



**AGH**

**AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W  
KRAKOWIE**

**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI,  
INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ**

KATEDRA INFORMATYKI STOSOWANEJ

Praca dyplomowa inżynierska

*Obliczenia rozproszone w języku Haskell*  
*Distributing tasks with Haskell*

Autor:	<i>Konrad Lewandowski</i>
Kierunek studiów:	<i>Informatyka</i>
Opiekun pracy:	<i>dr inż. Piotr Matyasik</i>

Kraków, 2018

*Upředzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształca taki utwór; artystyczne wykonanie, fonogram, wideogram lub nadanie.”, a także upředzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.): „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej «sądem koleżeńskim».”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.*

*Serdecznie dziękuję ...tu ciąg dalszych  
podziękowań np. dla promotora, żony, są-  
siada itp.*



# Spis treści

<b>1. Wprowadzenie .....</b>	<b>7</b>
1.1. Cel i założenia pracy.....	7
1.2. Istniejące rozwiązania .....	7
1.2.1. Celery .....	7
1.2.2. Resque .....	7
1.2.3. Cloud Haskell .....	8
<b>2. Implementacja rozpraszania zadań .....</b>	<b>9</b>
2.1. Broker RabbitMQ .....	9
2.2. Zarządzanie zasobami .....	10
2.3. Środowisko funkcji .....	12
2.4. Odczyt konfiguracji z pliku .....	14
2.5. Obsługa zadań.....	15
2.6. Raportowanie postępów wykonania .....	17
2.7. Schemat komunikacji z brokerem .....	18
2.8. Przerywanie zadań.....	19
2.9. Klient .....	20
<b>3. Podsumowanie .....</b>	<b>23</b>

## Listingi kodu

2.1	Łączenie z RabbitMQ . . . . .	10
2.2	Regionalizacja zasobów . . . . .	11
2.3	Problem funkcji asynchronicznych . . . . .	11
2.4	Typ reader . . . . .	12
2.5	Transformator typu Reader . . . . .	13
2.6	Przykładowy plik konfiguracyjny . . . . .	14
2.7	Odczyt konfiguracji . . . . .	14
2.8	Schemat obsługi zadania . . . . .	15
2.9	Problem typu zadania zdefiniowanego egzystencjalnie . . . . .	16
2.10	Ostateczna implementacja zadania . . . . .	17
2.11	Klient uruchamiający zadania . . . . .	20

# 1. Wprowadzenie

W czasach kiedy przewidywania Gordona Moore'a dotyczące dalszego wzrostu mocy obliczeniowej pojedynczych komputerów przestają się sprawdzać, coraz częściej wykorzystujemy metody wykonywania programów oparte na jednoczesnym przetwarzaniu rozproszonym na wielu połączonych ze sobą maszynach.

Pomimo pozornej prostoty takiego rozwiązania istnieje bardzo niewiele narzędzi ułatwiających programowanie w modelu rozproszonym.

## 1.1. Cel i założenia pracy

Celem poniższej pracy jest implementacja w języku Haskell podstawowej biblioteki do rozpraszania zadań na wielu komputerach z wykorzystaniem brokera RabbitMQ, umożliwiającą zlecanie zadań do wykonania, przerywanie zadań będących w trakcie wykonywania, raportowanie na bieżąco postępów wykonania, przesyłania wyników oraz wprowadzania zależności pomiędzy zadaniami.

## 1.2. Istniejące rozwiązania

Poniższa lista zawiera skrócony opis istniejących bibliotek programistycznych, służących do obliczeń rozproszonych:

### 1.2.1. Celery

**Celery** jest asynchroniczną kolejką zadań opartą o rozproszone komunikaty przesyłane między komputerami, napisaną w języku Python. Działa w czasie rzeczywistym, jednak umożliwia również szeregowanie zadań. Interfejs programistyczny pozwala na zlecanie zadań zarówno w sposób synchroniczny jak i asynchroniczny, oraz przesyłanie wyników. Wspiera wiele brokerów wiadomości (np. RabbitMQ, Redis, MongoDB).

### 1.2.2. Resque

**Resque** jest biblioteką języka Ruby, wykorzystującą bazę Redis w charakterze brokera wiadomości. Umożliwia asynchroniczne zlecanie powtarzalnych zadań na innych komputerach.

Biblioteka ta jest często używana przez programistów serwisów internetowych do wykonywania długotrwałych operacji (np. generowanie miniaturk zdjęć, rozsyłanie newslettera e-mail, tworzenie raportów, etc...)

### 1.2.3. Cloud Haskell

**Cloud Haskell** to biblioteka języka Haskell, udostępniająca warstwę transportową do komunikacji między węzłami, mechanizm serializacji domknięć pozwalający na zdalne uruchamianie procesów oraz API do programowania rozproszonego.



## 2. Implementacja rozpraszania zadań

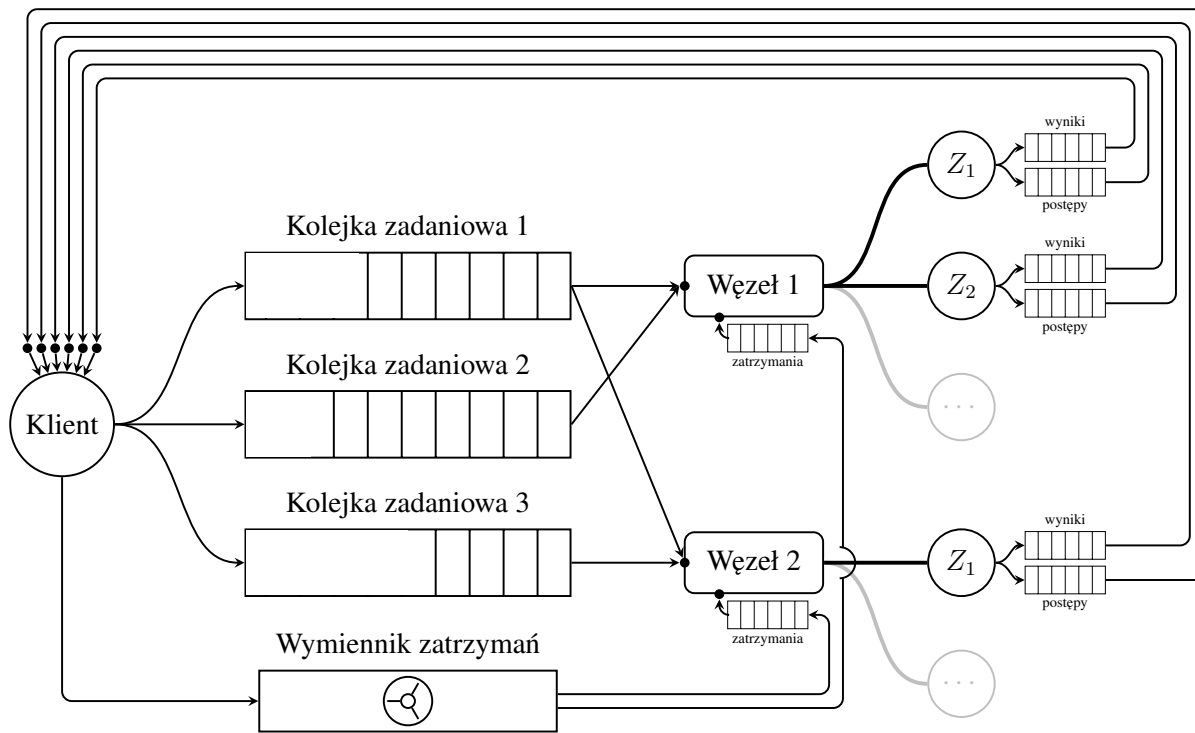
Poniższy rozdział został poświęcony szczególnie istotnym aspektom implementacyjnym biblioteki umożliwiającej uruchamianie zadań na wielu różnych komputerach w języku Haskell, ze szczególnym uwzględnieniem dodatkowych bibliotek i rozszerzeń tego języka.

### 2.1. Broker RabbitMQ

**RabbitMQ** to otwartoźródłowy broker wiadomości - oprogramowanie zapewniające odporny na zakłócenia mechanizm komunikacji sieciowej. Został zaimplementowany w języku Erlang z wykorzystaniem Open Telecom Platform. Posiada biblioteki dla wszystkich znaczących języków programowania - w tym dla Haskella (biblioteka `amqp` - od ang. Advanced Message Queuing Protocol). Dwie najważniejsze koncepcje implementowane przez RabbitMQ to - **kolejki** (ang. queues) oraz **wymienniki** (ang. exchanges). Kolejki to przetwarzane przez serwer struktury FIFO z sieciowym protokołem dostępu. Są najbardziej podstawową abstrakcją stosowaną w rozproszonym programowaniu współbieżnym, a w przypadku projektu będącego przedmiotem tej pracy pozwalają wielu węzłom "konkurować" o zadania w bezpieczny sposób (nigdy nie dojdzie do sytuacji, w której dwa zadania są w tym samym czasie wykonywane na dwóch różnych węzłach). Wymienniki są pośrednikami dostępu do zbiorów kolejek - wiadomość, która zostanie dodana do wymiennika może zostać powielona i rozdystrybuowana do połączonych z wymiennikiem kolejek w jeden z czterech możliwych sposobów zależnie od predefiniowanego typu wymiennika:

- direct** – wiadomość trafia tylko do kolejek z określonym kluczem routującym, identycznym z kluczem zawartym w nagłówku wiadomości
- fanout** – wiadomość trafia do wszystkich podłączonych kolejek
- topic** – wiadomość trafia do konkretnej kolejki tylko wtedy, kiedy jej klucz routujący spełnia wzorzec określony dla tej kolejki
- headers** – wiadomość trafia do konkretnej kolejki na podstawie innych niż klucz routujący parametrów zawartych w nagłówku

Topologia wykorzystywana w projekcie zakłada, że każdy węzeł może nasłuchiwać więcej niż jednej kolejki zadaniowej (jest to porządane w przypadku kiedy nie każdy węzeł jest w stanie



Rys. 2.1. Topologia kolejkowania

obsłużyć każdy typ zadania). Dodatkowo każdy węzeł tworzy własną kolejkę na zlecenia przerwania zadania i przypina ją do wspólnego wymiennika typu `fanout`. Gwarantuje to dostarczenie każdemu węzłowi komunikatu przerwania niezależnie od awarii komunikacji z brokerem:

## 2.2. Zarządzanie zasobami

Protokół AMQP operuje na zależnych od siebie zasobach, którymi należy w poprawny sposób zarządzać. Postawowy przykład z dokumentacji biblioteki AMQP w przypadku wystąpienia wyjątku nie gwarantuje zwolnienia zasobu, a kolejność wykonywania finalizatorów ze względu na leniwą ewaluację jest determinowana wyłącznie wyzwoleniem mechanizmu odśmiecania.

```

1 main = do
2   conn <- openConnection "127.0.0.1" "/" "guest" "guest"
3   ...
4   closeConnection conn

```

Listing 2.1. Łączenie z RabbitMQ

Jednym z istniejących rozwiązań tego problemu jest biblioteka `io-region`[1], umożliwiająca podział kodu na regiony odpowiedzialne za poszczególne zasoby oraz przenoszenie tych odpowiedzialności pomiędzy regionami

```

1 ...
2 region $ \r -> do
3   connection <- alloc_ r (AMQP.openConnection host vhost username password)
4                       AMQP.closeConnection
5   -- po opuszczeniu regionu następuje zwolnienie zasobów
6 ...

```

**Listing 2.2.** Regionalizacja zasobów

Rejestrowanie zasobów w obrębie odpowiednich regionów rozwiązuje również problem asynchronicznych wywołań zwrotnych, służących do obsługi nadchodzących komunikatów z brokera. Dla porównania, niewłaściwe rozwiązanie oparte o mechanizm **bracket** powoduje przedwczesne zamknięcie zasobu nadal wykorzystywanego przez funkcję uruchamianą asynchronicznie.

```

1 ...
2 bracket openConnection closeConnection $ \connection ->
3   bracket (openChannel connection) closeChannel $ \channel1 ->
4     consumeMsgs channel1 queue callback -- wywołanie asynchroniczne
5     -- nieporządkane zamknięcie kanału
6     -- otwarcie drugiego kanału
7   bracket (openChannel connection) closeChannel $ \channel2 ->
8     consumeMsgs channel2 queue callback -- wywołanie asynchroniczne
9     -- nieporządkane zamknięcie kanału
10  _ <- getLine -- oczekiwanie
11  -- zamknięcie połączenia
12
13 ...
14 region $ \r -> do
15   connection <- alloc_ r openConnection closeConnection
16   channel1 <- alloc_ r (openChannel connection) closeChannel
17   consumeMsgs channel1 queue callback
18   channel2 <- alloc_ r (openChannel connection) closeChannel
19   onsumeMsgs channel2 queue callback
20
21  _ <- getLine -- oczekiwanie
22  -- zwolnienie zasobów w poprawnej kolejności

```

**Listing 2.3.** Problem funkcji asynchronicznych

## 2.3. Środowisko funkcji

Większość funkcji związanych z protokołem AMQP wymaga do działania przekazania pewnego zasobu (jak na przykład obiekt `Connection` lub `Channel`). Robienie tego za każdym razem *explicite* prowadzi do zmniejszenia czytelności kodu. Idiomatycznym dla języka Haskell rozwiązaniem jest zastosowanie typu `Reader` [2]:

```

1 newtype Reader e a = Reader { runReader :: e -> a }
2
3 instance Functor (Reader e) where
4   fmap f r = Reader $ \e -> f (runReader r e)
5
6 instance Applicative (Reader e) where
7   pure a     = Reader $ \e -> a
8   ra <*> rb = Reader $ \e -> (runReader ra e) (runReader rb e)
9
10 instance Monad (Reader e) where
11   (Reader r) >>= f = Reader $ \e -> runReader (f (r e)) e
12
13 ask :: Reader a a
14 ask = Reader id

```

Listing 2.4. Typ reader

Typ `Reader` opakowuje funkcję przyjmującą jako argument środowisko jej wykonania i zwracającą pewien wynik obliczeń wykorzystujących to środowisko. Przykładowo dla:

```

1 testReader :: Reader Bool String
2 testReader = Reader $ \flag -> if flag then "Włącz" else "Wyłącz"
3
4 runReader testReader True == "Włącz"

```

Jednak istotą działania `Reader`'a jest jego monadyczny interfejs umożliwiający zapisanie funkcji `testReader` jako:

```

1 testReader :: Reader Bool String
2 testReader = do
3   flag <- ask
4   return $ if flag then "Włącz" else "Wyłącz"

```

Dzięki temu unikamy przekazywania tego samego środowiska za każdym razem jako argumentu funkcji, co jest szczególnie istotne w przypadku kiedy jest to wiele funkcji.

Niestety funkcjonalność samej monady `Reader` `reader` nie jest wystarczająca ze względu na mnogość operacji wykorzystujących operacje wejścia-wyjścia, więc wymagających typu `IO` do działania. Pomocne tutaj okazują się transformatory monad (ang. *Monad transformers*[3]):

```

1 newtype ReaderT e m a = ReaderT { runReaderT :: e -> m a }
2
3 instance Functor m => Functor (ReaderT e m) where
4   fmap f r = ReaderT $ \e -> fmap f (runReaderT r e)
5
6 instance Applicative m => Applicative (ReaderT e m) where
7   pure a    = ReaderT $ \e -> pure a
8   ra <*> rb = ReaderT $ \e -> (runReaderT ra e) <*> (runReaderT rb e)
9
10 instance Monad m => Monad (ReaderT e m) where
11   r >=> f = ReaderT $ \e -> do
12     a <- runReaderT r e
13     runReaderT (f a) e
14
15 ask :: Monad m => ReaderT a m a
16 ask = ReaderT return
17
18 lift :: m a -> ReaderT e m a
19 lift m = ReaderT (const m)

```

Listing 2.5. Transformator typu Reader

Dzięki funkcji `lift` możemy niejako „podciągać” operacje wykonane w ramach innej monady do typu `ReaderT`:

```

1 testReader :: ReaderT Bool IO String
2 testReader = do
3   flag <- ask
4   lift $ if flag then putStrLn "Włącz"
5           else putStrLn "Wyłącz"
6   return "Wynik"
7
8 > runReaderT testReader True
9 Włącz

```

## 2.4. Odczyt konfiguracji z pliku

Biblioteka `configurator`[4] umożliwia odczyt plików konfiguracyjnych i zapewnia uproszczony interfejs dostępu do przechowywanych wewnątrz wartości parametrów, a w przypadku ich braku korzysta z wartości domyślnej zapisanej „na sztywno” w kodzie konfigurowanego programu:

```
1 taskell {
2   rabbitmq {
3     host      = "localhost"
4     vhost     = "/"
5     username  = "guest"
6     password  = "guest"
7   }
8   abortExchange = "taskell.abort"
9   parallelism = 1
10  queues = ["taskell.q1", "taskell.q2"]
11 }
```

Listing 2.6. Przykładowy plik konfiguracyjny

- taskell.rabbitmq.host** – Nazwa sieciowa lub adres IP brokera
- taskell.rabbitmq.vhost** – URL hosta wirtualnego jeśli na fizycznej maszynie działa więcej niż jeden broker
- taskell.rabbitmq.username** – Nazwa użytkownika skonfigurowana na brokerze
- taskell.rabbitmq.password** – Hasło użytkownika skonfigurowane na brokerze
- taskell.abortExchange** – Nazwa wymiennika do którego każdy węzeł przypina swoją kolejkę celem nasłuchu zleceń przerwania zadania
- taskell.queues** – Lista kolejek, które węzeł ma nasłuchiwać w oczekiwaniu na zadanie
- taskell.parallelism** – Liczba zadań, które węzeł może przetwarzać jednocześnie

```
1 main = do
2   [cp] <- getArgs
3   config <- load [ Required cp ]
4   host      <- lookupDefault "localhost" config "taskell.rabbitmq.host"
5   vhost     <- lookupDefault "/" config "taskell.rabbitmq.vhost"
6   username  <- lookupDefault "guest" config "taskell.rabbitmq.username"
7   password  <- lookupDefault "password" config "taskell.rabbitmq.password"
```

Listing 2.7. Odczyt konfiguracji

## 2.5. Obsługa zadań

Mechanizm uruchamiania konkretnego zadania sprowadza się do odczytu odpowiedniej funkcji z tablicy mieszającej zawierającej wszystkie obsługiwane przez węzeł zadania na podstawie klucza będącego wybraną przez programistę nazwą zadania, a następnie uruchomienie (po odpowiednim sparametryzowaniu) tej funkcji wewnątrz oddzielnego wątku. Ogólny schemat wygląda następująco:

```

1 newtype Task = Task { runTask :: ByteString -> IO ByteString }
2
3 registeredTasks :: HashMap Text Task
4 registeredTasks = fromList [ ("name1", function1)
5                             , ("name2", function2)
6                             , ("name3", function3) ]
7 ...
8
9 handleTask (msg, env) = do
10   let Just taskName = AMQP.msgType msg
11   let taskArgs = AMQP.msgBody msg
12
13   forkIO $
14     result <- runTask (registeredTasks ! taskName) taskArgs
15     ...
16
17   AMQP.ackEnv env

```

**Listing 2.8.** Schemat obsługi zadania

Nieprzypadkowo typ `Task` to opakowana, monomorficzna funkcja przetwarzająca ciągi bajtów. Gdyby pokusić się o przeniesienie odpowiedzialności za deserializację argumentów i serializację wyników obliczeń, typ ten przyjąłby pozornie „bezpieczniejszą” formę egzystencjalną:

$$\forall_{a,b}(\text{Serializable } a, \text{Serializable } b) \Rightarrow a \rightarrow b$$

jednak w takiej sytuacji kompilator nie może ustalić typów polimorficznych  $a, b$  w kontekście fragmentu kodu obliczającego wartość:

```

1 {-# LANGUAGE ExistentialQuantification #-}
2 {-# LANGUAGE RankNTypes                #-}
3 module Test where
4
5 import Data.Store
6 import Data.Text
7 import Data.ByteString
8 import Data.HashMap.Strict
9
10 newtype Task = Task { runTask :: forall a b . (Store a, Store b)
11                      => a -> IO b }
12
13 registeredTasks :: HashMap Text Task
14 registeredTasks = ...
15
16 runTaskByName :: Text -> ByteString -> IO ByteString
17 runTaskByName taskName encodedArg = do
18   arg <- decodeIO encodedArg
19   result <- runTask (registeredTasks ! taskName) arg
20   -- Błąd sprawdzania jednoznaczności typów
21   return $ encode result

```

**Listing 2.9.** Problem typu zadania zdefiniowanego egzystencjalnie

Błąd ten jest analogiczny do przedstawionego poniżej bardziej podstawowego przykładu i wychwytyuje go mechanizm sprawdzania jednoznaczności instancjonowanych typów:

```

1 show (read "5" :: Int)           => "5"
2 show (read "5" :: Double)        => "5.0"
3 show (read "5") => Ambiguous type variable 'a2' arising from a use of 'read'
4                             prevents the constraint '(Read a2)' from being solved.

```

Inferencja typu dla trzeciego wyrażenia przebiega w systemie Hindleya-Milnera[5] rozszerzonym o klasy typów[6] następująco:

$$\frac{\frac{\text{show} : \forall \alpha. (\text{Show } \alpha). \alpha \rightarrow \text{String} \in \Gamma}{\Gamma \vdash \text{show} : \forall \alpha. (\text{Show } \alpha). \alpha \rightarrow \text{String}} [\text{Var}]}{\frac{\frac{\frac{\langle 5 \rangle : \text{String} \in \Gamma}{\Gamma \vdash \langle 5 \rangle : \text{String}} [\text{Var}]}{\text{read } \langle 5 \rangle : \forall \alpha. (\text{Read } \alpha). \alpha} [\text{App}]}{\text{show}(\text{read } \langle 5 \rangle) : \forall \alpha. (\text{Read } \alpha, \text{Show } \alpha). \text{String}} [\text{Comb}]}$$

**Rys. 2.2.** Inferencja niejednoznacznego typu

Wynikowy typ zawiera restrykcje odnoszące się do zmiennej  $\alpha$  niewystępującej poza kontekstem, więc wyrażenie takie nie posiada prawidłowego typu[7].



## 2.6. Raportowanie postępów wykonania

Każde zadanie oprócz zwracania rezultatu może również raportować na bieżąco postęp wykonywanych operacji. Skuteczną metodą obsługi tego mechanizmu są współprogramy (ang. *coroutines*). Biblioteka `monad-coroutine`[8] dostarcza spójny i idiomatyczny interfejs pozwalający na zaimplementowanie tej funkcjonalności:

```

1 newtype Task arg p r = Task { runTask :: arg -> Coroutine (Yield p) IO r }
2
3 instance Functor (Task arg p) where
4   fmap f (Task t) = Task $ \arg -> fmap f (t arg)
5
6 instance Applicative (Task arg p) where
7   pure t = Task $ \_ -> pure t
8   Task t1 <*> Task t2 = Task $ \arg -> t1 arg <*> t2 arg
9
10 instance Monad (Task arg p) where
11   (Task t) >>= f = Task $ \arg -> do
12     t' <- t arg
13     runTask (f t') arg
14
15 type TaskStore = HashMap Text RawTask
16
17 runTaskByName :: MonadIO m => TaskStore -> Text
18               -> ByteString           -- Zserializowany argument zadania
19               -> (ByteString -> IO a) -- Domknięcie raportujące postępy
20               -> m ByteString         -- Wynik zadania
21 runTaskByName ts key arg reportFn = liftIO $ do
22   let producer = runTask (ts ! key) arg
23   pogoStick (\(Yield x cont) -> lift (reportFn x) >> cont) producer
24
25 progress :: Monad m => p -> Coroutine (Yield p) m ()
26 progress = yield

```

Listing 2.10. Ostateczna implementacja zadania

Przykładowe zadanie sumujące liczby i raportujące postęp po przetworzeniu każdego elementu listy wygląda wtedy następująco:

```

1 summationTask :: Task [Int] Int Int
2 summationTask = Task $ \numbers ->
3   let indexedNumbers = zip numbers [1..]
4       count = fromIntegral $ length numbers
5       sumf acc (a, nr) = do
6         progress $ floor $ (fromIntegral nr / count) * 100
7         return $ acc + a
8   in foldM sumf 0 indexedNumbers

```



Zatwierdzenie odebrania zadania odbywa się dopiero po odesłaniu wyniku, co pozwala na wykorzystanie wbudowanego w RabbitMQ mechanizmu automatycznego ponownego kolejkowania niedokończonych zadań, na wypadek np. fizycznej awarii węzła.

## 2.8. Przerywanie zadań

Przerywanie zadań jest obsługiwane poprzez zapamiętanie identyfikatora wątku zadania w transakcyjnej mapie[10], gdzie kluczami są identyfikatory aktualnie wykonywanych zadań, a wartościami identyfikatory wątku obsługującego zadanie. Komunikat przerzucania zadania zawiera identyfikator zadania, które ma zostać przerwane, co umożliwia przerwanie odpowiedniego wątku:

```
1 type CurrentTasks = STM.Map UUID ThreadId
2
3 abortHandler :: CurrentTasks -> ReaderT (Env (Message, Envelope)) IO ()
4 abortHandler currentTasks = do
5   Env r log (msg, env) <- ask
6   deferAck r env
7
8   let Just taskId = fromASCIIBytes $ BL.toStrict (msgBody msg)
9   log $ "Got abort for task " <> toText taskId
10
11   let strategy k = return (k, STM.Focus.Remove)
12   abortCurrent <- atomically $ STM.focus strategy taskId currentTasks
13   case abortCurrent of
14     Just threadId -> do
15       liftIO $ killThread threadId
16       log "Abort received, thread killed"
17     Nothing -> log "Not my task, skipping"
```

## 2.9. Klient

Uruchamianie zadań jest możliwe za pomocą funkcji `enqueueTask`, a oczekiwanie na wyniki i postępy umożliwiają odpowiednio funkcje `onResult` oraz `onProgress` zdefiniowane następująco:

```

1 enqueueTask :: (MonadIO m, MonadReader (Env Channel) m)
2               => T.Text -> T.Text -> BL.ByteString -> m UUID
3 enqueueTask qname taskName taskArgs = do
4   taskId <- liftIO nextRandom
5   queue newQueue { queueName = qname, queueDurable = False
6                   , queueExclusive = True } $ do
7     publish newMsg { msgBody = taskArgs
8                     , msgID = Just $ toText taskId
9                     , msgType = Just taskName
10                    , msgDeliveryMode = Just Persistent }
11   return taskId
12
13 onSuffix :: (MonadIO m, MonadReader (Env Channel) m)
14           => T.Text -> UUID -> (BL.ByteString -> ReaderT (Env Channel) IO ())
15           -> m ()
16 onSuffix suffix taskId callback = do
17   env <- ask
18   queue newQueue { queueName = toText taskId <> suffix
19                   , queueAutoDelete = True } $
20     subscribe Ack $ do
21       Env r _ (msg, envelope) <- ask
22       deferAck r envelope
23       liftIO $ runReaderT (callback $ msgBody msg) env
24   return ()
25
26 onProgress = onSuffix ".progress"
27 onResult = onSuffix ".result"

```

**Listing 2.11.** Klient uruchamiający zadania

Uruchomienie zadania polega na umieszczeniu na wybranej przez użytkownika kolejce komunikatu zlecającego zadanie, którego nasłuchują węzły. Kiedy węzeł odeśle postęp, lub rezultat, możliwe jest zlecenie kolejnego zadania, w szczególności wykorzystującego rezultat poprzedniego jako swój argument. Dzięki niskopoziomowemu interfejsowi możemy wykorzystywać wiele różnych strategii serializacji i deserializacji przesyłanych danych oraz eliminować ich izomorficzne przekształcenia (niepotrzebne transformacje danych będących w odpowiednich formatach). Kosztem takiego rozwiązania jest brak bezpieczeństwa typów.

```
1 {-# LANGUAGE ViewPatterns #-}
2
3 deserialize :: Store a => BL.ByteString -> a
4 deserialize = decodeEx . BL.toStrict
5
6 serialize :: Store a => a -> BL.ByteString
7 serialize = BL.fromStrict . encode
8
9 printL :: (MonadIO m, Show a) => a -> m ()
10 printL = liftIO . print
11
12 main :: IO ()
13 main = logger defaultLogger $ connection "localhost" "/" "guest" "guest" $ do
14   channel "task" $ do
15     task1 <- enqueueTask "taskell.q1" "additionTask"
16               $ serialize (1 :: Int, 2 :: Int)
17     task1 `onProgress` \(deserialize -> p) -> printL (p :: Int)
18     task1 `onResult` \r -> do
19       task2 <- enqueueTask "taskell.q1" "dummyTask" r
20       task2 `onResult` \(deserialize -> p) -> printL (p :: Int)
```



### **3. Podsumowanie**





# Bibliografia

- [1] Yuras Shumovich. *The io-region package*. 2015. URL: <https://hackage.haskell.org/package/io-region>.
- [2] Mark P. Jones. „Functional Programming with Overloading and Higher-Order Polymorphism”. W: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. London, UK, UK: Springer-Verlag, 1995, s. 97–136. ISBN: 3-540-59451-5.
- [3] Sheng Liang, Paul Hudak i Mark Jones. „Monad Transformers and Modular Interpreters”. W: *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: ACM, 1995, s. 333–343. ISBN: 0-89791-692-1. DOI: 10.1145/199448.199528.
- [4] Bryan O’Sullivan. *The configurator package*. 2014. URL: <https://hackage.haskell.org/package/configurator>.
- [5] Luis Damas i Robin Milner. „Principal Type-schemes for Functional Programs”. W: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: ACM, 1982, s. 207–212. ISBN: 0-89791-065-6. DOI: 10.1145/582153.582176.
- [6] P. Wadler i S. Blott. „How to Make Ad-hoc Polymorphism Less Ad Hoc”. W: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: ACM, 1989, s. 60–76. ISBN: 0-89791-294-2. DOI: 10.1145/75277.75283.
- [7] Simon Marlow. „Haskell 2010 Language Report”. W: (2010).
- [8] Mario Blazevic. *The monad-coroutine package*. 2016. URL: <https://hackage.haskell.org/package/monad-coroutine>.
- [9] Tim Harris, Simon Marlow i Simon Peyton Jones. „Composable memory transactions”. W: *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM Press, sty. 2005, s. 48–60. ISBN: 1-59593-080-9.
- [10] Nikita Volkov. *The stm-containers package*. 2014. URL: <https://hackage.haskell.org/package/stm-containers>.