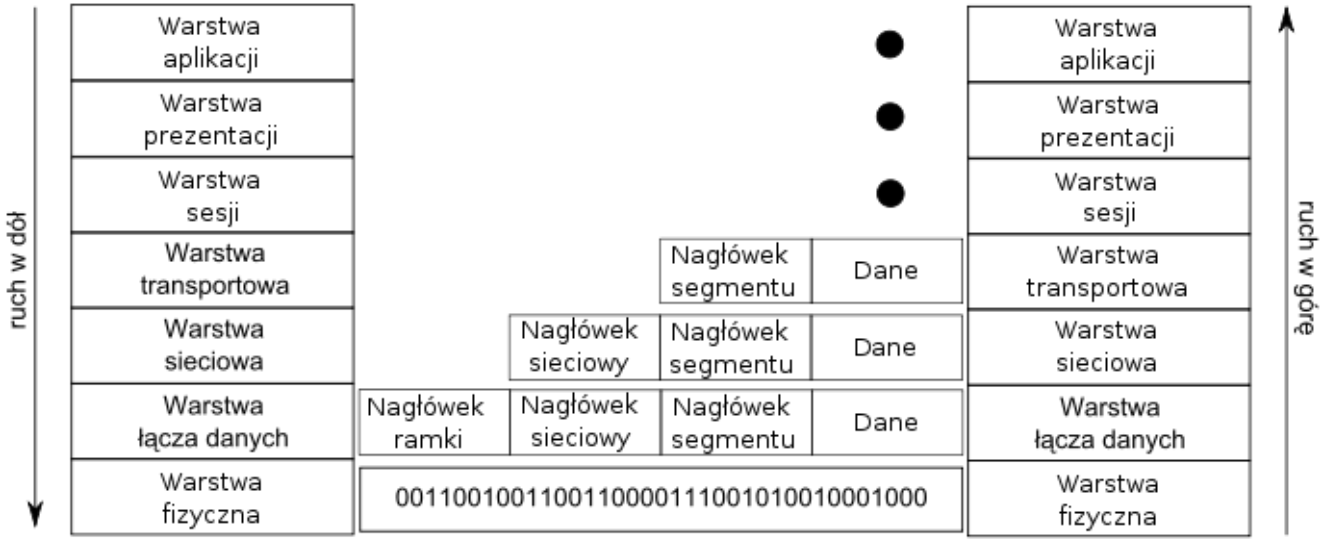
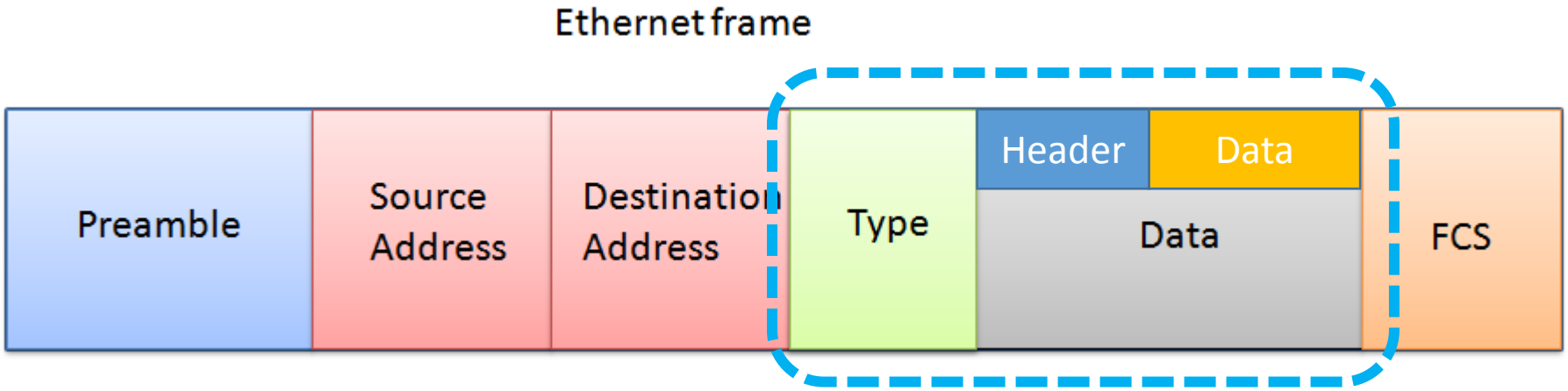
Komunikacja

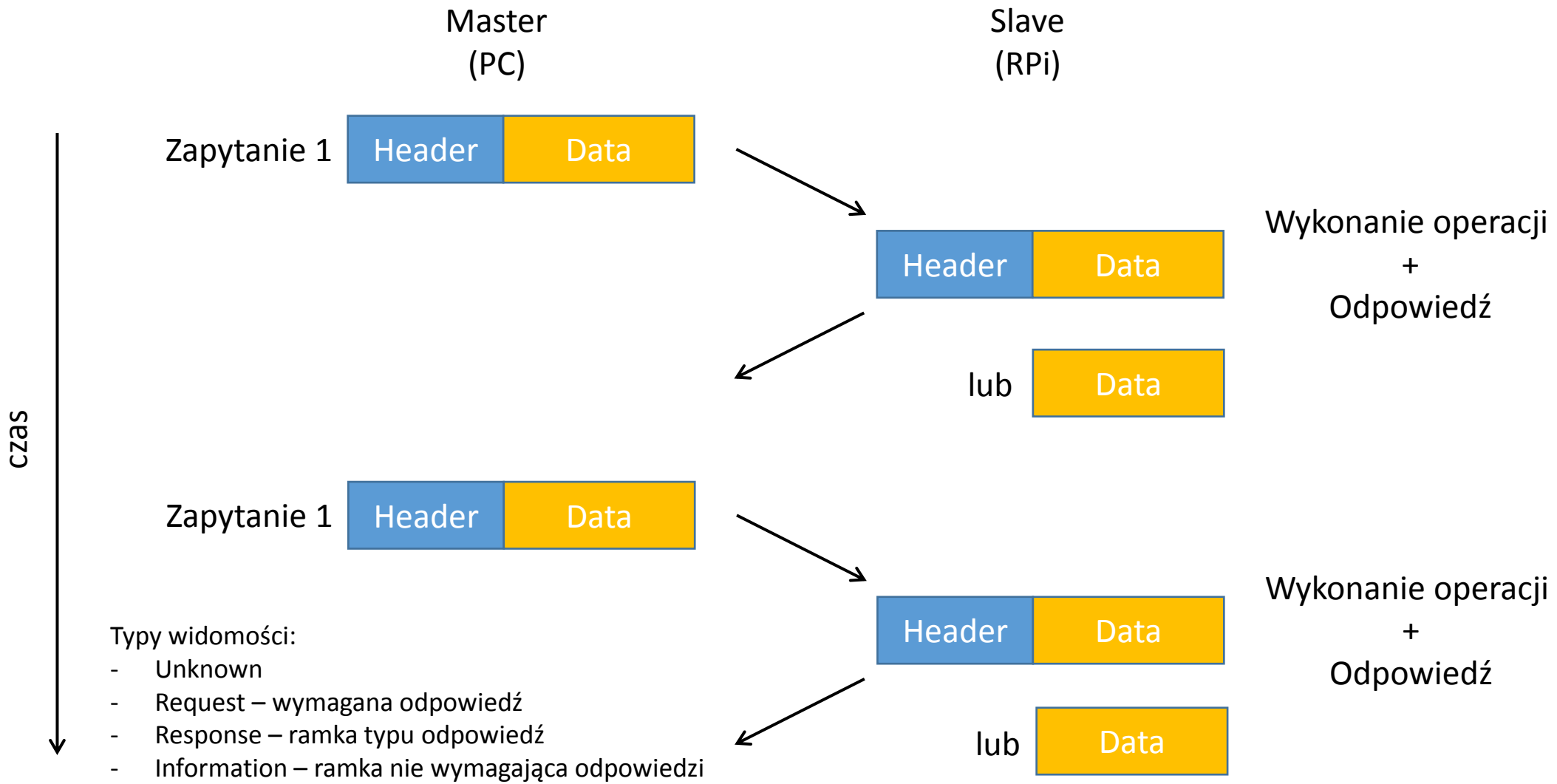
Budowa ramki



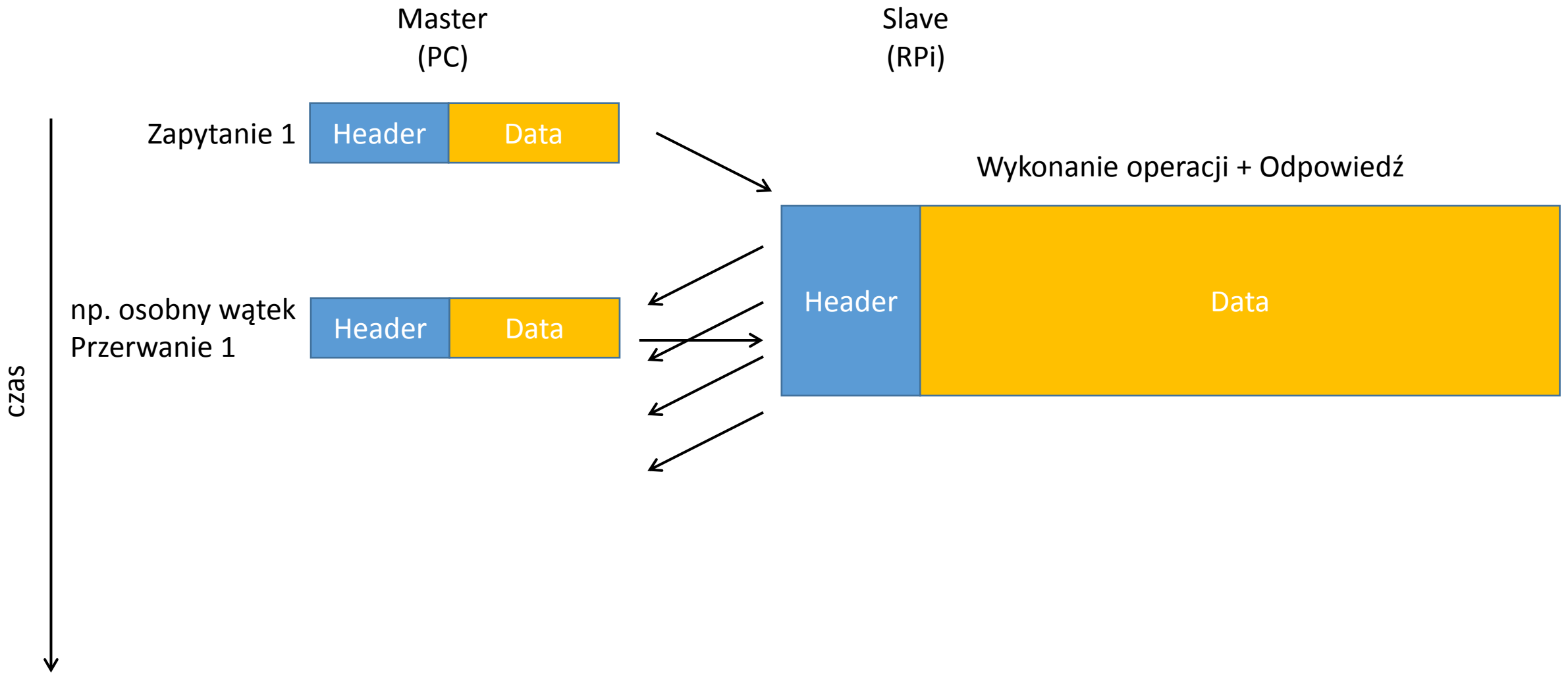
OSI



Master-Slave



Przerywanie długich operacji



Jednoczesne operacje

Problem:

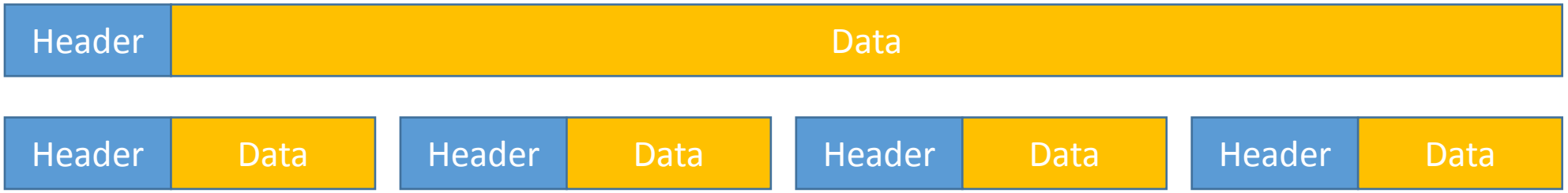
Wolne łącze, duże porcje danych lub długie operacje na serwerze mogą zmniejszyć „responsywność” naszej aplikacji – aplikacja będzie zamarzać na poziomie komunikacji będzie.

Rozwiązanie:

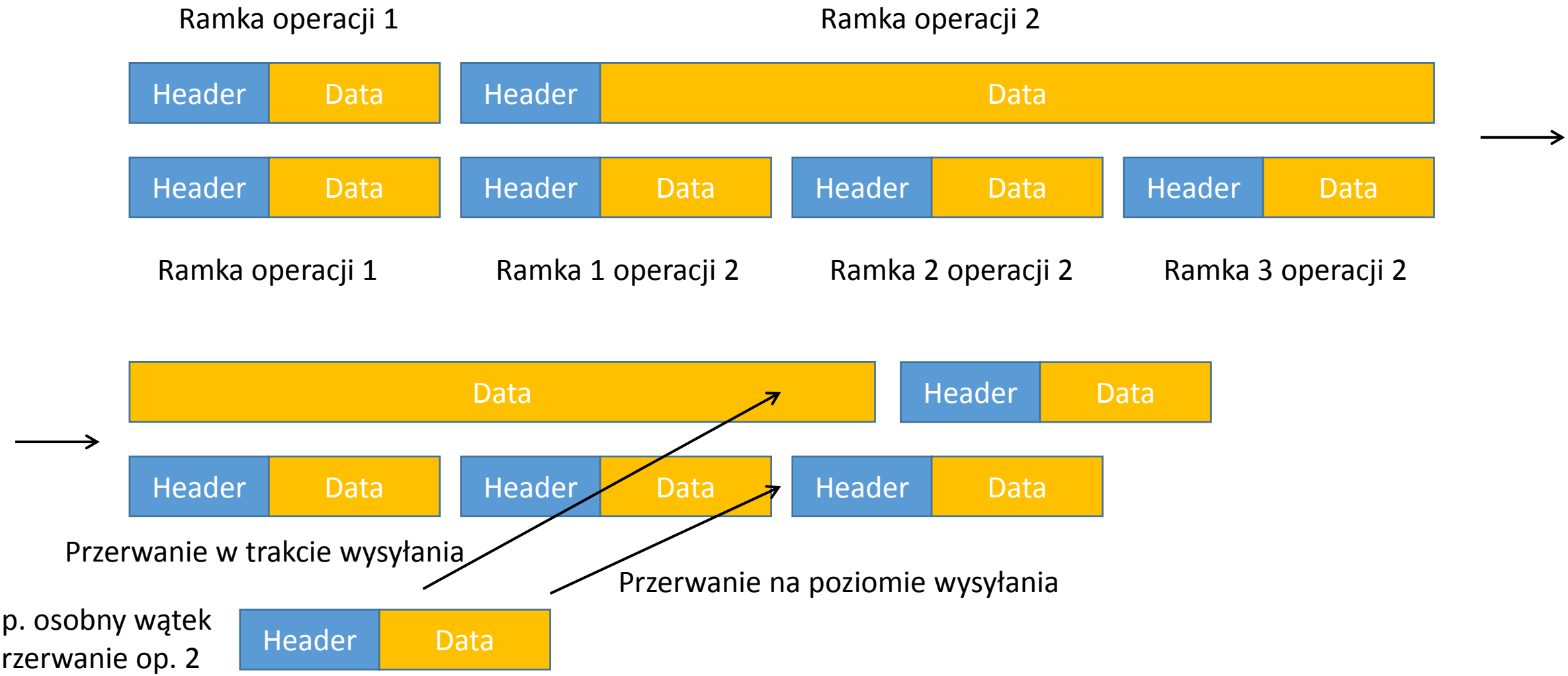
Podział danych na mniejsze ramki. Odbiorca nasłuchuje danych i w zależności od potrzeby podejmuje kroki.

Takie podejście redukuje ilość wymaganych wątków w trakcie działanie aplikacji.

Najprostsza wersja to jeden typ operacji w jednym czasie (Jedna kamera, Jeden odczyt diody, itp.).



Przerywanie jednoczesnych operacji



Forward'owanie ramek

Nie opłaca się za każdym razem implementować obsługi nowych typów ramek jeśli trzeba je tylko przekazać dalej.

Serwer nie zawsze wnika w naturę ramek więc potrzebna jest dodatkowa informacja o rozmiarze ramki.

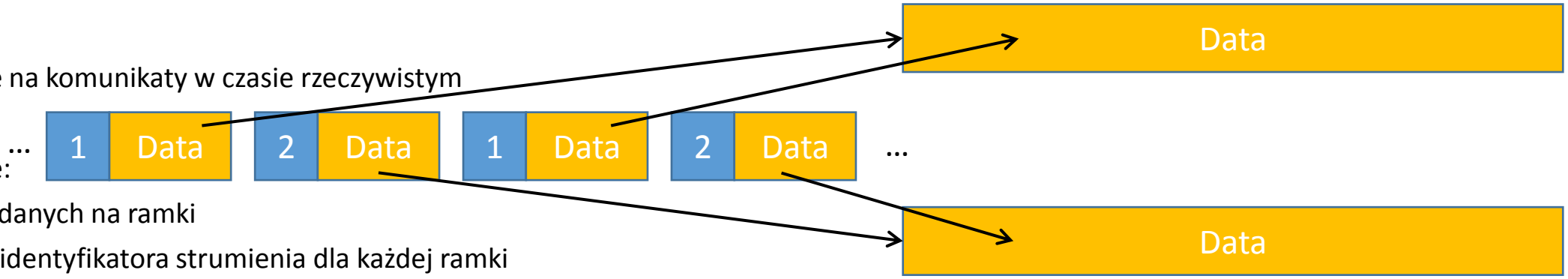
Wiele strumieni

Przydatne w przypadku wielu usług na jednym porcie (uniwersalne rozwiązanie).

1. Komunikacja z kamerą 1
2. Komunikacja sterowania urządzeniami

Problem:
Reagowanie na komunikaty w czasie rzeczywistym

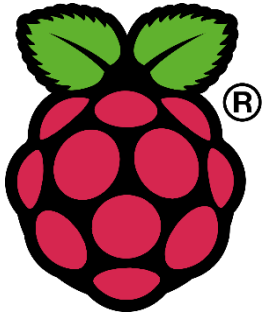
Rozwiązanie:



- Pocięcie danych na ramki
- Dodanie identyfikatora strumienia dla każdej ramki

```
class StreamReader // rozbija dane na strumienie
{
        int registerStream(); // dodaje nowy strumień
        bool deregisterStream(int number); // usuwa wskazany strumień
        bool addRawData(uint8_t *data, int length); // dodaje dane z socket'u
        int readStreamData(int streamNumer, uint8_t *data, int length); // wydobywa dane dla konkretnego strumienia
};
```

Typy ramek/operacji - numeracja



Najlepszym rozwiązaniem jest stworzenie wspólnego zestawu typów wyliczeniowych. Wspólny projekt (Common) spięty z klientem i serwerem zagwarantuje brak pomyłek w przypadku pojawienia się nowych typów ramek.

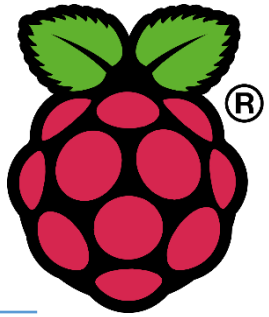
Ramka:

1. Nadawca
2. Rozmiar // opcjonalnie
3. **Typ ramki**
4. Dane

```
Enum FrameType
{
        Unknown = 0,
        FT_Camera,
        FT_Temperature,
        FT_Led
};
```

```
Enum LedOperationType
{
        Unknown = 0,
        LOT_Enable,
        LOT_Disable,
        LOT_SetInMode,
        LOT_SetOutMode,
        LOT_SetHighValue,
        LOT_SetLowValue
};
```

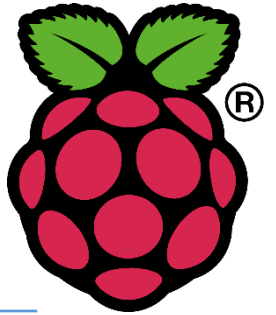
Typy ramek – scenariusz



```
Enum FrameType
{
        FT_Unknown = 0,
        FT_Camera,
        FT_Temperature,
        FT_Led,
        FT_Count
};

uint8_t frameType = socket.readByte();
switch(frameType)
{
   case FT_Camera:
      // operacje…
      break;
   case FT_Temperature:
      // operacja:
      // double temperature = 25.4;
      // uint8_t *frame = (uint8_t *)&temperature;
      // client.write(frame, 8);
      break;
   case FT_Led:
      // operacje…
      break;
}
```

```
typedef bool (*Function)(uint8_t *data, int length);

bool doCamera (Socket socket){ /* logika*/ }
bool doTemperature (Socket socket){ /* logika*/ }
bool doLed (Socket socket){ /* logika*/ }

Function functions[FT_Count];

functions[FT_Unknown] = NULL;
functions[FT_Camera] = doCamera;
functions[FT_Temperature] = doTemperature;
functions[FT_Led] = doLed;

uint8_t frameType = socket.readByte();
if(frameType < FT_Count)
{
   Function function = functions[frameType];
   if(function != NULL)
      function(socket);
}
```

Kamera - operacje

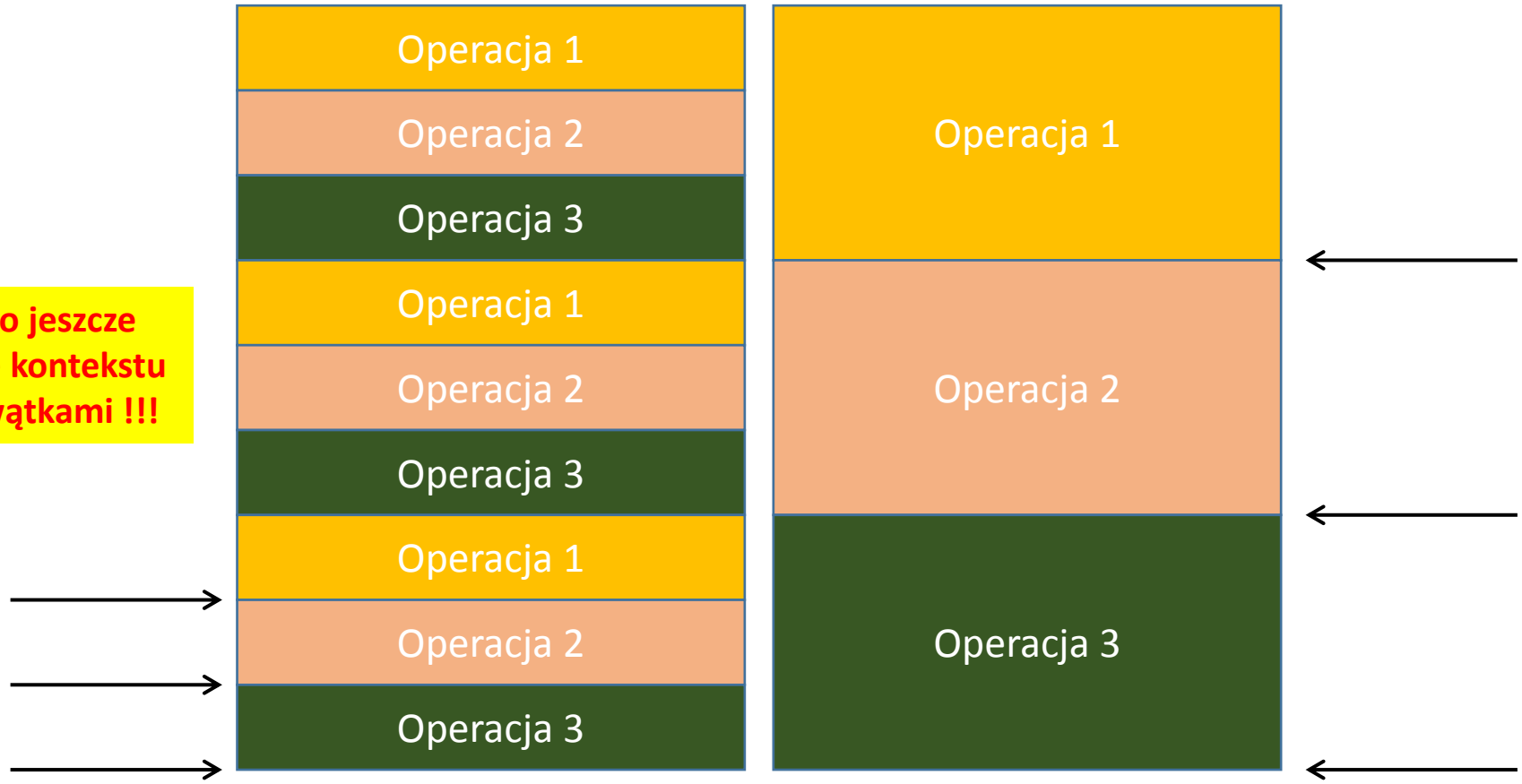


```
Enum CameraOperationType
{
        COT_Unknown = 0,
        COT_Enable,
        COT_Disable,
        COT_Capture
};
```

```
Rpi camera;

bool doCamera(Socket socket)
{
uint8_t operationType = socket.readByte();
switch(frameType)
{
   case COT_Enable:
      camera.open();
      break;
 case COT_Disable:
      camera.close();
      break;
 case COT_Capture:
      //TODO: sprawdzenie warunków
      uint32_t length = 1920 * 1080 * 3; // 24 bit RGB
      uint8_t *data = new uint8_t[length];
      camera.capture(data);
      socket.writeUInt32(length);
      socket.write(data);
      break;
}
```

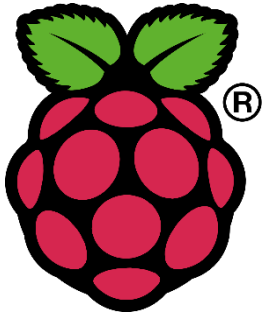
Ilość wątków a efektywność

Operacja sekwencyjna a wielu użytkowników (wiele operacji)



Dodatkowo jeszcze przełączanie kontekstu pomiędzy wątkami !!!

Komunikacja asynchroniczna

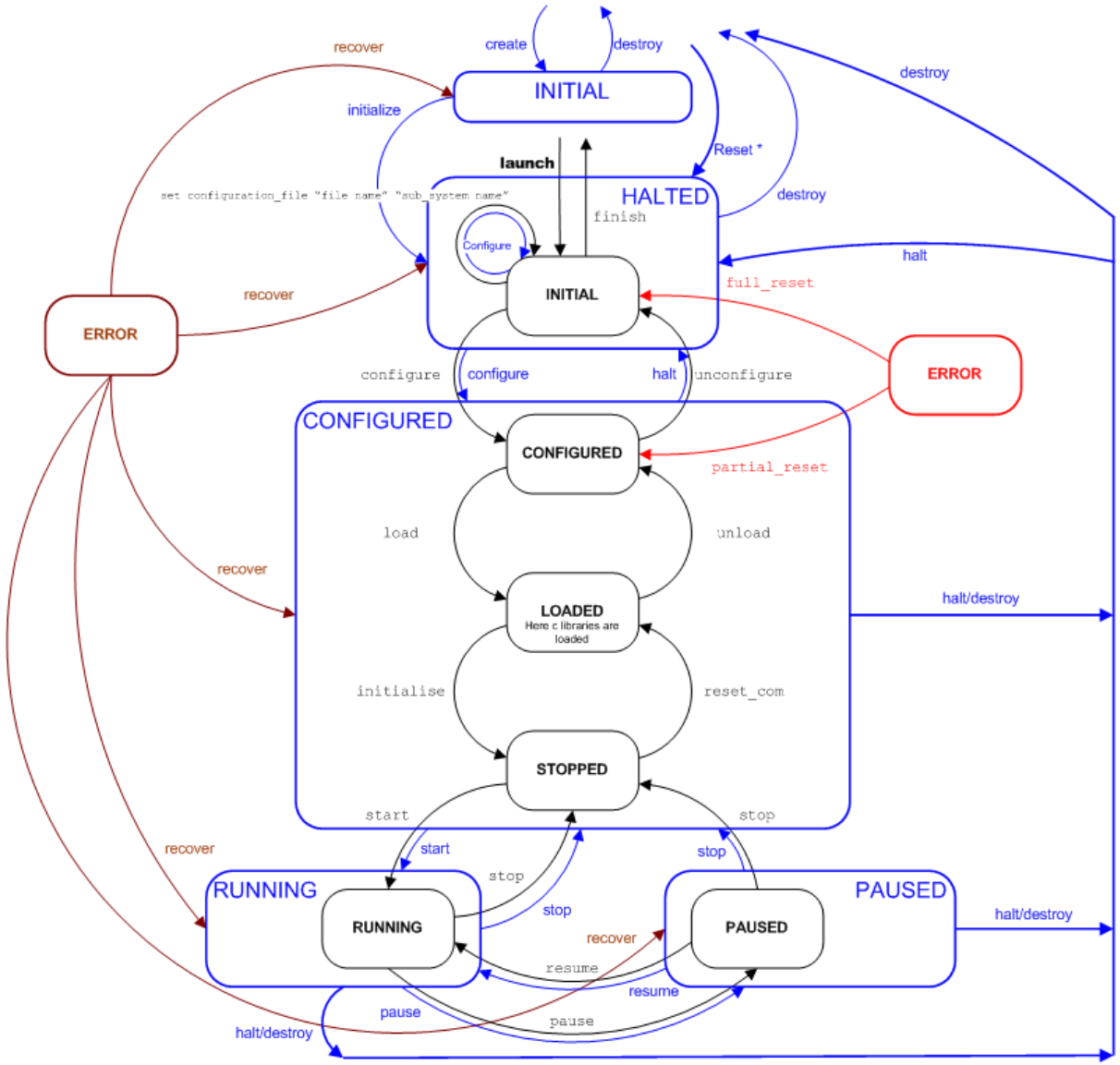


Wiele różnych operacji w jednym czasie.

- Wymagane jest wprowadzenie identyfikatora dla każdej operacji
- Podczas wykonywania operacji można zapisywać lokalnie informacje o oczekiwaniu na odpowiedź z wskaźnikiem do funkcji która ma się wykonać po dotarciu odpowiedzi.
- Potrzebny jest status/stan operacji (jedna operacja może wymagać przesłania 20 ramek)

Warto wykorzystać nieblokujące socket’y

Przykładowa maszyna stanowa

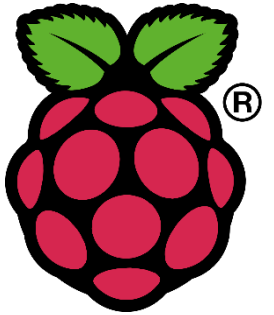


State Machines

- Narval
- Narval errors
- Function Manager
- FM Errors

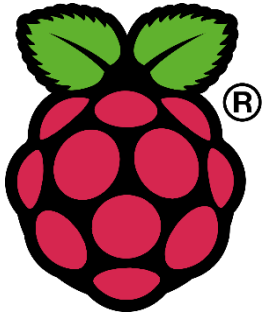
* Reset command can be sent from all the FM States but "INITIAL"

Maszyny stanowe - asynchroniczność



1. Instrukcje warunkowe oraz zmienne opisujące każdy ze stanów
   Wady:
   - brak enkapsulacji pewnych danych
   - przy złożonych scenariuszach wymagane jest wykonanie dużej ilości warunków
   - przy większych projektach:
     1. może prowadzić do dużej ilości błędów
     2. generowanie dużej ilości warunków
     3. dużo zmian w kodzie podczas zmian scenariusza
2. Wskaźniki do aktywnych funkcji
   - wymagany jest zestaw funkcji wykonywanych podczas pewnych scenariuszy
   - można uzyskać złożoność dla dostępu do konkretnych stanów na poziomie N(1)
   - łatwość implementacji nawet bardzo złożonych maszyn stanowych
   - funkcje mogą pracować na specyficznych obiektach przekazywanych jako argument wywołania funkcji, np. konkretny użytkownik może mieć własny obiekt przechowujący jego stan
3. Klasy dziedziczące po wspólnej abstrakcyjnej klasie
   - własne obiekty dla każdego użytkownika w argumencie wywoływanej funkcji,
   - możliwość ukrywania funkcjonalności związanej z pewnym stanem
   - łatwość implementacji nawet bardzo złożonych maszyn stanowych
   - bardzo mała złożoność dostępu do konkretnego stanu (nawet na poziomie N(1))

Implementacja na klasach



Praktyczny przykład