Typed Protocol Pipelining

Marcin Szamotulski

3rd January 2022

https://coot.me/presentations/typed-protocol-pipelining.pdf

```
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE EmptyCase #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE StandaloneKindSignatures #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}

module Presentation.TypedProtocolPipelining where

import Data.Kind (Type)
import Data.Singletons
```

Ping Pong Protocol

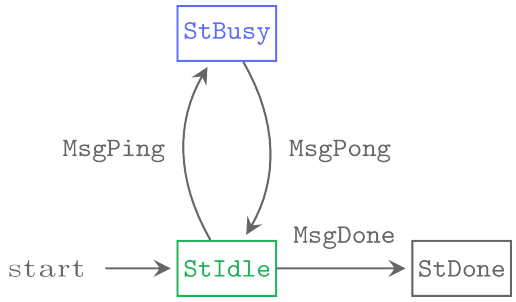


Figure: PingPong protocol state diagram

Protocol pipelining

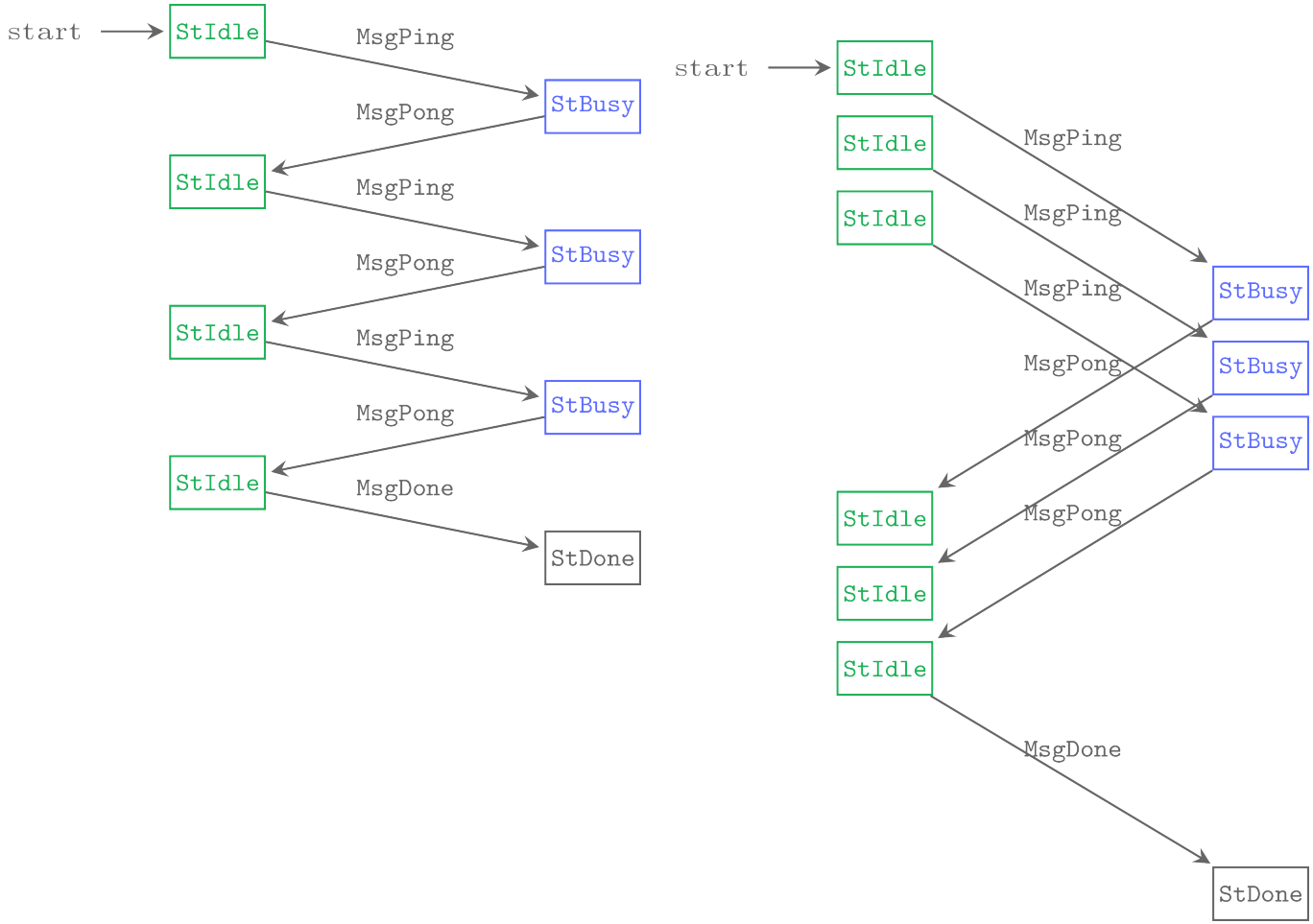


Figure: non-pipelined vs pipelined ping pong client

Protocol pipelining

- **latency hidding**
- **network utilisation** *network is utilised best when constant pressure is applied to avoid shrinking of the tcp window (e.g. tcp flow control mechanism). Network utilisation is a balance between keeping the most constrained resource busy - but only just - too busy and delay increases and application responsiveness can drop.*
- pipelined transitions are no longer a continous flow of matching transitions (i.e. composition of state transitions).
- pipelining must keep relative order of reqests and response transitions, but can mix both groups.

Towards non-pipelined protocol description

```
data PingPong where
  StIdle  ::PingPong
  StBusy ::PingPong
  StDone::PingPong

data MessageSimplePingPong (st::PingPong) (st'::PingPong) where
  MsgSimplePing ::MessageSimplePingPong StIdle StBusy
  MsgSimplePong::MessageSimplePingPong StBusy StIdle
  MsgSimpleDone::MessageSimplePingPong StIdle StDone

data SimplePingPongClient (st::PingPong) a where
  SendMsg    :: MessageSimplePingPong StIdle st
             →(SimplePingPongClient st a)
             →SimplePingPongClient StIdle a
  RecvMsg    :: (MessageSimplePingPong StBusy StIdle
                 →(SimplePingPongClient StIdle a))
             →SimplePingPongClient StBusy a
  ClientDone :: a
             →SimplePingPongClient StDone a
```

Towards non-pipelined protocol description

```
simplePingPongClient :: a → SimplePingPongClient StIdle a
simplePingPongClient a =
    SendMsg    MsgSimplePing
$ RecvMsg $ λ MsgSimplePong →
    SendMsg    MsgSimplePing
$ RecvMsg $ λ MsgSimplePong →
    SendMsg    MsgSimplePing
$ RecvMsg $ λ MsgSimplePong →
    SendMsg    MsgSimpleDone
$ ClientDone a
```

Towards pipelined protocol description

```
data N=Z|S N


data SimplePipelinedPingPongClient (st::PingPong) (n::N) c a where
  PipelinedSendMsg :: MessageSimplePingPong StIdle st
                   →PingPongReceiver                StBusy StIdle c
                   →SimplePipelinedPingPongClient StIdle (S n)    c a
                   →SimplePipelinedPingPongClient StIdle    n     c a
  CollectResponse  :: (c→SimplePipelinedPingPongClient StIdle    n   c a)
                   →SimplePipelinedPingPongClient      StIdle (S n) c a
  SendMsgDone      :: MessageSimplePingPong         StIdle   StDone
                   →SimplePipelinedPingPongClient StDone Z c a
                   →SimplePipelinedPingPongClient StIdle   Z c a
  PipelinedDone    :: a
                   →SimplePipelinedPingPongClient StDone Z c a

data PingPongReceiver (st ::PingPong)
                      (st'::PingPong) c where
  RecvPipelinedMsg :: (MessageSimplePingPong StBusy StIdle→c)
                   →PingPongReceiver StBusy StIdle c
```

Towards pipelined protocol description

```
simplePipelinedPingPongClient
  ::    a   -- fixed result, for simplicity
  →     c   -- fixed collected value, for simplicity
  →     SimplePipelinedPingPongClient StIdle Z c a
simplePipelinedPingPongClient a c=
  PipelinedSendMsg
    MsgSimplePing
    (RecvPipelinedMsg   $λMsgSimplePong→c)
    (PipelinedSendMsg
      MsgSimplePing
      (RecvPipelinedMsg$λMsgSimplePong→c)
      (       CollectResponse
      $λ_→CollectResponse
      $λ_→SendMsgDone MsgSimpleDone
      $       PipelinedDone a
      )
    )
```

Towards pipelined protocol description

Branching in *PipelinedSendMsg* requires that the interpretation of *SimplePipelinedPingPongClient* needs to concurrent execution:

```
PipelinedSendMsg :: MessageSimplePingPong StIdle st
                 → PingPongReceiver                 StBusy StIdle c
                 → SimplePipelinedPingPongClient StIdle (S n) c a
                 → SimplePipelinedPingPongClient StIdle    n  c a
```

Typed Protocol Description

Protocol Type Class

```
data Agency where
  ClientAgency  :: Agency
  ServerAgency  :: Agency
  NobodyAgency  :: Agency
```

Protocol type class provides messages and state type family.

```
class Protocol ps where
  data Message ps (st :: ps) (st' :: ps)
  type StateAgency (st :: ps) :: Agency

instance Protocol PingPong where
  data Message PingPong from to where
    MsgPing :: Message PingPong StIdle StBusy
    MsgPong :: Message PingPong StBusy StIdle
    MsgDone :: Message PingPong StIdle StDone

  type StateAgency StIdle  = ClientAgency
  type StateAgency StBusy  = ServerAgency
  type StateAgency StDone  = NobodyAgency
```

Typed Protocol Description

Relative Agency

```
data PeerRole = AsClient | AsServer


data RelativeAgency where
  WeHaveAgency     :: RelativeAgency
  TheyHaveAgency   :: RelativeAgency
  NobodyHasAgency  :: RelativeAgency

type Relative :: PeerRole -> Agency -> RelativeAgency
type family Relative pr a where
  Relative AsClient ClientAgency = WeHaveAgency
  Relative AsClient ServerAgency = TheyHaveAgency
  Relative AsClient NobodyAgency = NobodyHasAgency

  Relative AsServer ClientAgency = TheyHaveAgency
  Relative AsServer ServerAgency = WeHaveAgency
  Relative AsServer NobodyAgency = NobodyHasAgency
```

Typed Protocol Description

Relative Agency

Type equality for *RelativeAgency* which also carries information about agency.

```
type ReflRelativeAgency
  :: Agency → RelativeAgency → RelativeAgency → Type
data ReflRelativeAgency a r r' where
  ReflClientAgency  :: ReflRelativeAgency ClientAgency  r r
  ReflServerAgency  :: ReflRelativeAgency ServerAgency  r r
  ReflNobodyAgency  :: ReflRelativeAgency NobodyAgency  r r
```

An evidence that both relative agencies are equal to 'NobodyHasAgency'.

```
type ReflNobodyHasAgency
  :: RelativeAgency → RelativeAgency → Type
data ReflNobodyHasAgency ra ra' where
  ReflNobodyHasAgency :: ReflNobodyHasAgency
                           NobodyHasAgency
                           NobodyHasAgency
```

Typed Protocol Description

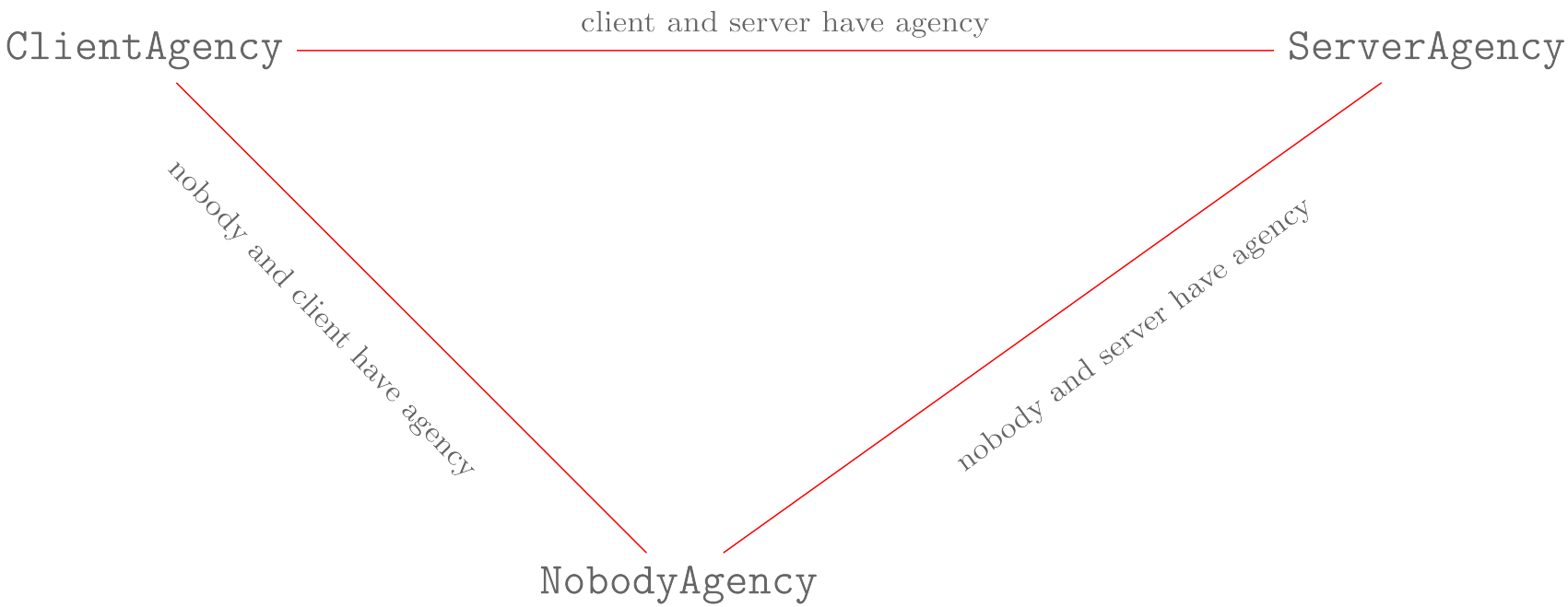
Relative Agency

A type family which swaps the client and server roles.

```
type FlipAgency::PeerRole→PeerRole
type family FlipAgency pr where
  FlipAgency AsClient=AsServer
  FlipAgency AsServer=AsClient
```

Exclusion Lemmas

Agency



Exclusion Lemmas

Relative Agency

A proof that if both *Relative pr a* and *Relative* (*FlipAgency pr*) *a* are equal then nobody has agency. In particual this lemma excludes the possibility that client and server has agency at the same state.

```
exclusionLemma_ClientAndServerHaveAgency
  :: ∀pr::PeerRole (a::Agency) (ra::RelativeAgency).
     SingPeerRole pr
  →ReflRelativeAgency a ra (Relative (            pr) a)
  →ReflRelativeAgency a ra (Relative (FlipAgency pr) a)
  →ReflNobodyHasAgency (Relative (            pr) a)
                       (Relative (FlipAgency  pr) a)
```

```
exclusionLemma_ClientAndServerHaveAgency
  SingAsClient ReflNobodyAgency ReflNobodyAgency
  =ReflNobodyHasAgency
```

```
exclusionLemma_ClientAndServerHaveAgency
  SingAsServer ReflNobodyAgency ReflNobodyAgency
  =ReflNobodyHasAgency
```

Exclusion Lemmas

Relative Agency

A proof that if one side has terminated, then the other side terminated as well.

```
terminationLemma_1
    :: SingPeerRole pr
    → ReflRelativeAgency a ra                  (Relative (        pr) a)
    → ReflRelativeAgency a NobodyHasAgency     (Relative (FlipAgency pr) a)
    → ReflNobodyHasAgency                      (Relative (        pr) a)
                                               (Relative (FlipAgency pr) a)
terminationLemma_1
  SingAsClient ReflNobodyAgency ReflNobodyAgency
    = ReflNobodyHasAgency
terminationLemma_1
  SingAsServer ReflNobodyAgency ReflNobodyAgency
    = ReflNobodyHasAgency
```

Exclusion Lemmas

Relative Agency

```
terminationLemma_2
    :: SingPeerRole pr
    → ReflRelativeAgency a ra                  (Relative (FlipAgency pr) a)
    → ReflRelativeAgency a NobodyHasAgency     (Relative (             pr) a)
    → ReflNobodyHasAgency                      (Relative (FlipAgency pr) a)
                                               (Relative (             pr) a)
terminationLemma_2
  SingAsClient ReflNobodyAgency ReflNobodyAgency
    = ReflNobodyHasAgency
terminationLemma_2
  SingAsServer ReflNobodyAgency ReflNobodyAgency
    = ReflNobodyHasAgency
```

Protocol Application API

singletons

```
data Trans ps where
  Tr::∀ps.ps→ps→Trans ps

data Queue ps where
  Empty::Queue ps
  Cons  ::Trans ps→Queue ps→Queue ps

type ( ◁ )  ::Trans ps→Queue ps→Queue ps
type a ◁ as=Cons a as
infixr 5 ◁

type ( ▷ )      ::Queue ps→Trans ps→Queue ps
type family as ▷ b where
  Empty       ▷ b=Cons b Empty
  (a ◁ as)    ▷ b=a ◁ (as ▷ b)
infixr 5 ▷
```

Protocol Application API

deep embedding: non-pipelined primitives

```
data Pipelined = NonPipelined | Pipelined
data Peer ps
   (pr :: PeerRole)
   (pl :: Pipelined)
   (q :: Queue ps)
   (st :: ps) m a where

   Effect :: m (Peer ps pr pl q st m a)
          -> Peer ps pr pl q st m a

   Done :: SingI st
          => (ReflRelativeAgency (StateAgency st)
                                 NobodyHasAgency
                                 (Relative pr (StateAgency st)))
          -> a
          -> Peer ps pr pl Empty st m a
```

Protocol Application API

deep embedding: non-pipelined primitives

Send a message (non-pipelined) and continue in a new state. Requires a proof that the sender has agency (*WeHaveAgency*).

```
Yield :: SingI st
      ⇒(ReflRelativeAgency (StateAgency st)
                           WeHaveAgency
                           (Relative pr (StateAgency st)))
      →Message ps st st′
      →Peer ps pr pl Empty st′ m a
      →Peer ps pr pl Empty st m a
```

Receive a message (non-pipelined), and continue at a new state. Requires an evidence that the remote side has agency (*TheyHaveAgency*).

```
Await :: SingI st
      ⇒(ReflRelativeAgency (StateAgency st)
                           TheyHaveAgency
                           (Relative pr (StateAgency st)))
      →(∀st′.Message ps st st′→Peer ps pr pl Empty st′ m a)
      →Peer ps pr pl Empty st m a
```

Protocol Application API

deep embedding: pipelined primitives

Pipeline a message, register the expected transition in the queue of suspended transitions, and continue possibly pipelining more messages.

$$
\begin{aligned}
&\mathit{YieldPipelined}\\
&\quad :: (\mathit{SingI}\ \mathit{st}, \mathit{SingI}\ \mathit{st}')\\
&\quad \Rightarrow (\mathit{ReflRelativeAgency}\ (\mathit{StateAgency}\ \mathit{st})\\
&\qquad\qquad\qquad\qquad\ \mathit{WeHaveAgency}\\
&\qquad\qquad\qquad\qquad\ (\mathit{Relative}\ \mathit{pr}\ (\mathit{StateAgency}\ \mathit{st})))\\
&\quad \rightarrow \mathit{Message}\ \mathit{ps}\ \mathit{st}\ \mathit{st}'\\
&\quad \rightarrow \mathit{Peer}\ \mathit{ps}\ \mathit{pr}\ \mathit{Pipelined}\ (q \triangleright \mathit{Tr}\ \mathit{st}'\ \mathit{st}'')\ \mathit{st}''\ \ m\ a\\
&\quad \rightarrow \mathit{Peer}\ \mathit{ps}\ \mathit{pr}\ \mathit{Pipelined}\ (q)\qquad\qquad\ \ \mathit{st}\quad m\ a
\end{aligned}
$$

Protocol Application API

deep embedding: pipelined primitives

Receive a message as part of suspended transition. It requires an evidence that the remote side has agency for the state st'.

```
Collect
  :: SingI st'
  ⇒(ReflRelativeAgency (StateAgency st')
                       TheyHaveAgency
                       (Relative pr (StateAgency st')))
  →Maybe (Peer ps pr Pipelined (Tr st'  st'' ◁ q) st m a)
  →(∀stNext.Message ps st' stNext
      →Peer ps pr Pipelined (Tr stNext st'' ◁ q) st m a)
  →Peer      ps pr Pipelined (Tr st'     st'' ◁ q) st m a
```

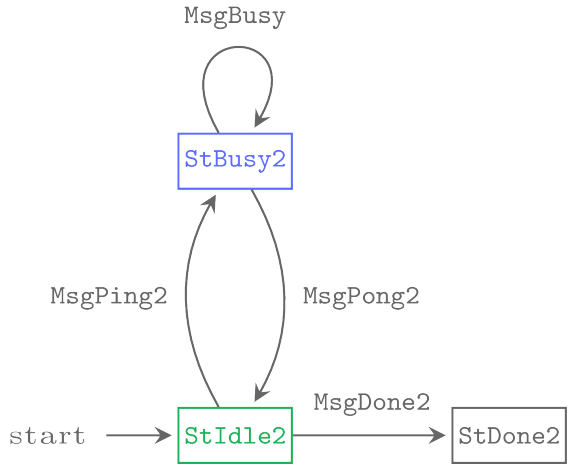
Eliminate an identity transition from the front of the queue.

```
CollectDone
  :: Peer ps pr Pipelined (          q) st m a
  →Peer ps pr Pipelined (Tr st st ◁ q) st m a
```

Pipelined Ping Pong client

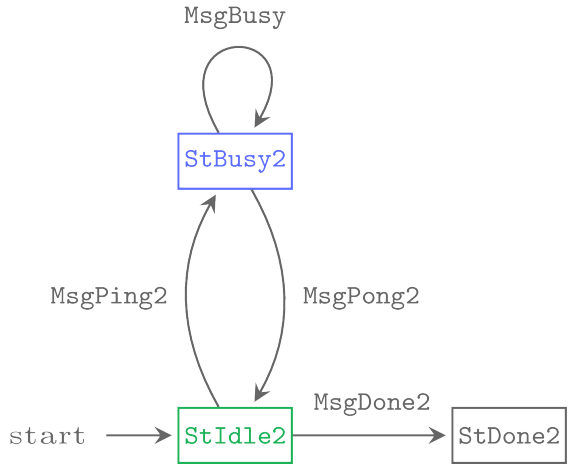
```
pingPongClientPipelined
  ::Peer PingPong AsClient Pipelined Empty StIdle m ()
pingPongClientPipelined
  =YieldPipelined ReflClientAgency MsgPing
  $ YieldPipelined ReflClientAgency MsgPing
  $ YieldPipelined ReflClientAgency MsgPing
  $ collect
  $ collect
  $ collect
  $ Yield ReflClientAgency MsgDone
  $ Done ReflNobodyAgency ()
where
  collect :: Peer PingPong AsClient Pipelined q StIdle m ()
          →Peer PingPong AsClient Pipelined
                (Tr StBusy StIdle ◁ q) StIdle m ()
  collect k
    =Collect ReflServerAgency Nothing
    $ λMsgPong→CollectDone k
```

Ping Pong v2



newtype *PingPong2* = *Wrap PingPong*
type *StIdle2* = *Wrap StIdle*
type *StBusy2* = *Wrap StBusy*
type *StDone2* = *Wrap StDone*

Ping Pong v2




```
instance Protocol PingPong2 where
  data Message PingPong2 from to where
    MsgPingPong
      :: Message PingPong  (        st)       (        st')
      -> Message PingPong2 (Wrap st)          (Wrap st')
    MsgBusy
      :: Message PingPong2 (Wrap StBusy) (Wrap StBusy)

  type StateAgency (Wrap StIdle)  = StateAgency StIdle
  type StateAgency (Wrap StBusy)  = StateAgency StBusy
  type StateAgency (Wrap StDone)  = StateAgency StDone
```

Pipelined Ping Pong v2 Client

```
pingPongClientPipeliend2
  ::Peer PingPong2 AsClient Pipelined Empty StIdle2 m Int
pingPongClientPipeliend2
  =YieldPipelined ReflClientAgency (MsgPingPong MsgPing)
  $ YieldPipelined ReflClientAgency (MsgPingPong MsgPing)
  $ YieldPipelined ReflClientAgency (MsgPingPong MsgPing)
  $          collect 0
  $ λn1→collect n1
  $ λn2→collect n2
  $ λn3→Yield ReflClientAgency (MsgPingPong MsgDone)
  $          Done ReflNobodyAgency n3
where
  collect :: Int
          →(Int→Peer PingPong2 AsClient Pipelined q StIdle2 m Int)
          →Peer PingPong2 AsClient Pipelined
                (Tr StBusy2 StIdle2 ◁ q) StIdle2 m Int
  collect ! n k
    =Collect ReflServerAgency Nothing
    $ λmsg→case msg of
      MsgBusy                    →collect (n + 1) k
      (MsgPingPong MsgPong)→CollectDone (k n)
```

Non-pipelined Duality

```
data TerminalStates ps (pr::PeerRole) where
  TerminalStates
    ::∀ps pr (st::ps) (st'::ps).
      Sing st
    →ReflRelativeAgency (StateAgency st)
                        NobodyHasAgency
                        (Relative (              pr) (StateAgency st))
    →Sing st'
    →ReflRelativeAgency (StateAgency st')
                        NobodyHasAgency
                        (Relative (FlipAgency pr) (StateAgency st'))
    →TerminalStates ps pr

theorem_nonpipelined_duality
  ::∀ps (pr::PeerRole) (initSt::ps) m a b.
    (Monad m, SingI pr)
  ⇒Peer ps (              pr) NonPipelined Empty initSt m a
  →Peer ps (FlipAgency pr) NonPipelined Empty initSt m b
  →m (a, b, TerminalStates ps pr)
```

Link to the proof. The proof relies on exclusion lemmas.

Removing pipelining

```
theorem_unpipeline
  ::∀ps (pr::PeerRole)
    (pl::Pipelined)
    (initSt::ps) m a.
    Functor m
  ⇒[Bool]
    -- interleaving choices for pipelining allowed by
    -- 'Collect' primitive. False values or '[]' give no
    -- pipelining.
  →Peer ps pr pl                Empty initSt m a
  →Peer ps pr NonPipelined Empty initSt m a
```

Link to the proof.

Pipelined Duality

```
theorem_duality
  :: ∀ps (pr::PeerRole)
         (pl::Pipelined)
         (pl'::Pipelined)
         (st::ps) m a b.
     (Monad m, SingI pr)
  ⇒[Bool]→[Bool]
  →Peer ps (                pr) pl  Empty st m a
  →Peer ps (FlipAgency pr) pl' Empty st m b
  →m (a, b, TerminalStates ps pr)

theorem_duality csA csB a b=
  theorem_nonpipelined_duality (theorem_unpipeline csA a)
                               (theorem_unpipeline csB b)
```

Remarks

- The duality theorem relies on 1-1 encoding of protocol messages; Non injective encodings can lead to deadlocks, or premature termination.
- Non injective encodings are useful! A protocol that handles simultaneous TCP open is an example.
- The presented *Peer* type was first discovered in *Agda*, and then re-implemented in Haskell. *Agda*'s more expressive type system, and quite similar syntax to Haskell, makes it ideal for type level experiments which as in this case can lead to simpler API.

Acknowledgement

- Alex Vieth, Well-Typed
- Duncan Coutts, Well-Typed

https://coot.me/presentations/typed-protocol-pipelinging.pdf

https://github.com/input-output-hk/ouroboros-network/tree/coot/typed-protocols-rewrite/typed-protocols