krzysztof.Urbanski@pwr.wroc.pl</glowny>
```

Inny przykład:

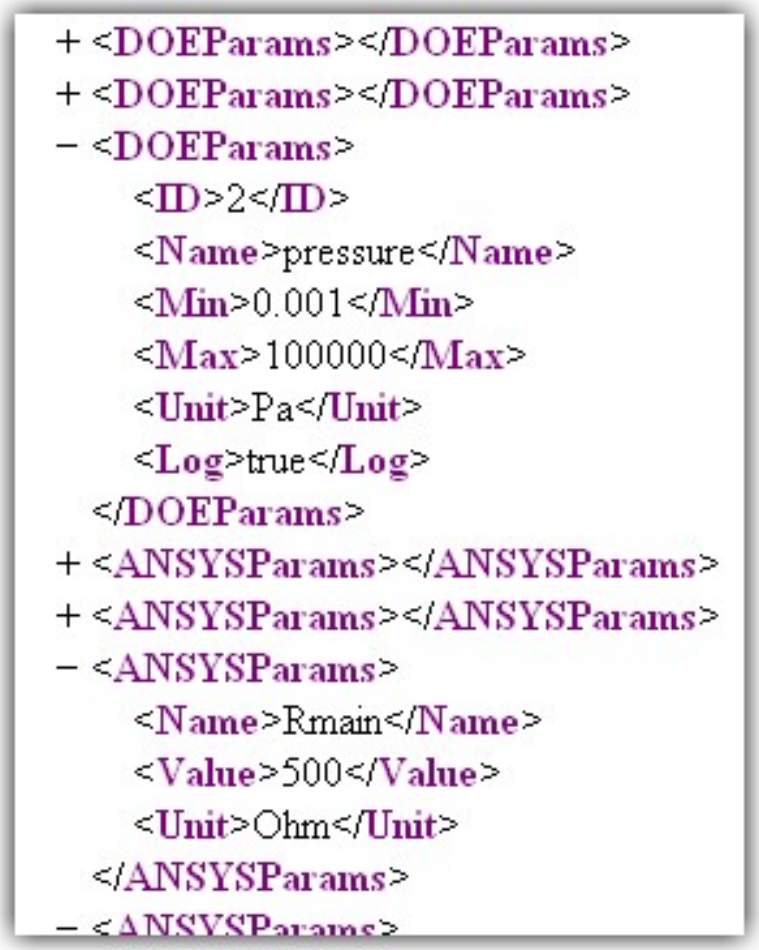
```
<!DOCTYPE kju [ <!ENTITY deg "&#176;"> ]>
```

W dokumencie XML możemy teraz pisać: °

Plik XML: jak go widzi przeglądarka?



IE



Firefox

W praktyce nie ma to jednak większego znaczenia, gdyż możliwe jest…

…skojarzenie XML z arkuszem CSS

Arkusz stylu [arkusz_dla_xml.css]

```
Pozdrowienia {
    display: block;
    font-size: 24pt;
    font-weight: bold;
    color: #9a342d;                    ← oczywiście kolory logotypu PWr!
    background-color: #fff7e0;
}
```

Plik [helloworld.xml]

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/css" href="arkusz_dla_xml.css"?>
<Pozdrowienia>Hello World!</Pozdrowienia>
```

Skoro XML „potrafi” przechować prawie każdy zestaw danych, to mogą to także być dane obiektów, a struktura samego dokumentu może uwzględniać też wzajemne ich powiązania.

Jeśli połączyć tę „umiejętność” XML z faktem, że jest to plik tekstowy i w dodatku standard ściśle opisuje budowę takiego pliku, otrzymamy przenośny format wymiany danych między różnymi systemami i różnymi programami.

Idąc dalej: skoro podstawową jednostką danych w XML jest znak, omijamy jeden z problemów transmisji sieciowych, jakim jest różnica w kodowaniu kolejności bajtów w liczbach (np. int, short, float, double).

Stąd już tylko krok do poznania ciekawego mechanizmu, którym jest…

…Serializacja z użyciem XML

czyli przekształcanie obiektów w strumień bajtów. Taki strumień można np. zapisać na dysku lub przesłać przez sieć, a później na jego podstawie odtworzyć pierwotny obiekt, czyli wykonać **deserializację**.

Dla obiektów w Javie, Pythonie czy C# są to umiejętności wrodzone,

obiekty w C++ muszą się tego uczyć od podstaw.

JSON Connor (JavaScript Object Notation)

- **lekki** format wymiany danych tekstowych
- **łatwy** do odczytania i zapisu **dla ludzi**
- **łatwy** do parsowania i generowania **dla maszyn**.
- JavaScript to zmyłka: JSON jest niezależny od języka programowania i powszechnie używany w wielu miejscach.

Kluczowe cechy JSON

Struktura JSON opiera się na parach klucz-wartość oraz listach.

Wspierany przez różne języki, np. Python, Java, C#, PHP, Ruby, Go itd.

Zastosowania:

- API webowe: najczęstszy format danych w komunikacji między klientami a serwerami (RESTful API).
- Pliki konfiguracyjne.
- Przechowywanie danych: np. MongoDB używa BSON (Binary JSON) do przechowywania dokumentów.
- Komunikacja międzyprocesowa.

Obiekt (Object): **nieuporządkowany** zbiór par klucz-wartość

Zaczyna się i kończy nawiasami klamrowymi { }.

Klucze muszą być ciągami znaków ujętymi w cudzysłowy (").

Wartości mogą być dowolnego typu danych JSON.

Pary klucz-wartość są oddzielone przecinkami.

Przykład: {"imie": "Jan", "wiek": 99}

Tablica (Array): **uporządkowana** lista wartości

Zaczyna się i kończy nawiasami kwadratowymi [ ].

Wartości są oddzielone przecinkami.

Wartości mogą być dowolnego typu danych JSON.

Przykład: ["TCP", "UDP", "HTTP"]

Typy danych

- Ciąg znaków (String) – ujęty w znaki cudzysłów.
- Liczba (Number) - całkowita lub zmiennoprzecinkowa 3.14, 1e5, 2.
- Wartość logiczna (Boolean): true albo false.
- Tablica (Array) – j.w.
- Obiekt (Object) – j.w.
- Null: zapisywane jako null

```
{
   "imie": "Jan",
   "wiek": 99,
   "zdal": null,
   "oceny": { "SWA-W": 3.0, "SWA-L": "5.5" },
   "hobby": [ "Python", "embedded", "sieci komputerowe" ],
   "obecny": true
}
```