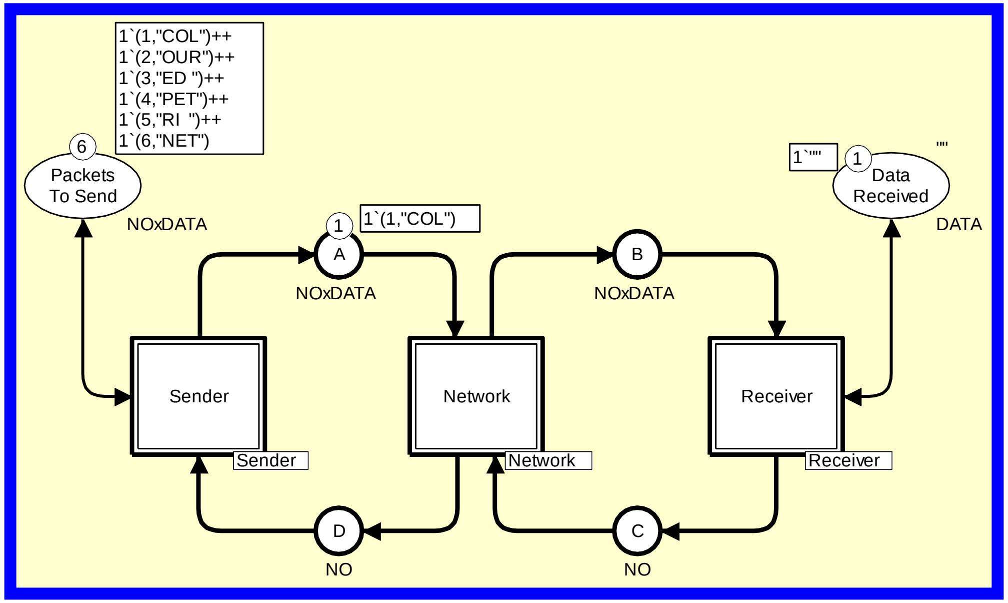
Coloured Petri Nets

Modelling and Validation of Concurrent Systems

Chapter 5: Hierarchical Coloured Petri Nets

Kurt Jensen &
Lars Michael Kristensen

{kjensen,lmkristensen}
@cs.au.dk



CPN models can be divided into modules

- CPN modules play a similar role as modules in ordinary programming languages.
- They allow the model to be split into manageable parts with well-defined interfaces.
- CP-nets with modules are also called hierarchical Coloured Petri Nets.

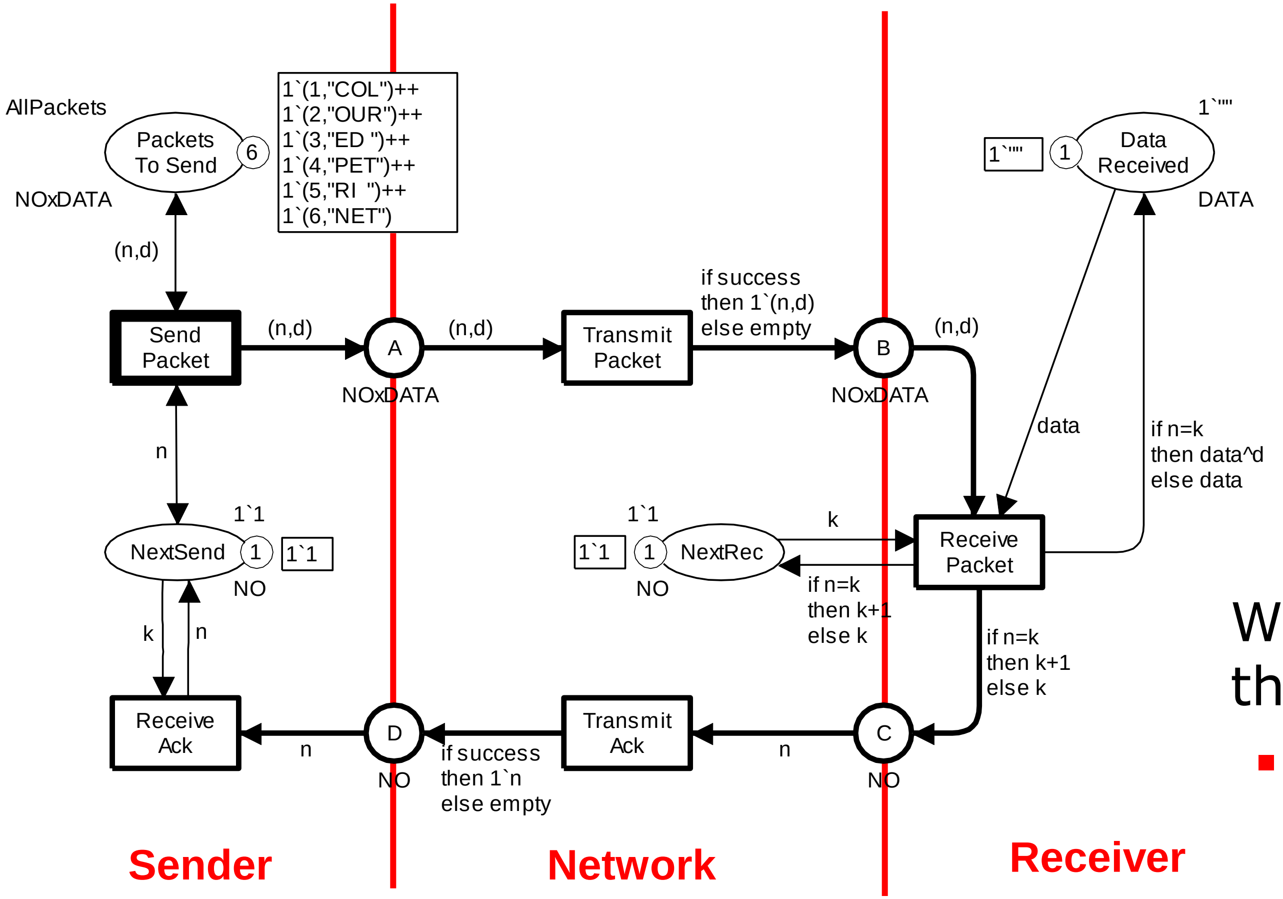
Why do we need modules?

- Impractical to draw a CPN model of a large system as a single net.
  - Would become very large and unhandy.
  - Could be printed on separate sheets and glued together, but it would be difficult to get an overview and make a nice layout.

- The human modeller needs abstractions that make it possible to concentrate on only a few details at a time.
  - CPN modules can be seen as black-boxes, where the modeller (when desired) can forget about the details within the modules.
  - This makes it possible to work at different abstraction levels.

- There are often system components that are used repeatedly.
  - Inefficient to model these components several times.
  - Instead we define a module, and use the module repeatedly.
  - In this way there is only one description to read, and one description to modify when changes are necessary.

Simple protocol



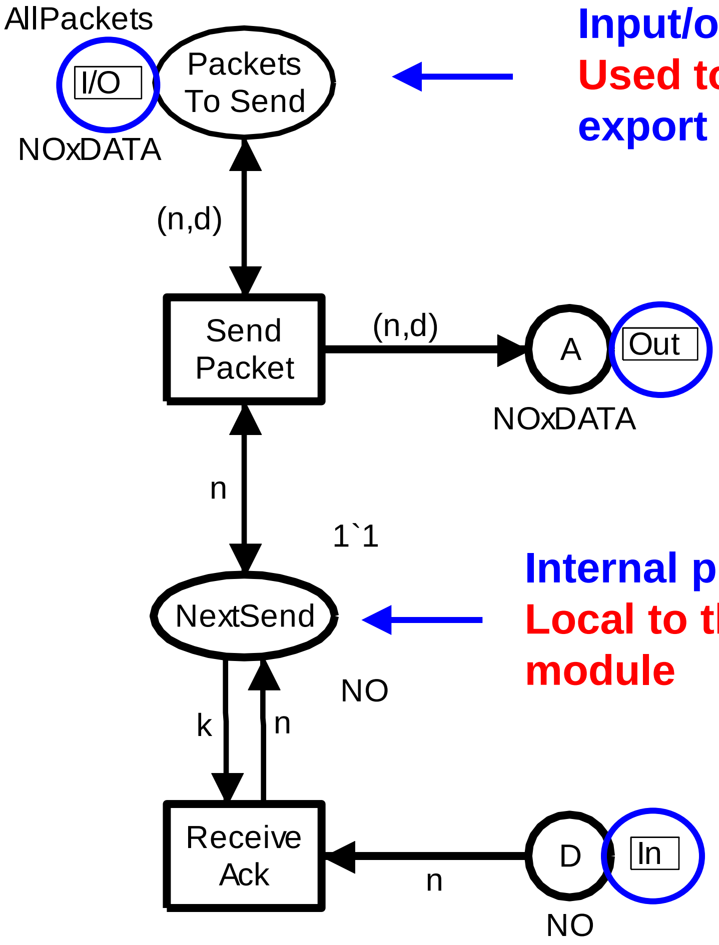
The protocol model can be divided into three modules:

- Sender.
- Network.
- Receiver.

We "cut" the model into three parts:

- Using the buffer places as interfaces between the modules.

Sender module



Input/output port
Used to import and export tokens

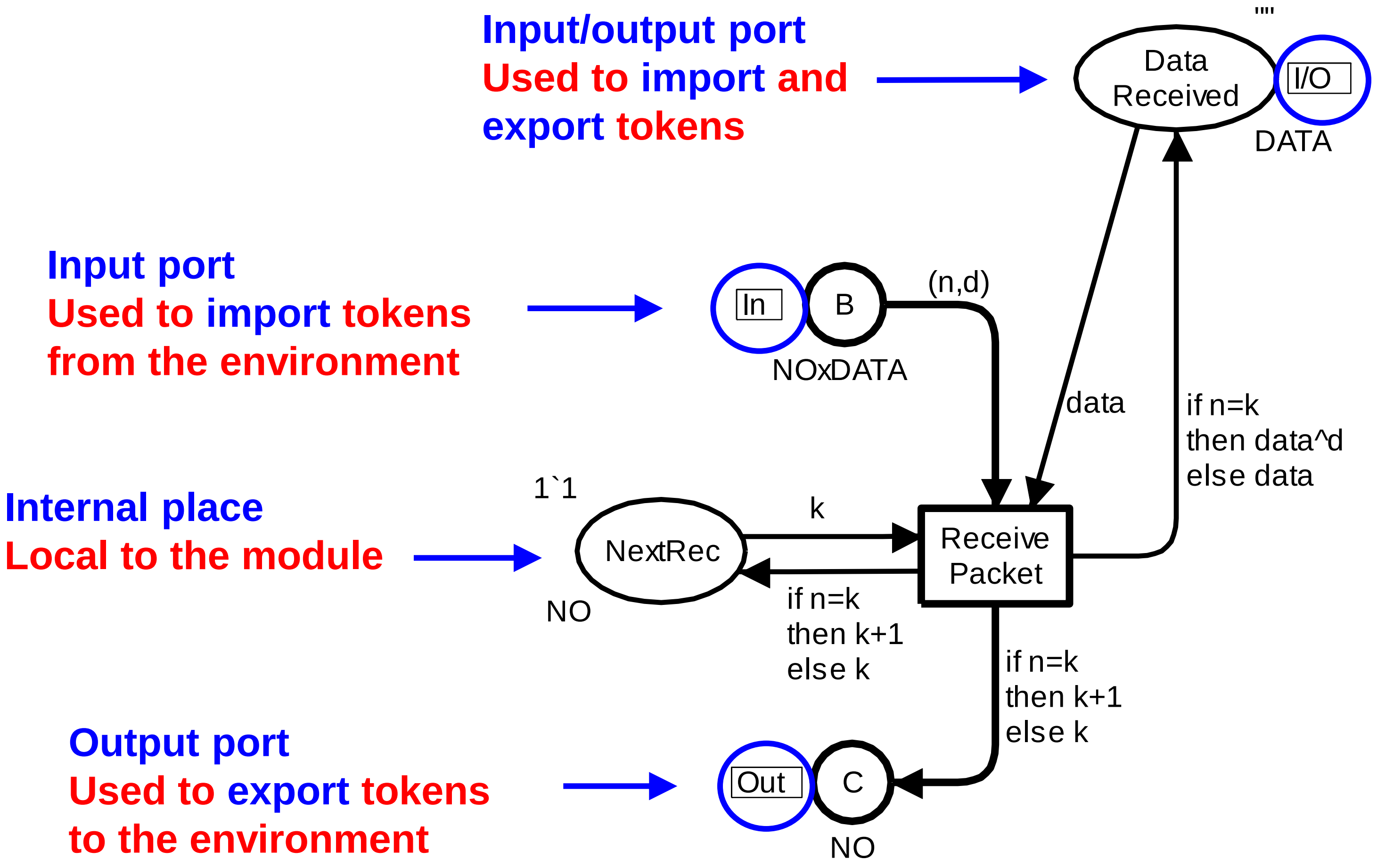
Output port
Used to export tokens to the environment

Internal place
Local to the module

Input port
Used to import tokens from the environment

The module interacts with other modules via three port places.

Receiver module



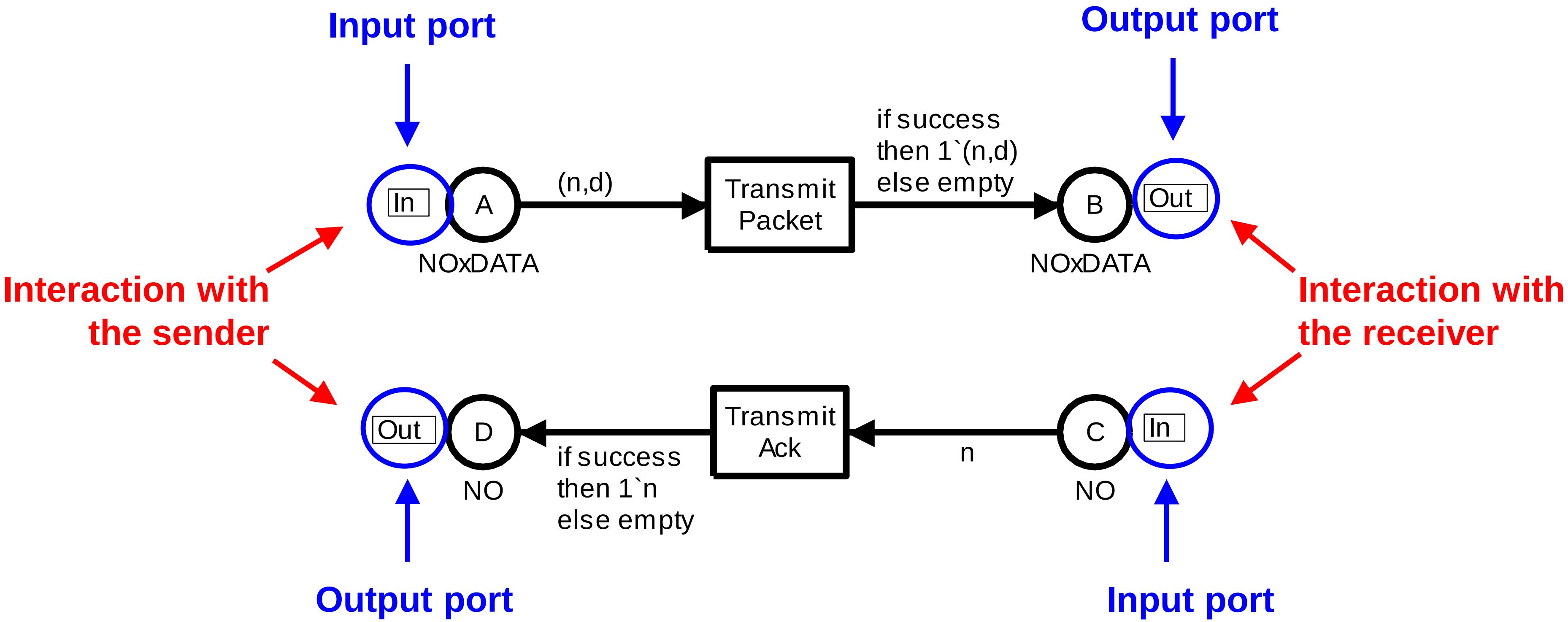
Input/output port
Used to import and export tokens

Input port
Used to import tokens from the environment

Internal place
Local to the module

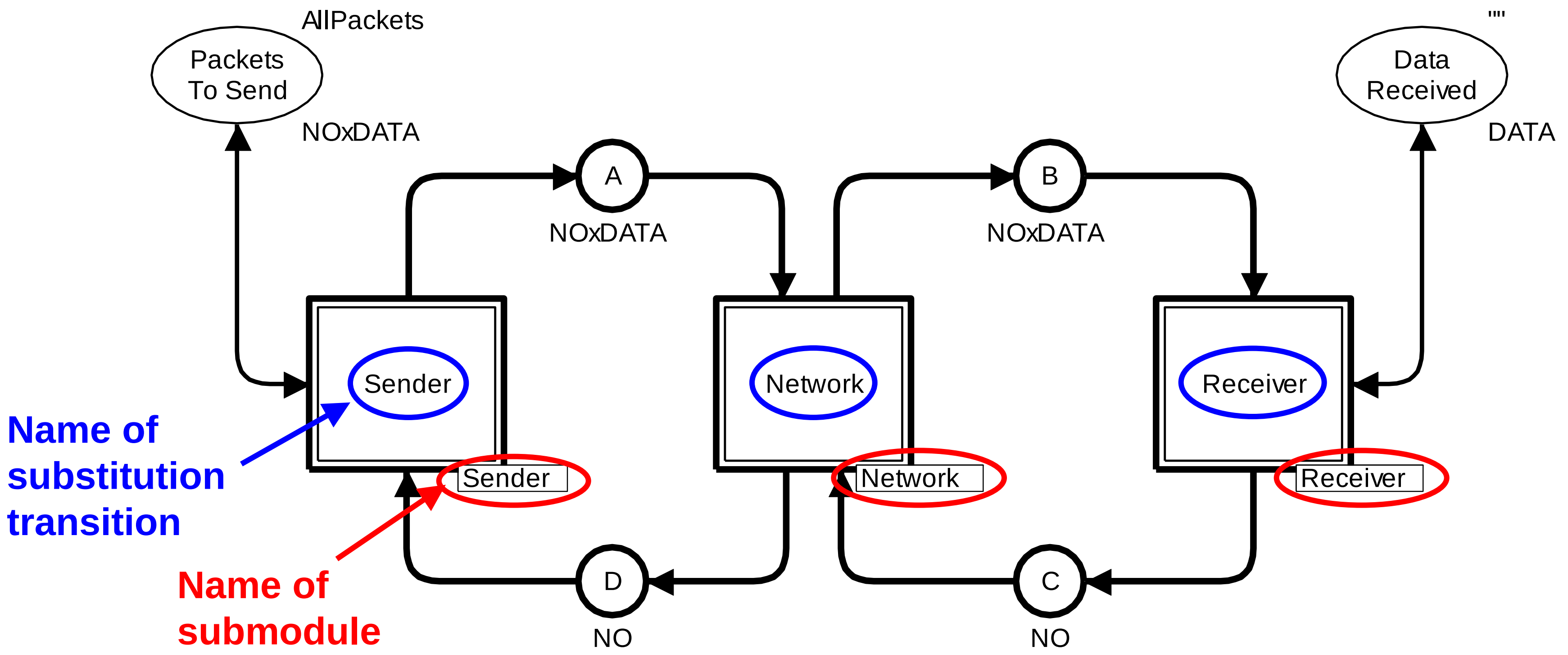
Output port
Used to export tokens to the environment

Network module


Input port
Output port
In
A
(n,d)
Transmit Packet
if success then 1`(n,d) else empty
B
Out
NOxDATA
NOxDATA
Interaction with the sender
Interaction with the receiver
Out
D
Transmit Ack
n
C
In
NO
if success then 1`n else empty
NO
Output port
Input port

Protocol module

- Provides a more abstract view of the protocol system.
- “Glues” the three other modules together.

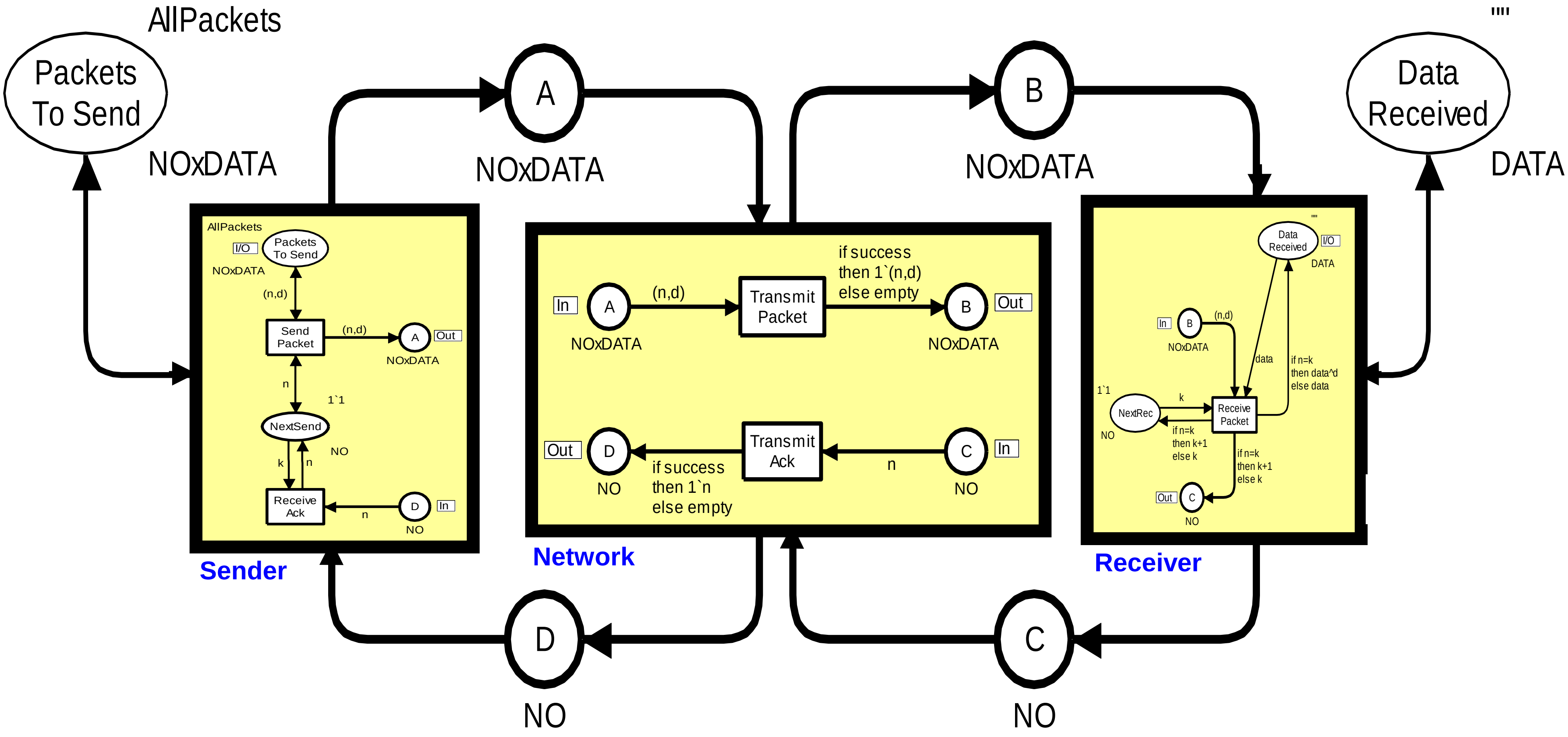


Name of substitution transition

Name of submodule

- Three substitution transitions referring to three different modules.

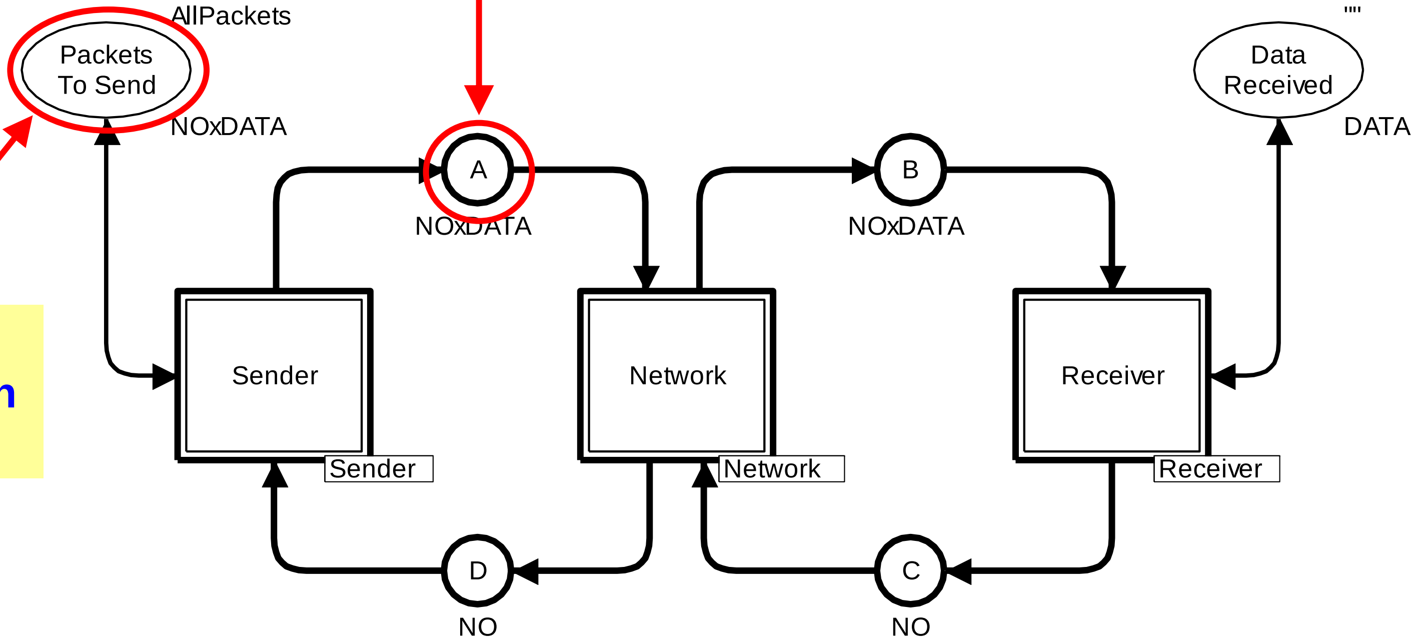
Protocol module



Protocol module

- The places surrounding substitution transitions are socket places.
- They constitute the interface for the substitution transition.

Input socket for Network substitution transition
Output socket for Sender substitution transition

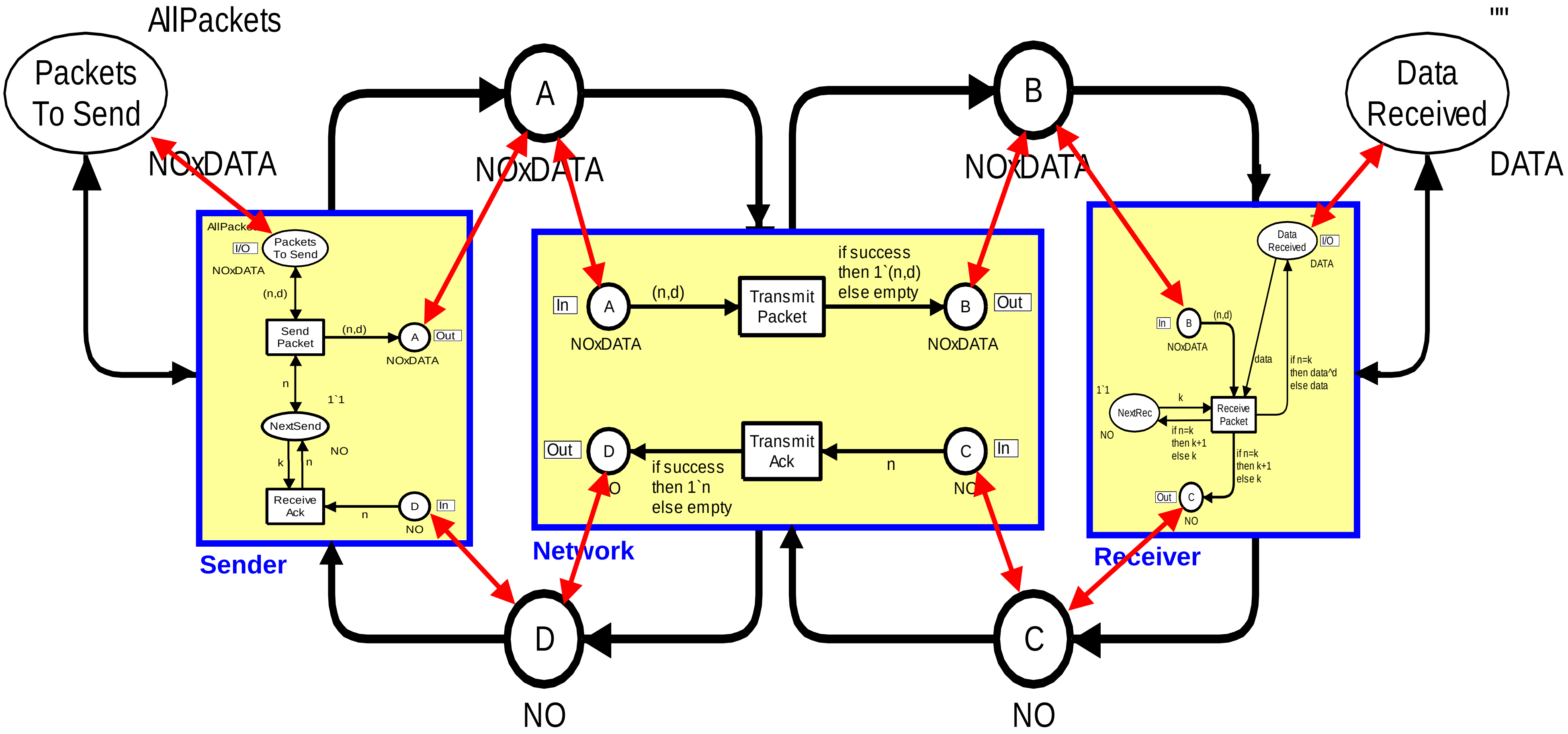


Input/output socket for Sender substitution transition

Port-socket relation

- The interfaces must be related to each other.
- Each port place of a submodule is related to a socket place of its substitution transition:
  - input port ↔ input socket.
  - output port ↔ output socket.
  - input/output port ↔ input/output socket.
- Ports and sockets that are related to each other constitute different views of a single compound place.
  - They have the same marking.
  - When a token is added/removed at one of them it is also added/removed at the other.
  - Also the colour sets and initial markings are identical.
- For the protocol system ports and sockets have identical names, but this is not required in general.

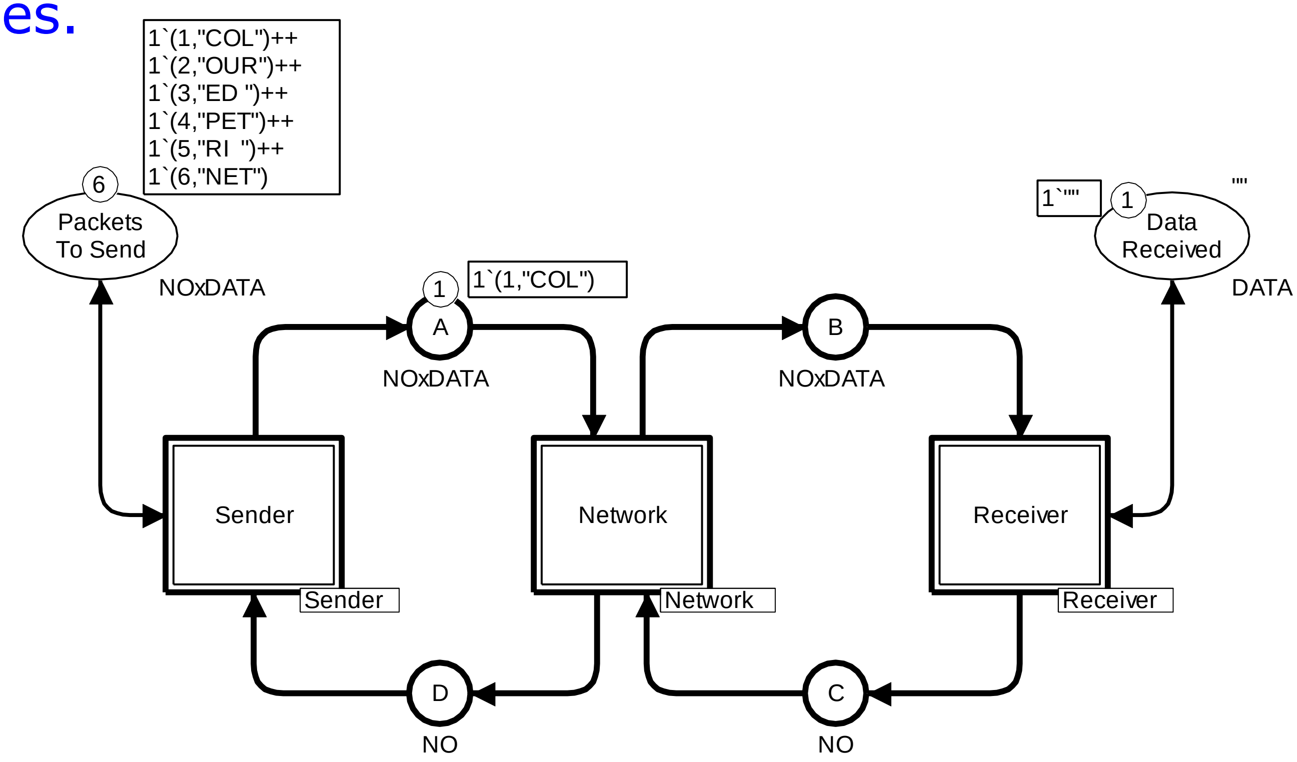
Port-socket relation



Demonstration in CPN Tools

Substitution transitions

- Substitution transitions do not have arc expressions and guards.
- They do not become enabled and they do not occur.
- Instead they represent the compound behaviour of their submodules.



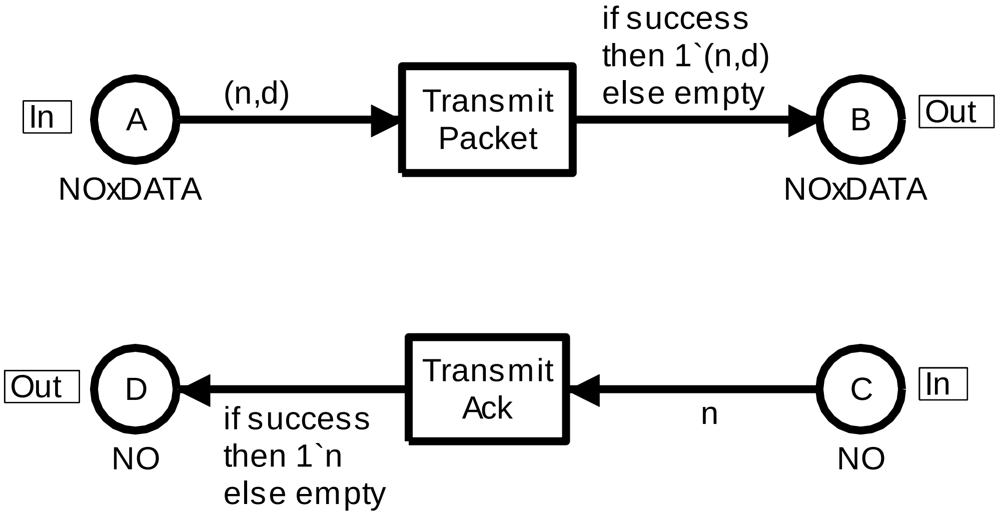
Abstraction levels

- In the protocol system we only have two different levels of abstraction.
  - Highest: Protocol module.
  - Lowest: Sender, Network and Receiver modules.

- CPN models of larger systems typically have up to 10 abstraction levels.

Second version of hierarchical protocol

- The two transitions in the Network module have a similar behaviour.
  - The upper transition transmits packets.
  - The lower transition transmits acknowledgements.



- We would like to define a single Transmit module and use this twice.
- To do this we need to make the colours sets identical.

Packets and acknowledgements

- Until now:

One colour set for acknowledgement packets

```
colset NO      = int;
colset DATA    = string;
colset NOxDATA = product NO * DATA;
```

Another colour set for data packets

- Instead we define a common colour set which can be used for both data packets and acknowledgement packets:

```
colset PACKET  =  union Data : NoxDATA + Ack : NO;
```

`Data(1,"COL")`

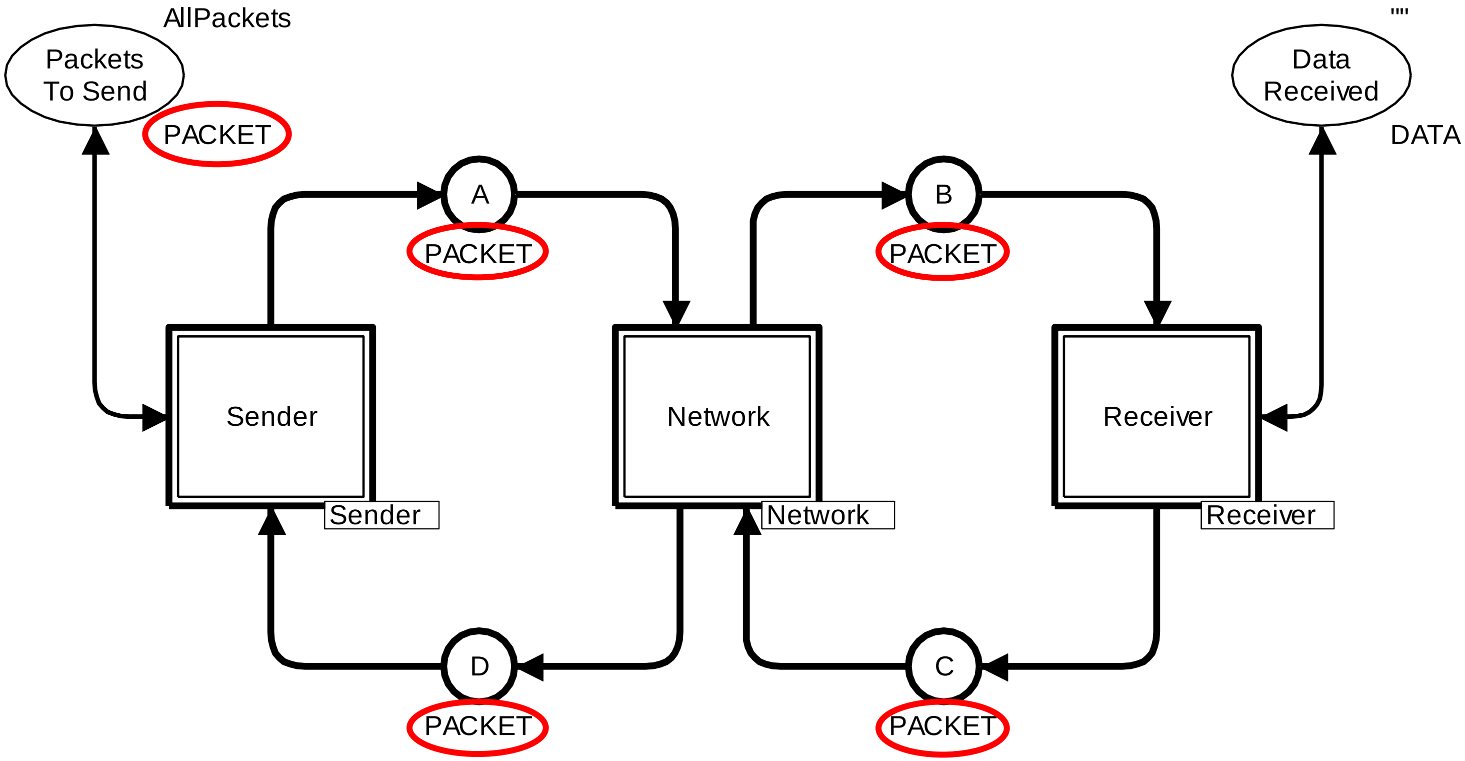
`Ack(2)`

Data

Acknowledgement

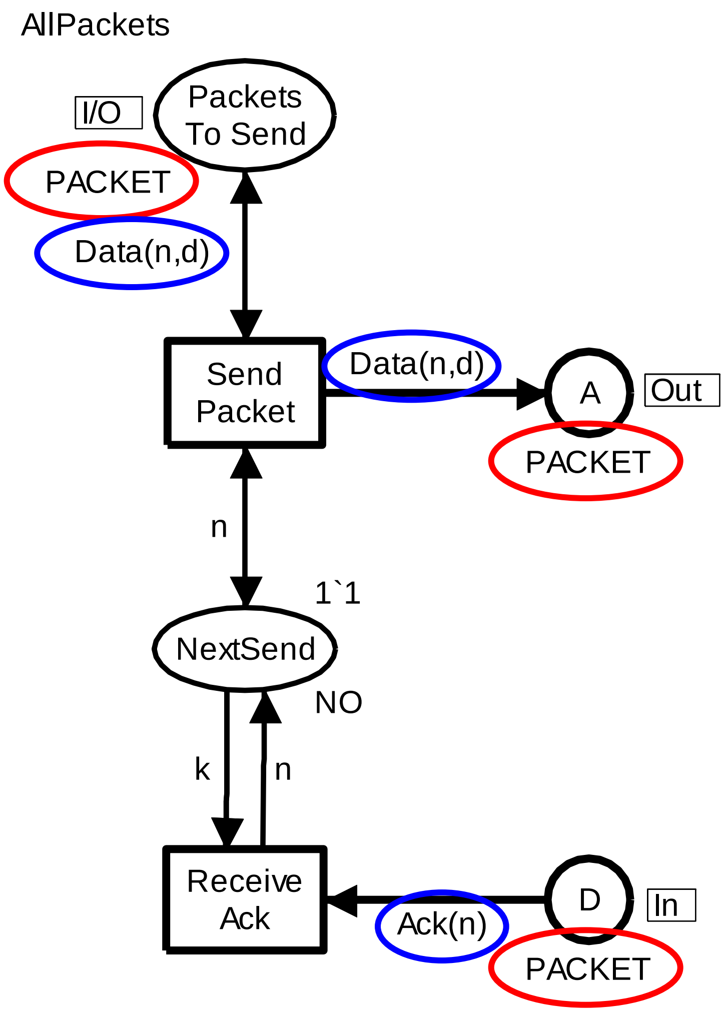
Modified Protocol module

- Uses the new type. Otherwise there are no changes.



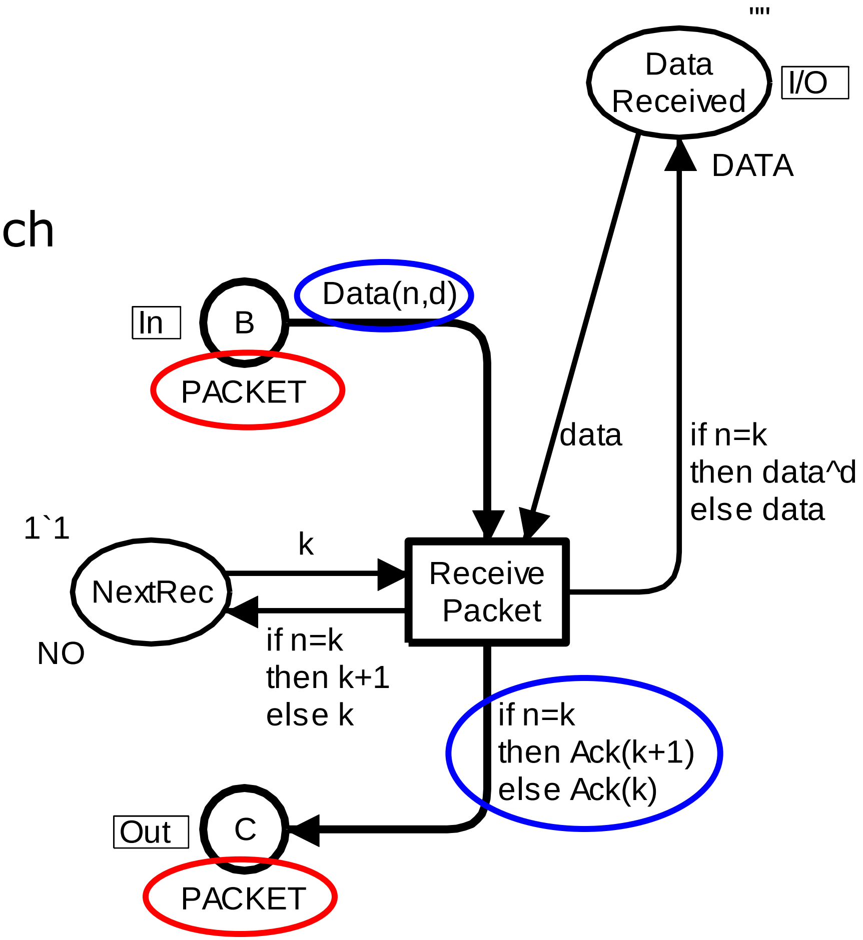
Modified Sender Module

- Uses the new type.
- Arc expressions have been modified to match the new type.



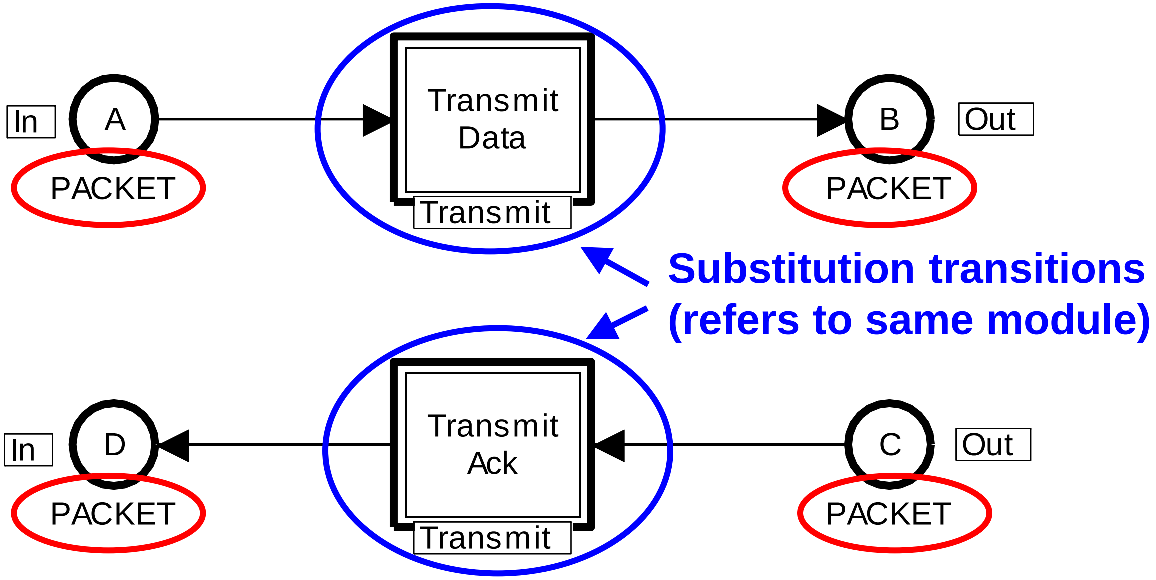
Modified Receiver module

- Uses the new type.
- Arc expressions have been modified to match the new type.



Modified Network module

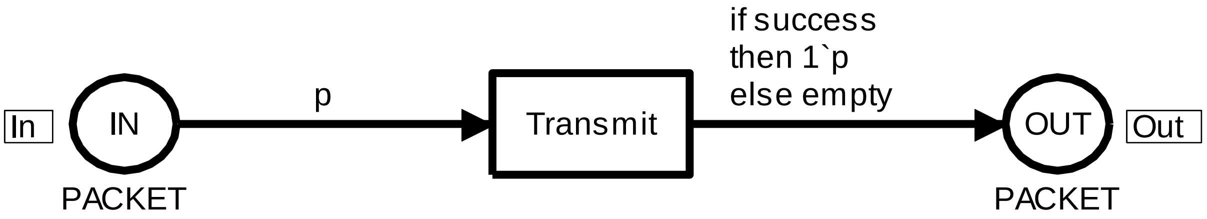
- Uses the new type.
- Contains two substitution transitions.



- The two substitution transitions refer to the same submodule.

Transmit module

- The Transmit module can handle both data packets and acknowledgement packets.

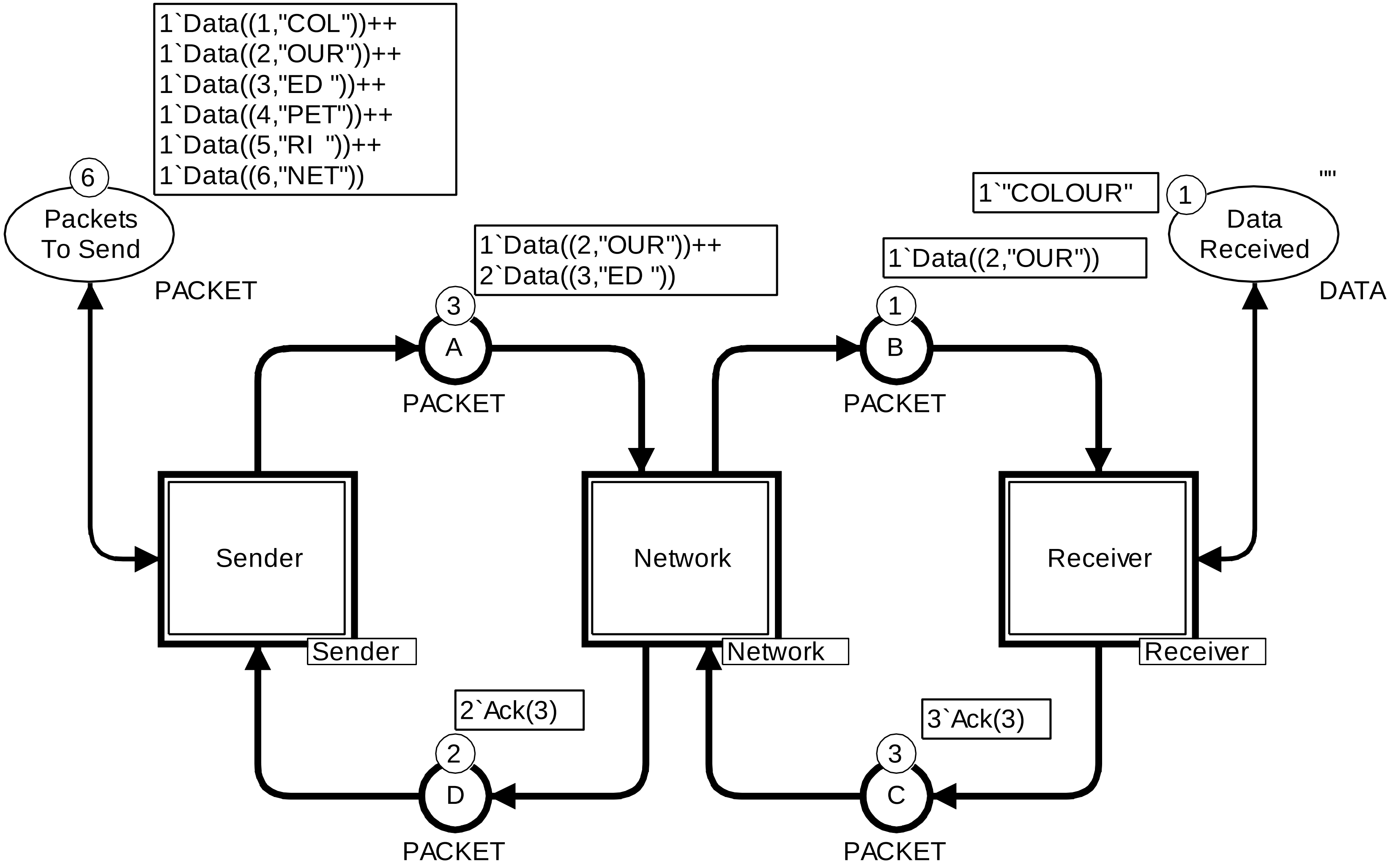


```
colset PACKET = union Data : NOxDATA + Ack : NO;
var p : PACKET;
```

- The variable p can be bound to a data packet such as Data(1,"COL").
- It can also be bound to an acknowledgement packet such as Ack(2).

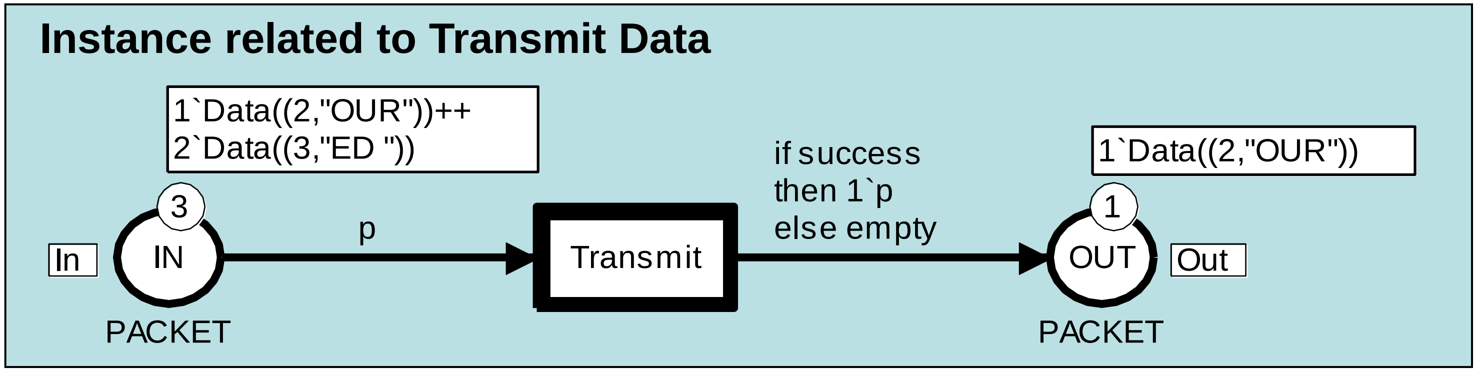
- Both kinds of packets are handled in the same way.

Marking of Protocol module after some steps

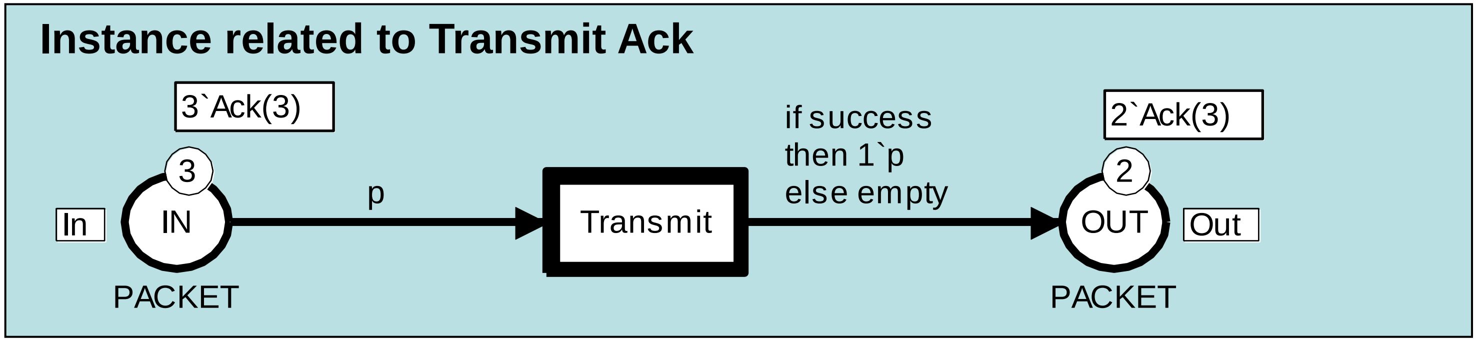


Marking of Transmit module

- There are two instances of the transmit module.
- One instance for each substitution transition that uses to the module.



Instance related to Transmit Data



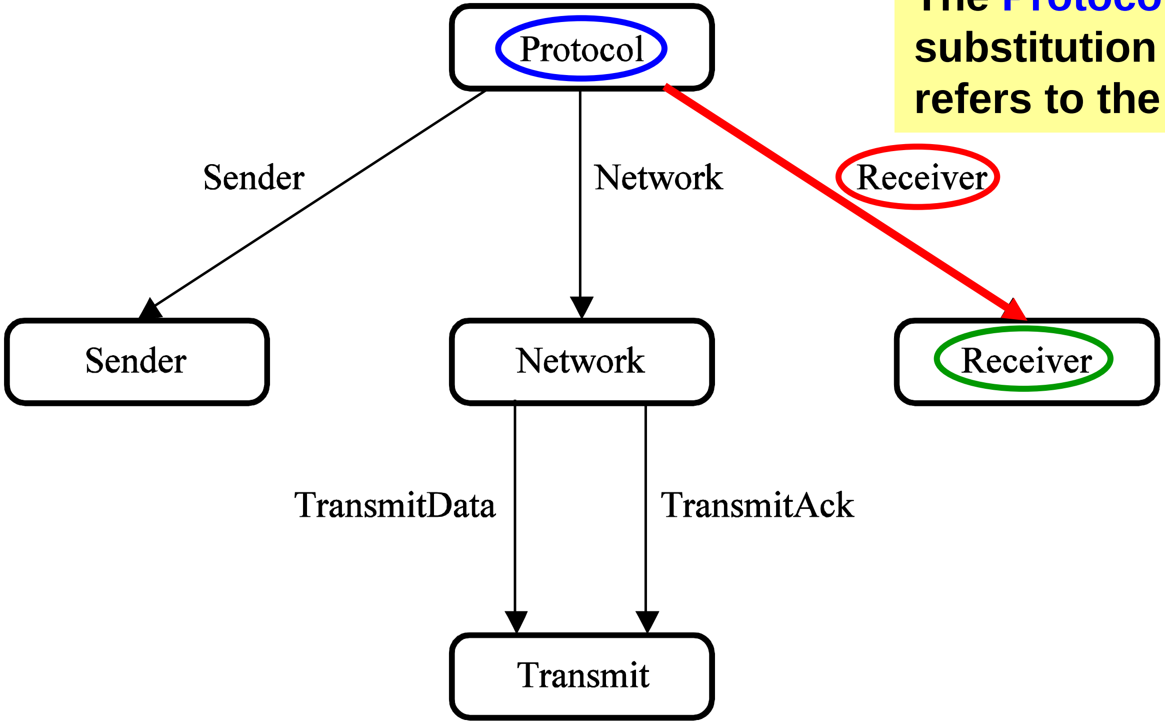
Instance related to Transmit Ack

- Each instance has its own marking which is independent of the marking of the other instance.

Module hierarchy

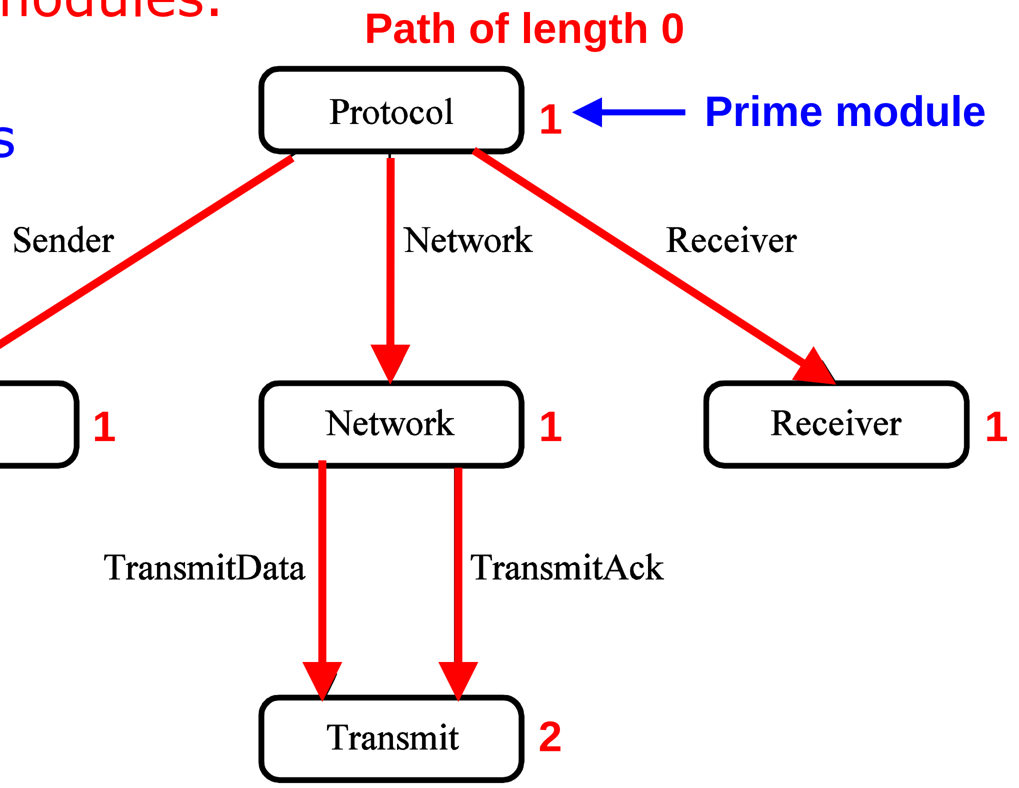
- Represents the relationship between modules.
- Each node represents a module.
- Each arc is labelled by a substitution transition.

The Protocol module contains a substitution transition Receiver which refers to the Receiver module



Module hierarchy

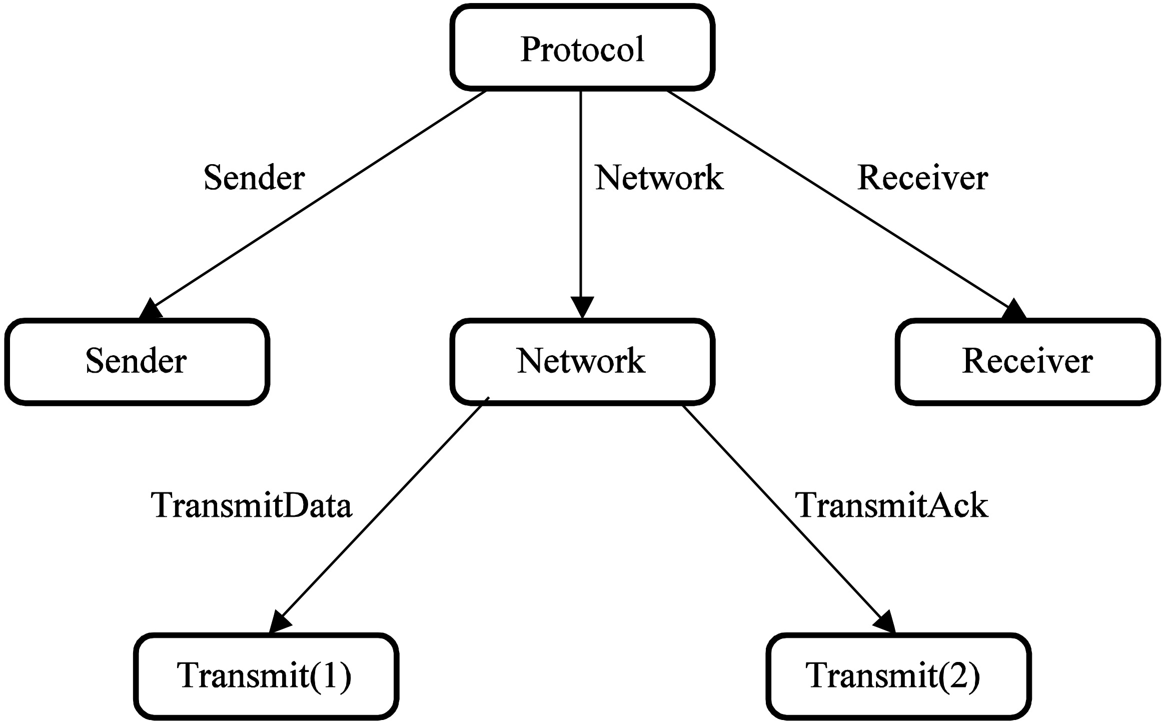
- The roots are called prime modules.
- Each module has as many instances as there are paths from a prime module.



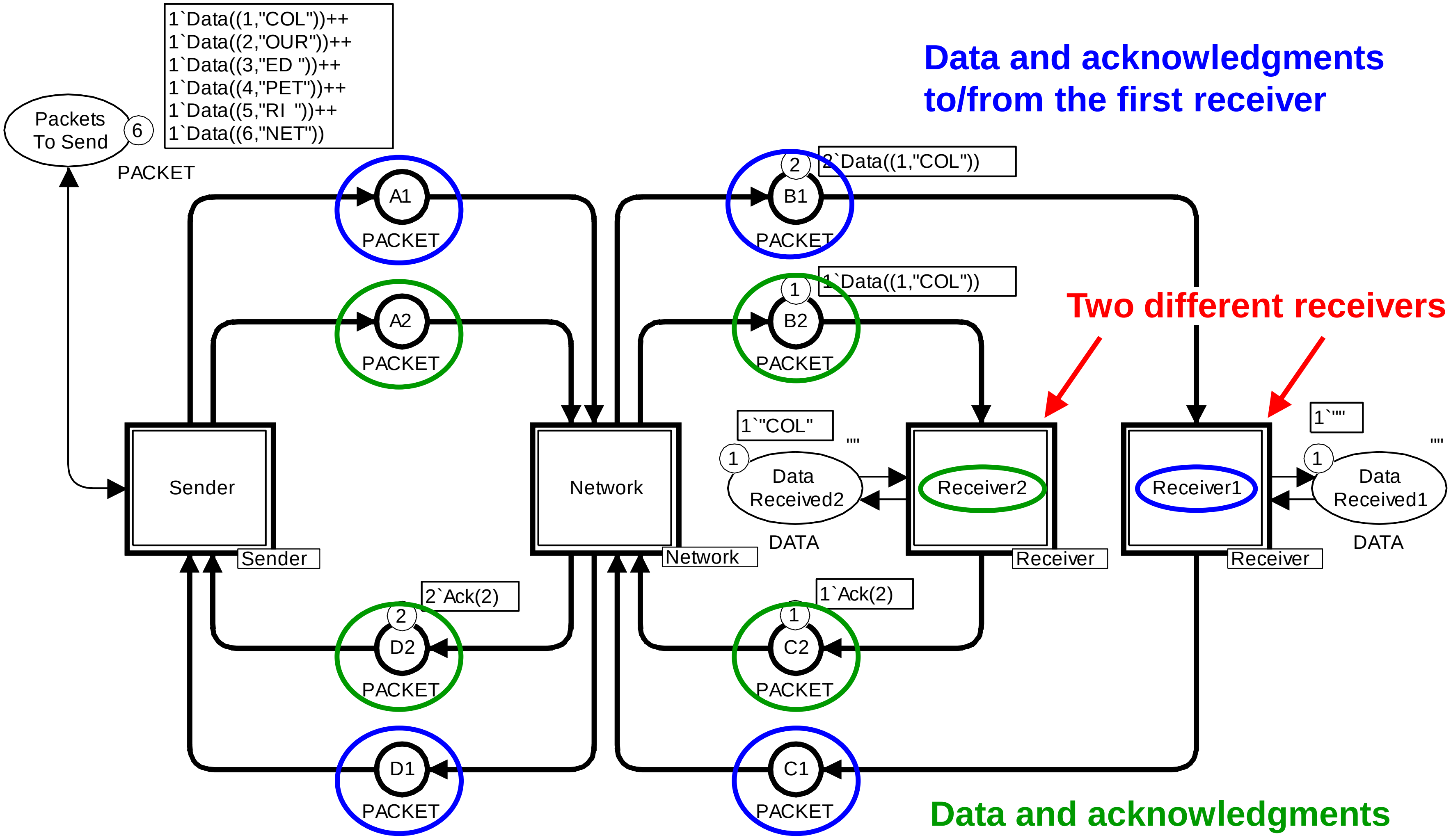
- The module hierarchy must be an acyclic graph.
- This guarantees that each module has a finite number of instances.

Instance hierarchy

- Similar to the module hierarchy, but now the two instances of the Transmit module are drawn as two separate nodes.
- The instance hierarchy is an unfolded version of the module hierarchy. It is a directed tree.



Protocol with multiple receivers

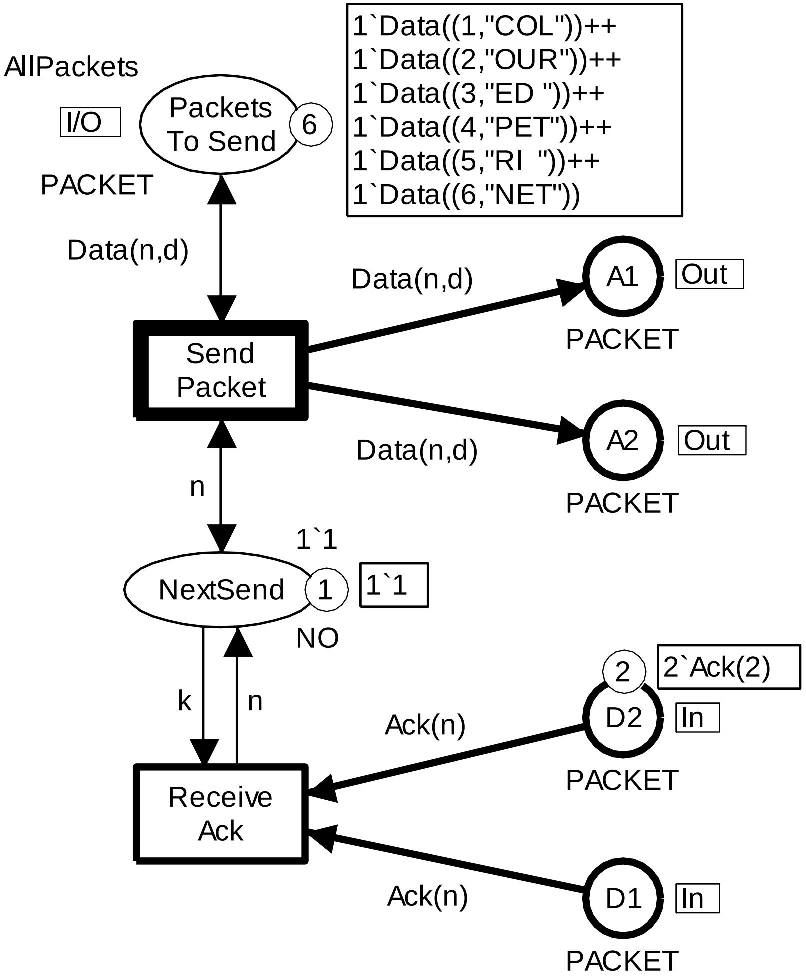


Data and acknowledgments
to/from the first receiver

Two different receivers

Data and acknowledgments
to/from the second receiver

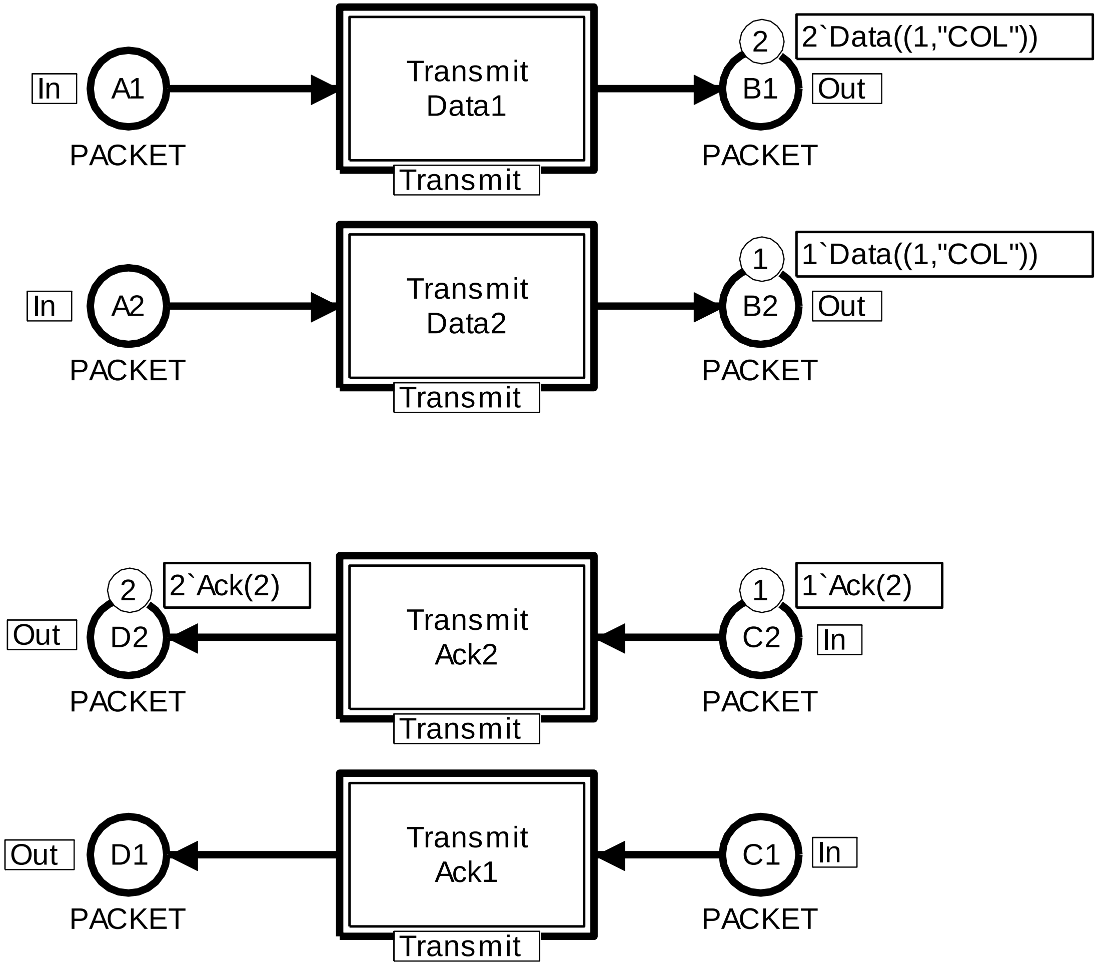
Sender module



- Identical data packets are broadcasted to the two receivers (via A1 and A2).

- ReceiveAck can only occur when there are identical acknowledgements from the two receivers (on D1 and D2).

Network module



Data to the first receiver

Data to the second receiver

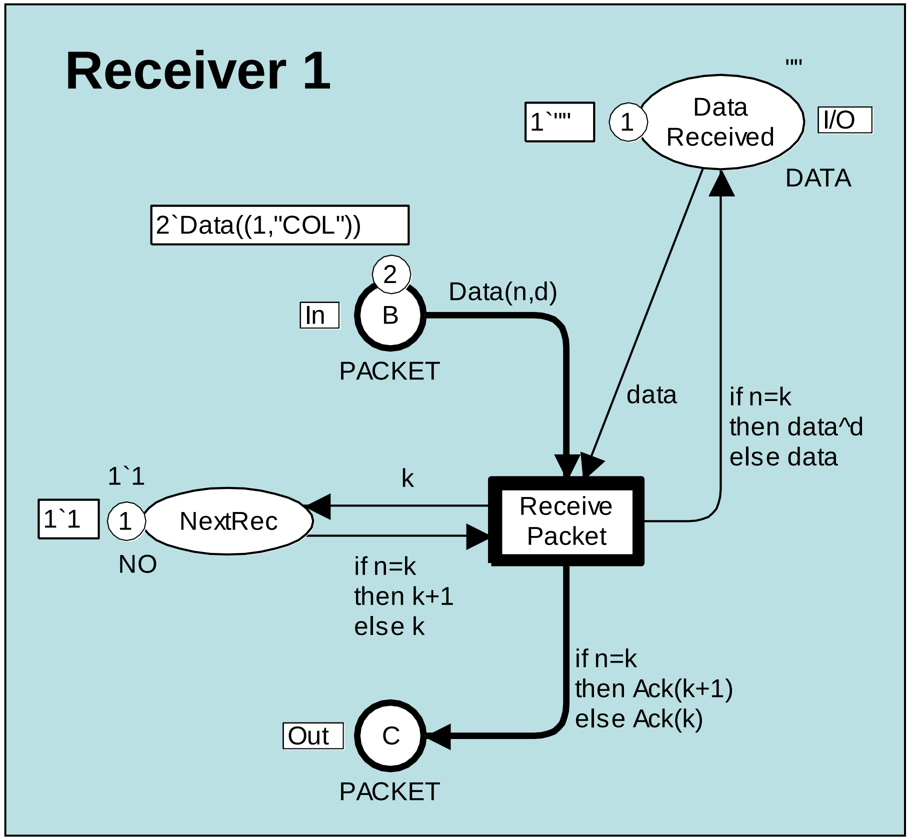
Acknowledgements from second receiver

Acknowledgements from first receiver

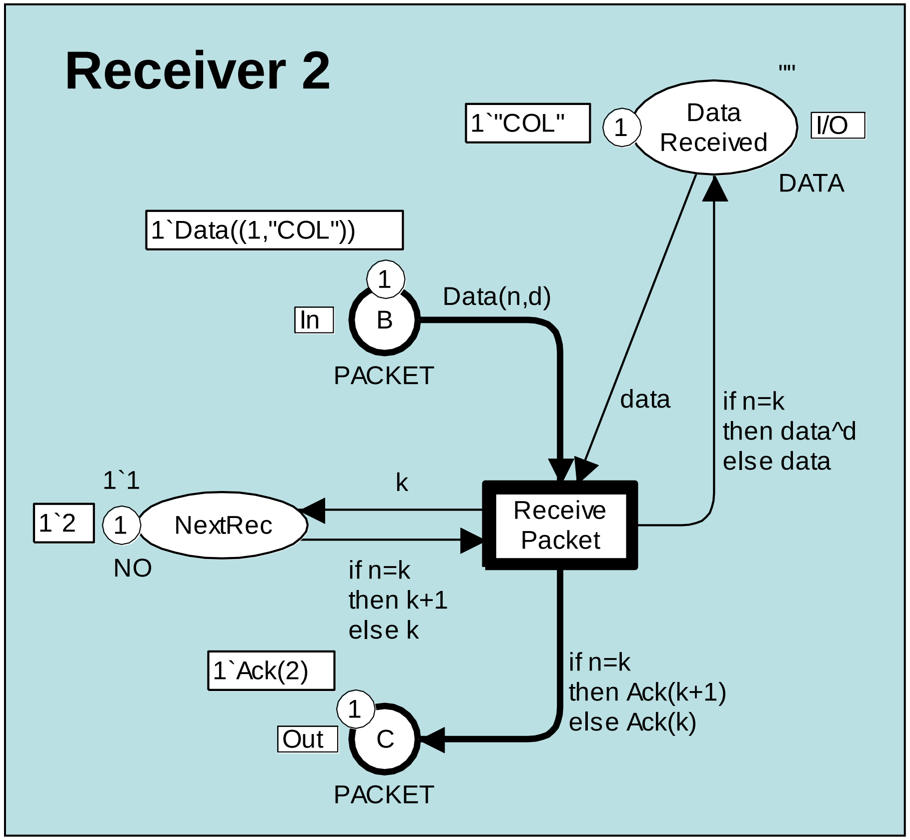
Receiver module (2 instances)

- Each instance has its own marking.

Waiting for packet no. 1

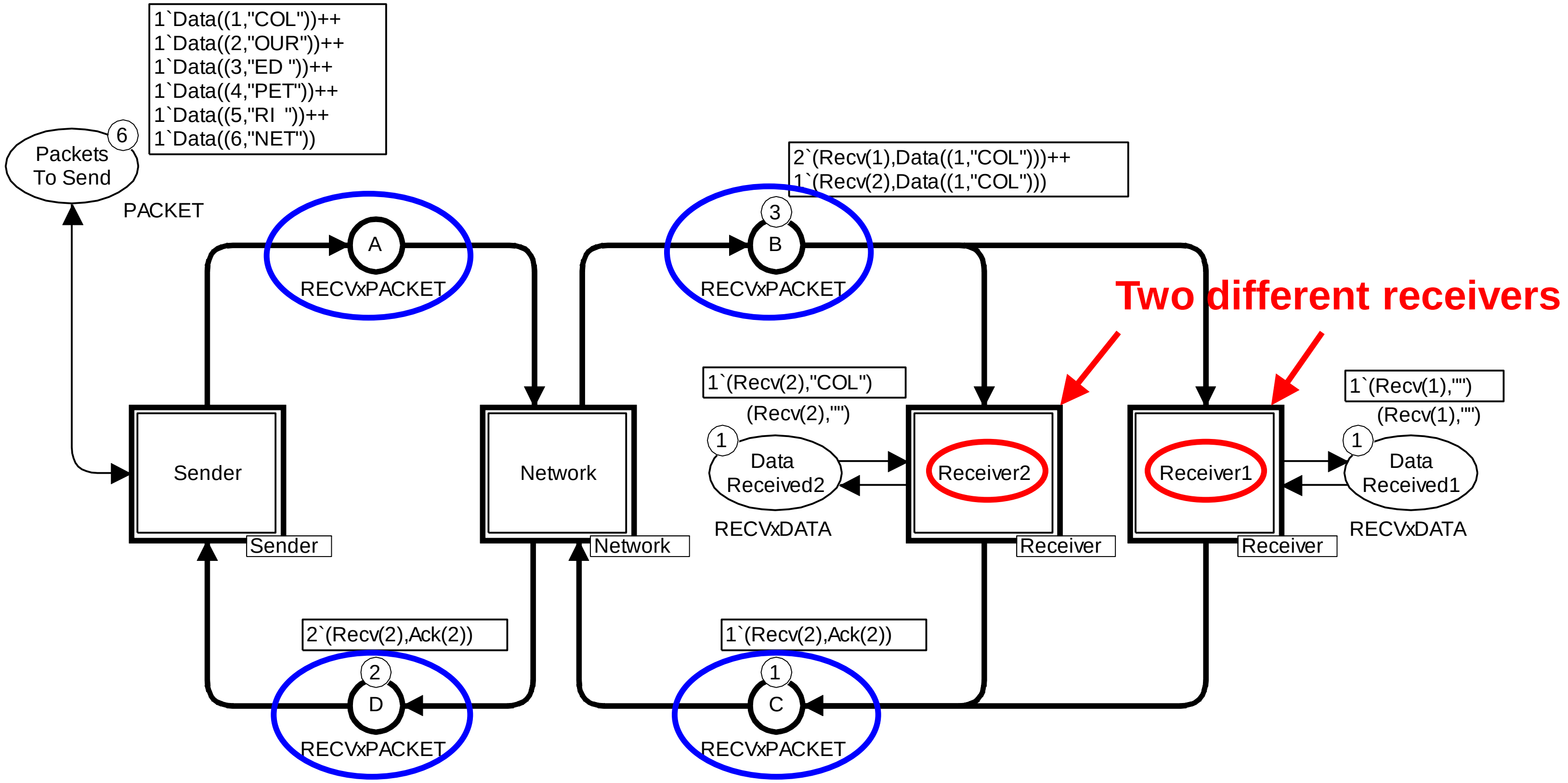


Waiting for packet no. 2



Second version with multiple receivers

- A1 and A2 have been folded into a single place A.
- Analogously, for B1 and B2, C1 and C2, D1 and D2.



Packets and acknowledgements

- To be able to make the folding we define an index colour set:

```
colset RECV = index Recv with 1..2;
```

```
Recv(1)
```

```
Recv(2)
```

Two different values

- Instead of using two different places such as A1 and A2 we use the token colour to tell which receiver the data goes to / acknowledgements comes from.

```
colset RECVxPACKET = product RECV * PACKET;
```

```
(Recv(1),Data(2,"OUR"))
```

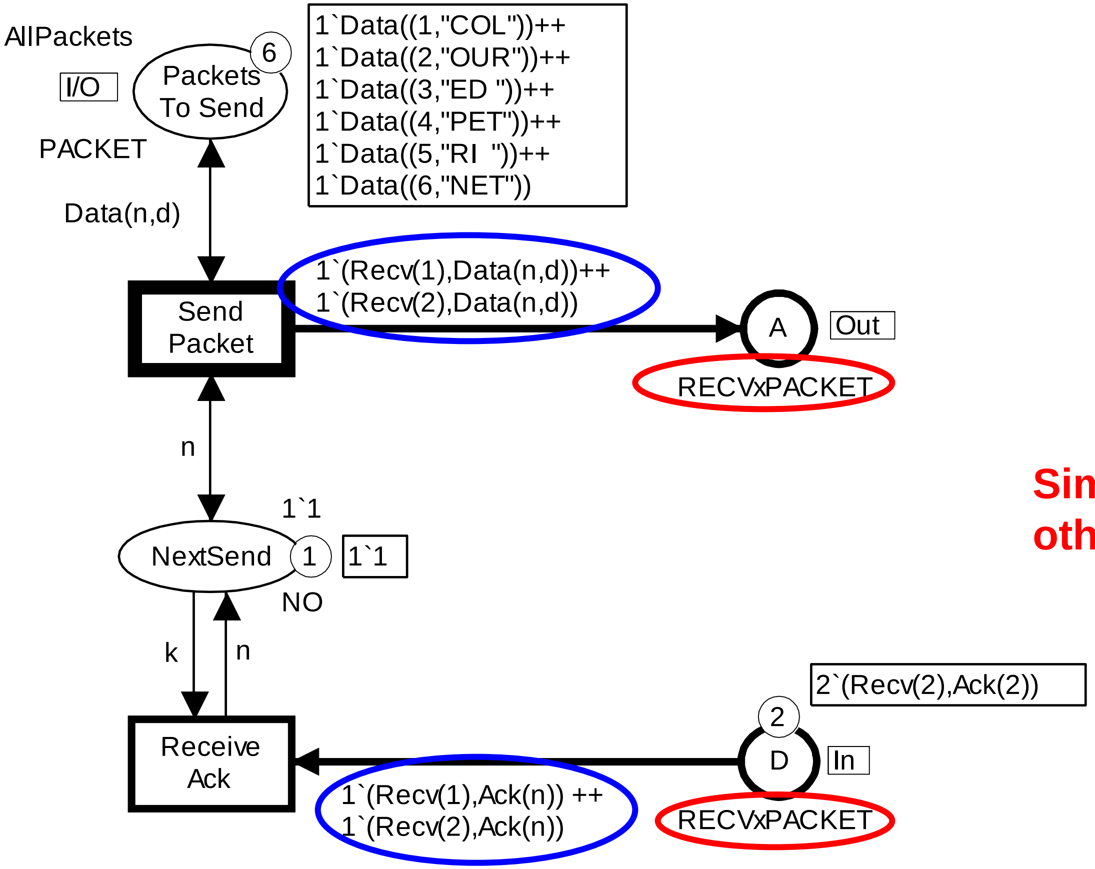
Data packet for the first receiver

```
(Recv(2),Ack(3))
```

Acknowledgement from the second receiver

Sender module

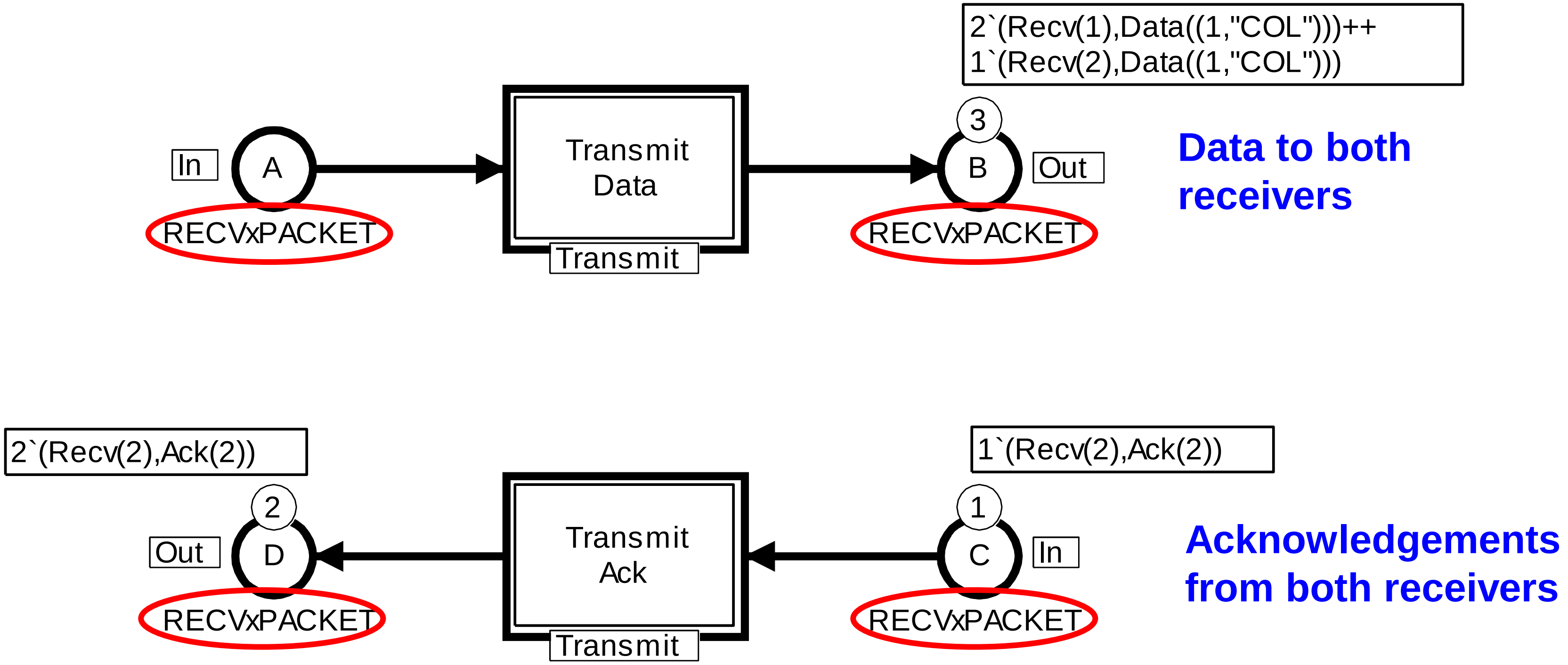
- Uses the new type.
- Arc expressions have been modified to match the new type.



Similar changes in the other three modules

Network module

- Uses the new type.
- Now we only need two instances of the Transmit module.

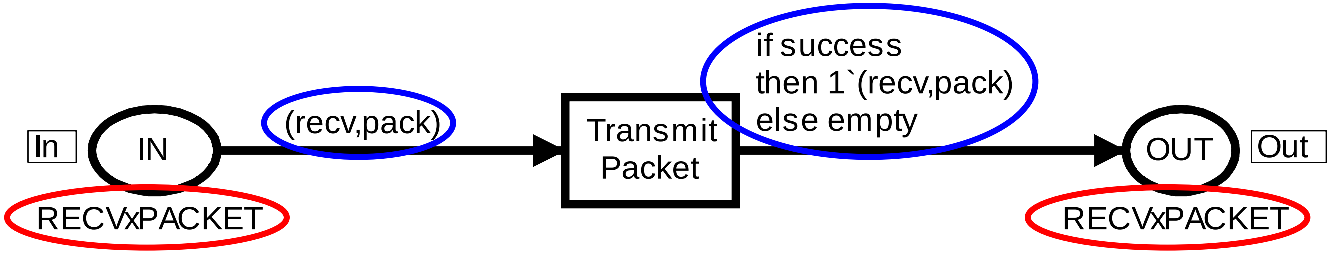


Data to both receivers

Acknowledgements from both receivers

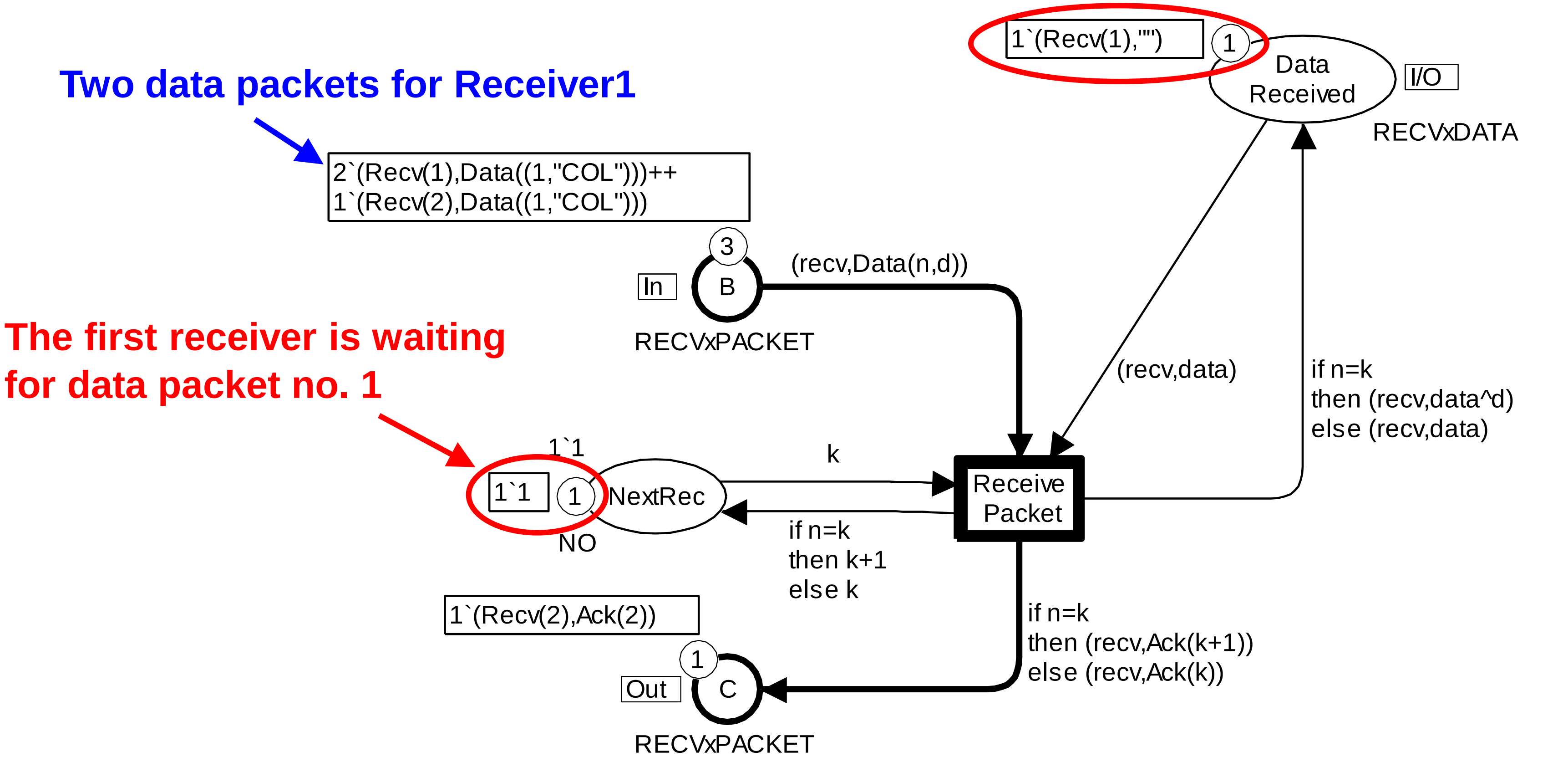
Transmit module

- Uses the new type.
- Arc expressions have been modified to match the new type.
- A new variable has been introduced.



```
var recv : RECV;
```

Receiver module Receiver no 1

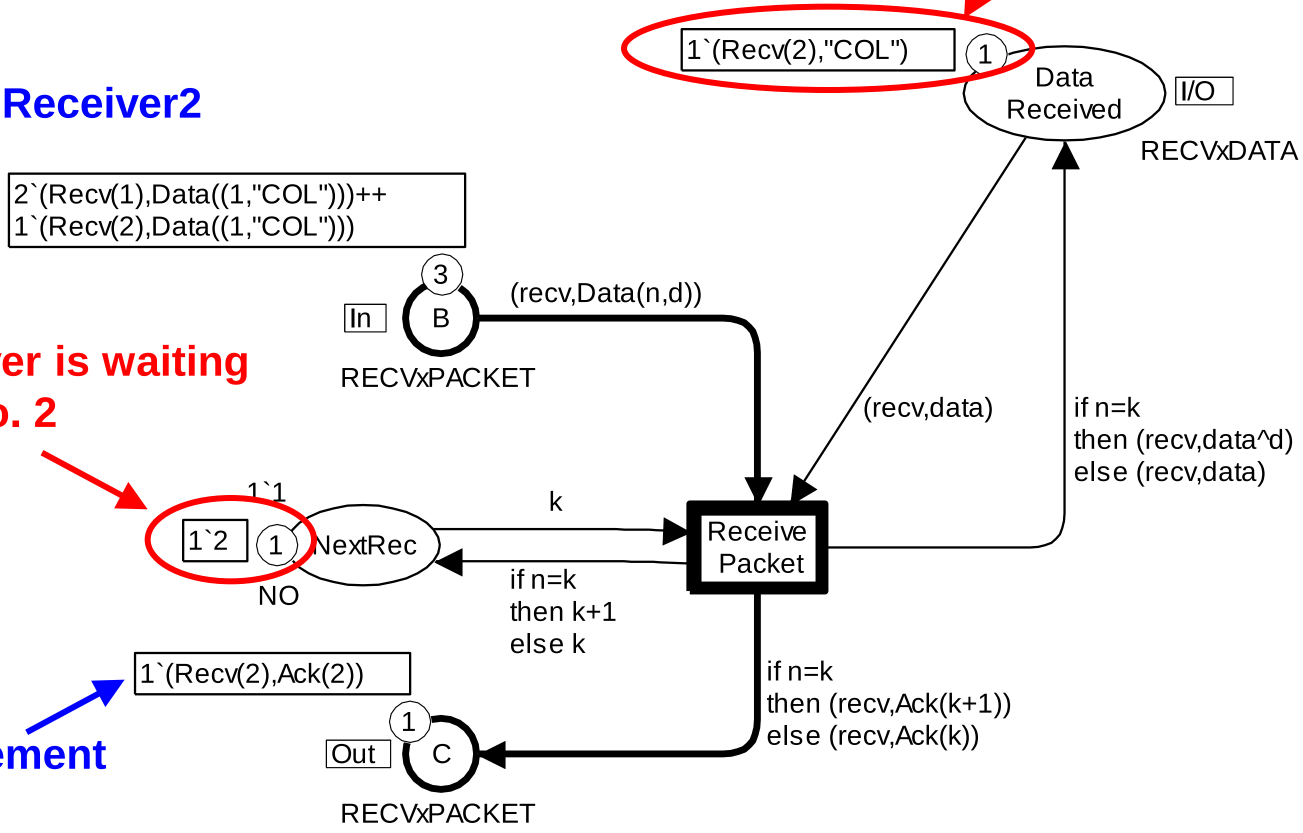


The first receiver has not yet received any data

Two data packets for Receiver1

The first receiver is waiting for data packet no. 1

Receiver module Receiver no 2



The second receiver has received the first data packet

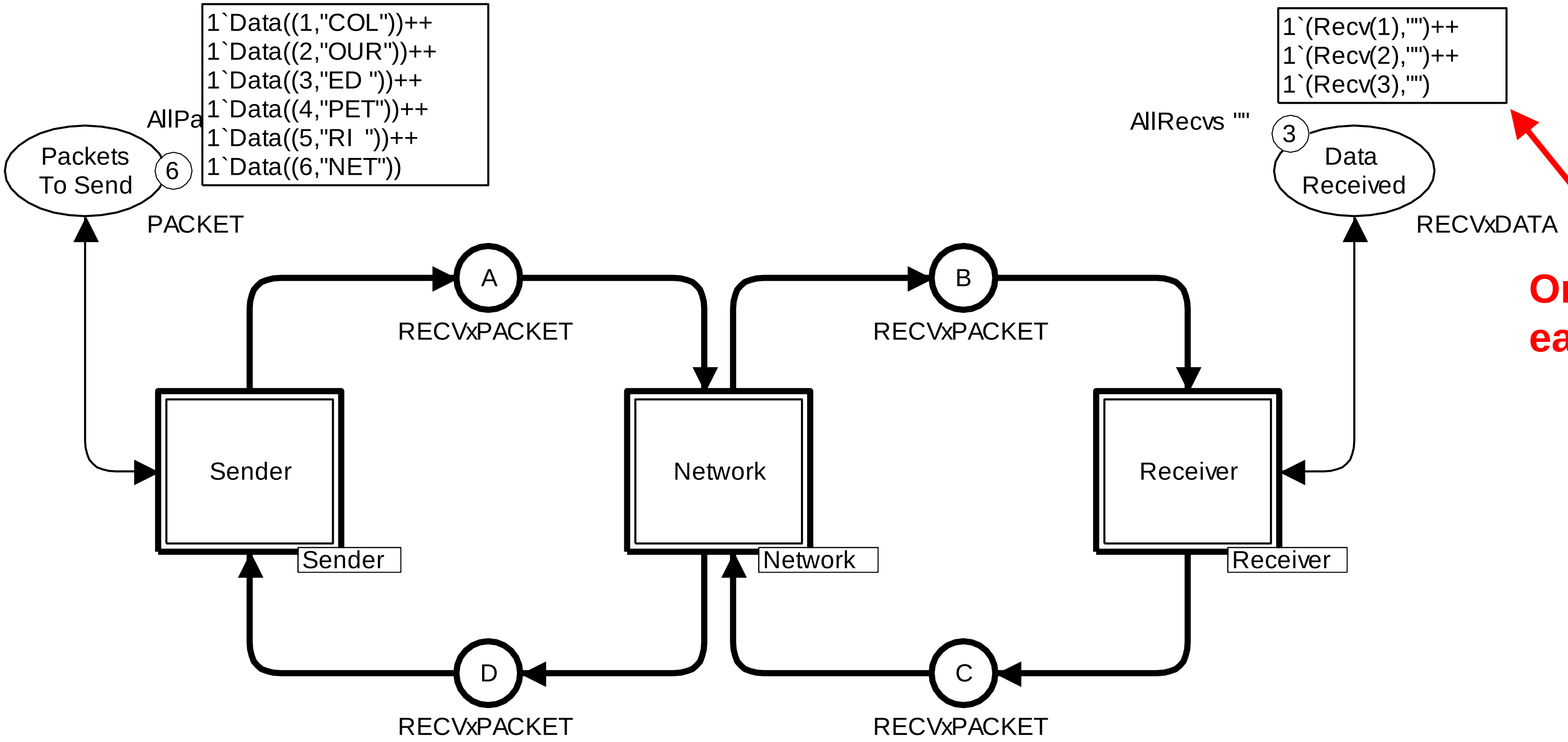
A data packet for Receiver2

The second receiver is waiting for data packet no. 2

An acknowledgement from Receiver2

Third version with multiple receivers

- We only have a single Receiver module.
- The Receiver module represents an arbitrary number of receivers.
- We will use token colours to distinguish the receiver tokens from each other.



One token for each receiver

Packets and acknowledgements

- We define the number of receivers by means of a constant.
- Can easily be changed without modifying the other definitions.

```
val NoRecv = 3;
```

```
colset RECV = index Recv with 1..NoRecv;
```

- As before we add a RECV component to DATA tokens and PACKET tokens:

```
colset RECVxDATA   = product RECV * DATA;
```

```
colset RECVxPACKET = product RECV * PACKET;
```

- We do the same for NO tokens:

```
colset RECVxNO     = product RECV * NO;
```

Initial marking for DataReceived

- For three receivers we want the initial marking to be:

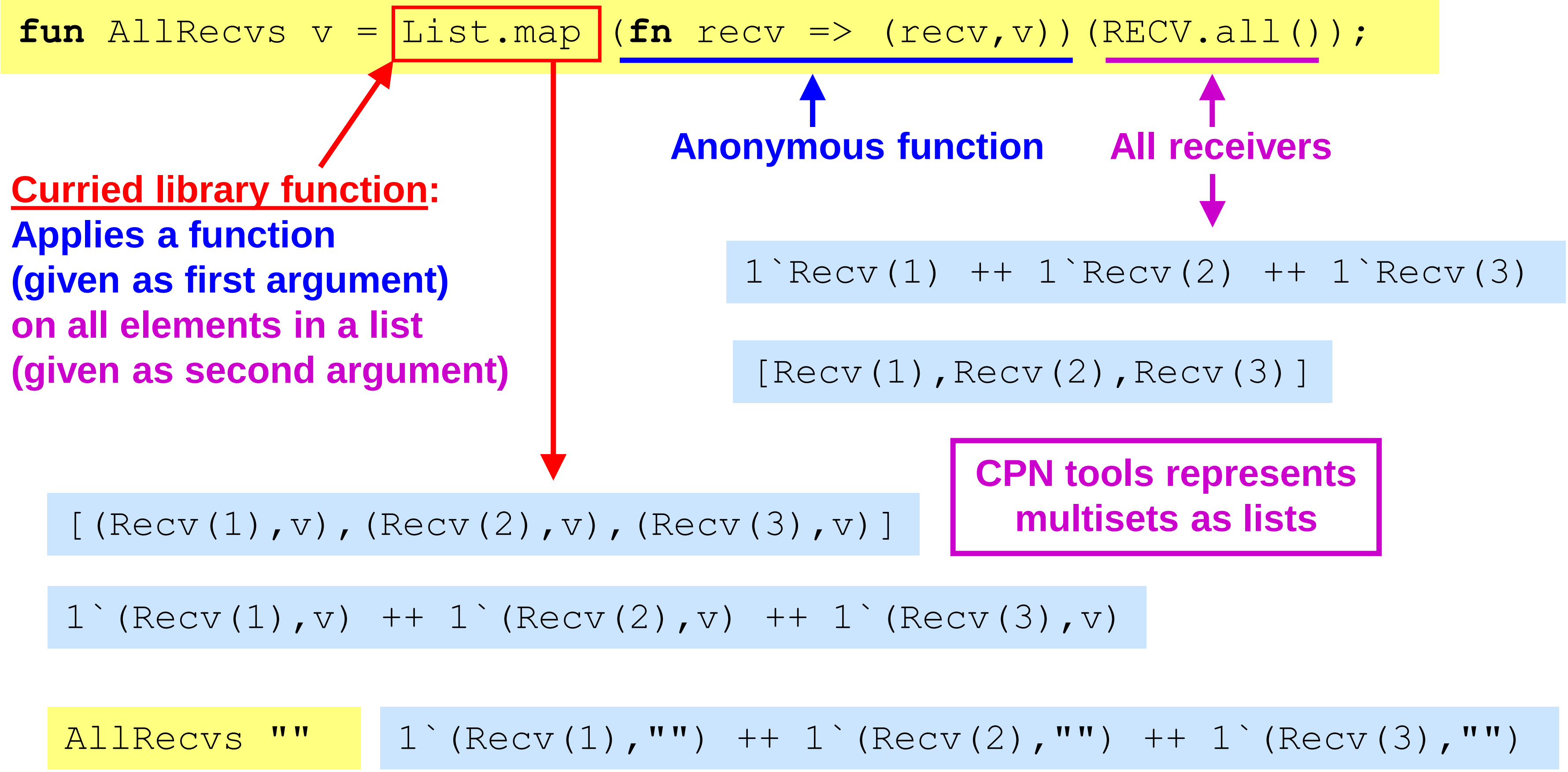
```
1`(Recv(1),"") ++ 1`(Recv(2),"") ++ 1`(Recv(3),"")
```

- For six receivers we want the initial marking to be:

```
1`(Recv(1),"") ++ 1`(Recv(2),"") ++ 1`(Recv(3),"") ++
1`(Recv(4),"") ++ 1`(Recv(5),"") ++ 1`(Recv(6),"")
```

- The initial marking should change automatically when the number of receivers changes.
- To obtain this, we define the initial marking by means of a function called AllRecvs.

Definition of function AllRecvs



```
fun AllRecvs v = List.map (fn recv => (recv,v))(RECV.all());
```

Anonymous function

All receivers

```
1`Recv(1) ++ 1`Recv(2) ++ 1`Recv(3)
```

```
[Recv(1),Recv(2),Recv(3)]
```

Curried library function:
Applies a function
(given as first argument)
on all elements in a list
(given as second argument)

```
[(Recv(1),v),(Recv(2),v),(Recv(3),v)]
```

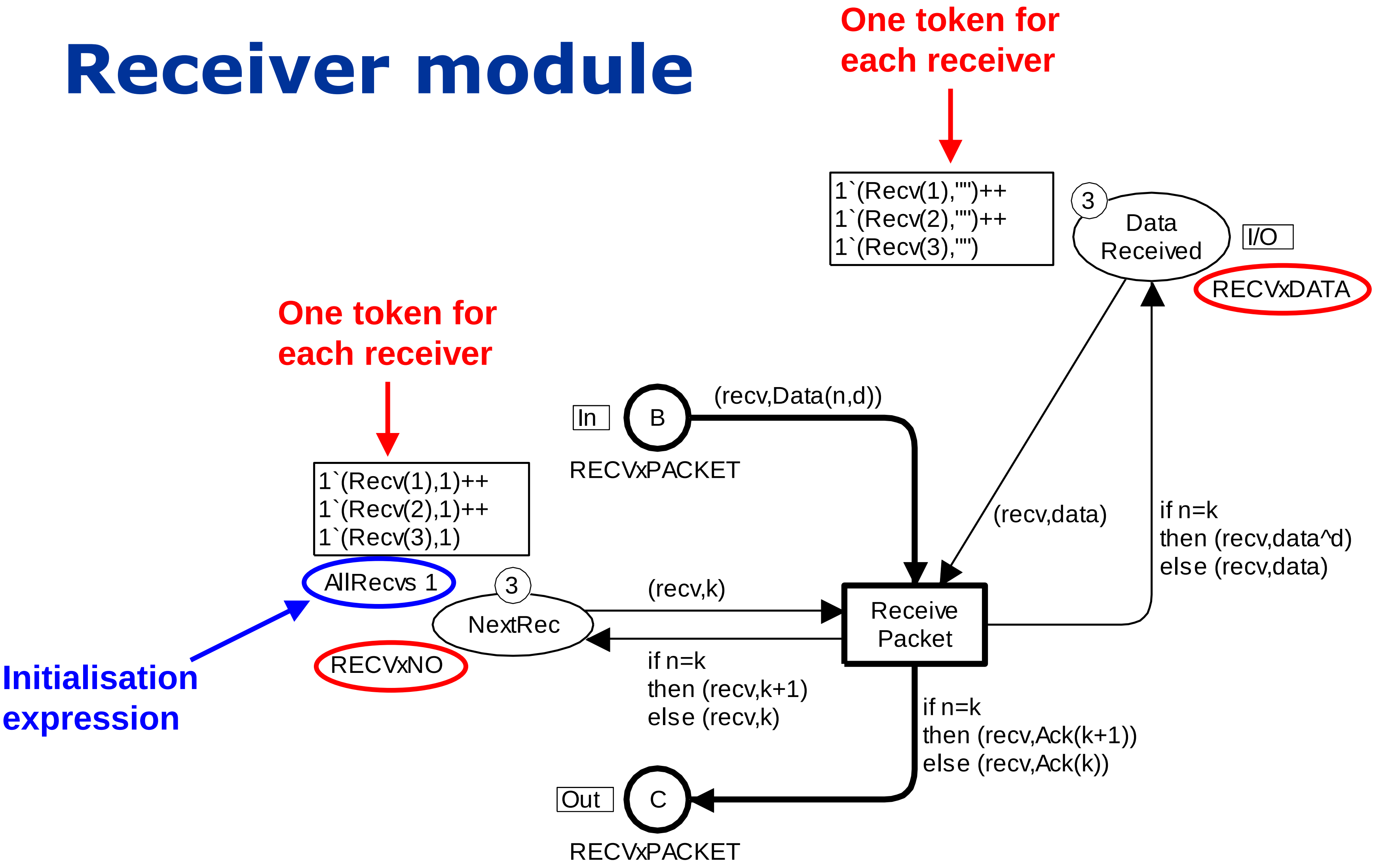
CPN tools represents multisets as lists

```
1`(Recv(1),v) ++ 1`(Recv(2),v) ++ 1`(Recv(3),v)
```

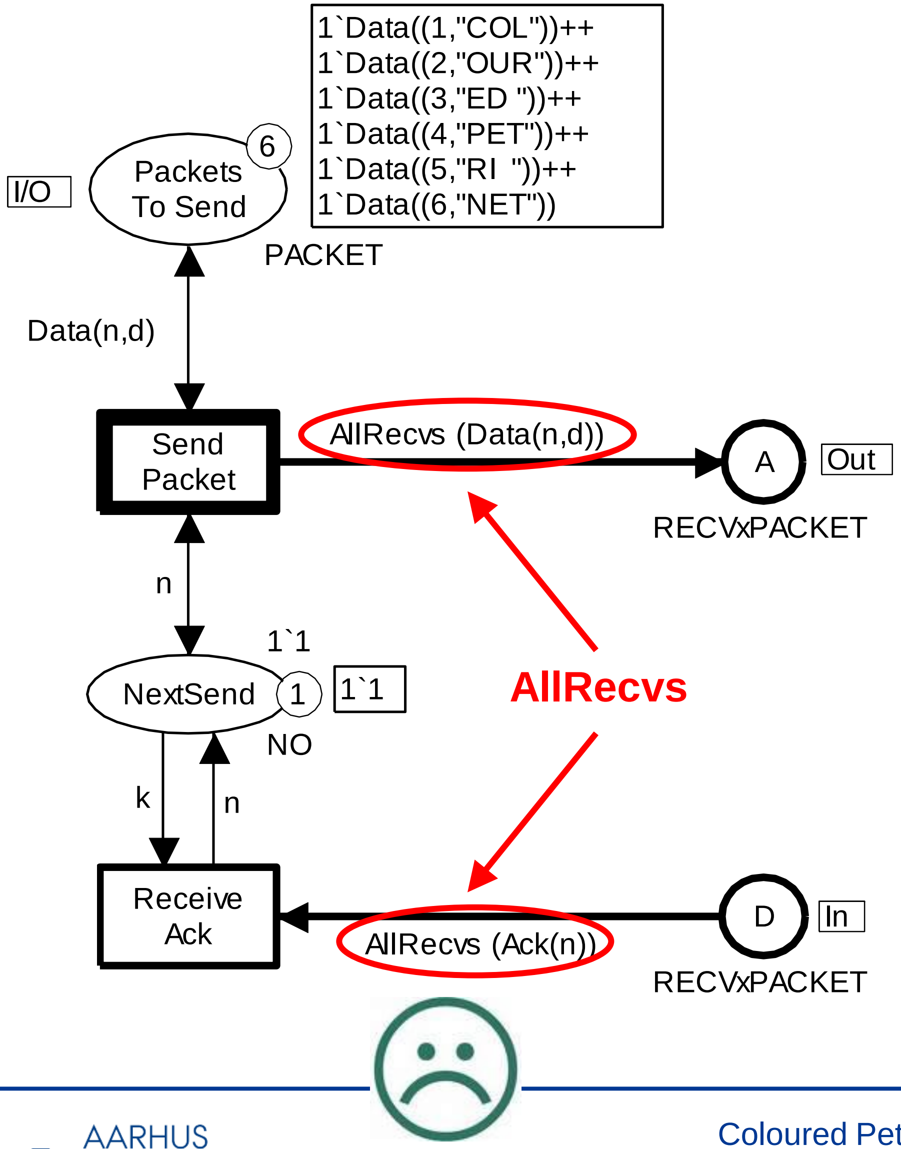
```
AllRecvs ""
```

```
1`(Recv(1),"") ++ 1`(Recv(2),"") ++ 1`(Recv(3),"")
```

Receiver module

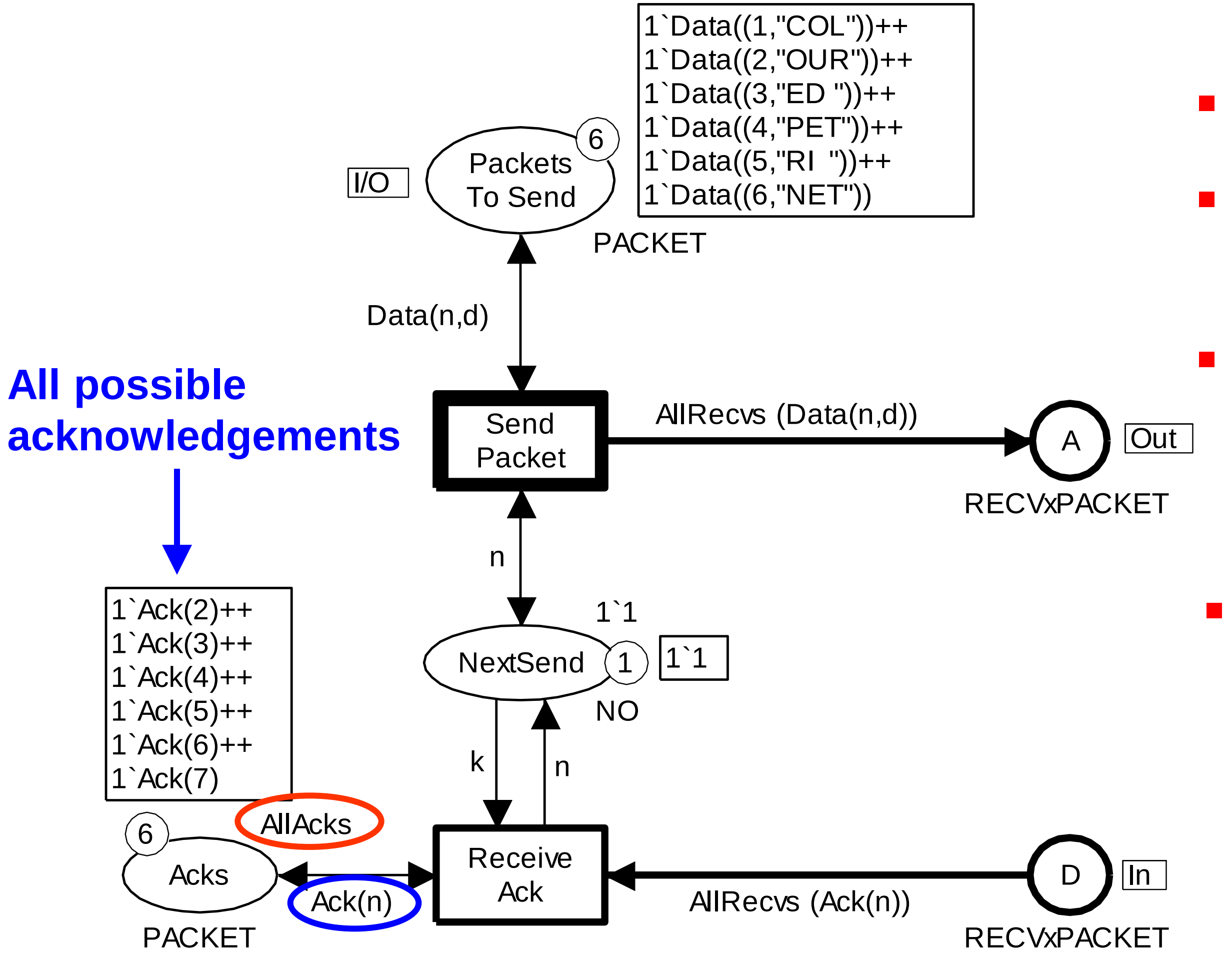


Sender module



- AllRecvs (Ack(n)) is not a pattern – because it involves a function call.
- This implies that the CPN simulator cannot find a binding for the variable n.
- To solve this problem we add a new place Acks.

Modified Sender module



- Ack(n) is a pattern.
- Ack is a data constructor – not a function call.
- Hence the CPN simulator can find a binding for the variable n.
- The initial marking of Acks is defined by means of the symbolic constant AllAcks.

Definition of AllAcks

```
val AllAcks = List.map (fn Data(n,_) => Ack(n+1)) AllPackets;
```

Anonymous function

All packets

Curried library function:
Applies a function
(given as first argument)
on all elements in a list
(given as second argument)

```
1‘Data((1,"COL")) ++ 1‘Data((2,"OUR")) ++
1‘Data((3,"ED ")) ++ 1‘Data((4,"PET")) ++
1‘Data((5,"RI ")) ++ 1‘Data((6,"NET"))
```

```
[Data((1,"COL")),Data((2,"OUR")),
 Data((3,"ED ")),Data((4,"PET")),
 Data((5,"RI ")),Data((6,"NET"))]
```

```
[Ack(2),Ack(3),Ack(4),Ack(5),Ack(6),Ack(7)]
```

```
1`Ack(2) ++ 1`Ack(3) ++ 1`Ack(4) ++
1`Ack(5) ++ 1`Ack(6) ++ 1`Ack(7)
```

CPN tools represents multisets as lists

Fusion sets

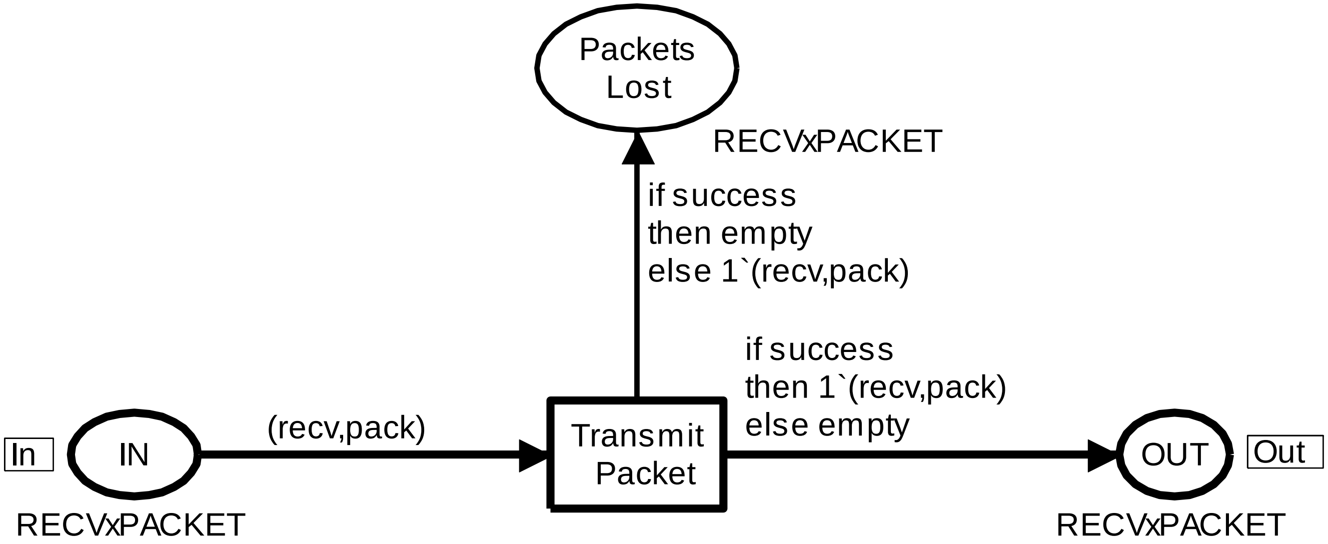
- Modules can exchange tokens via:
  - Port and socket places.
  - Fusion sets.

- Fusion sets allow places in different modules to be "glued" to one compound place — across the hierarchical structure of the model.

- Fusion sets are in some sense similar to global variables known from many programming languages and should therefore be used with care.

Collecting lost packets

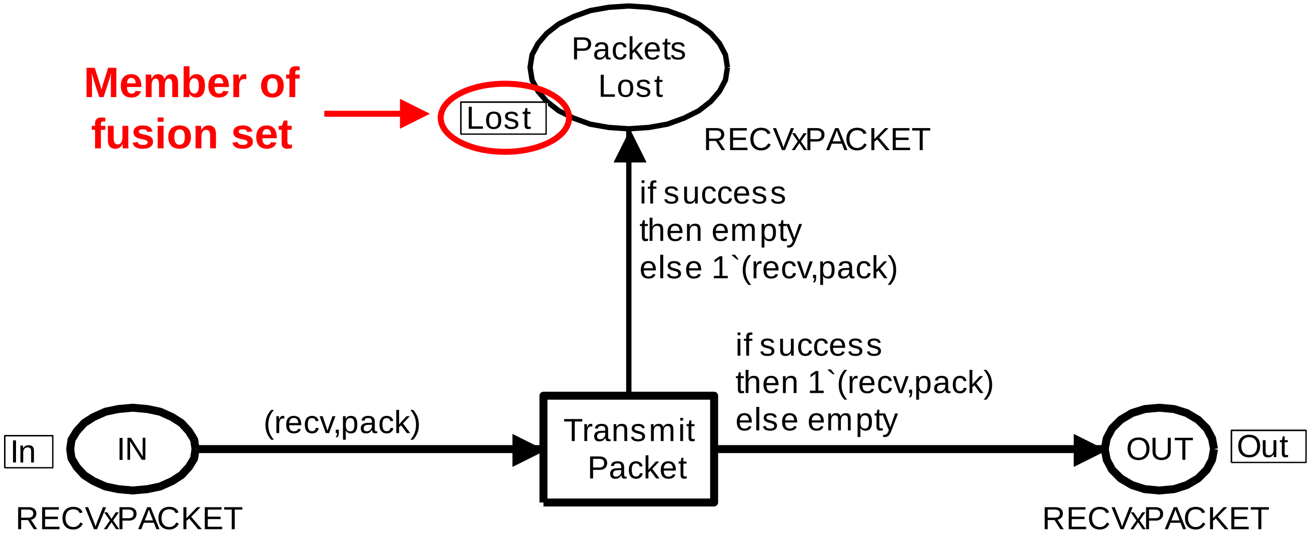
- We collect a copy of the lost packets on place PacketsLost.
- Each instance of the Transmit module collects separately.



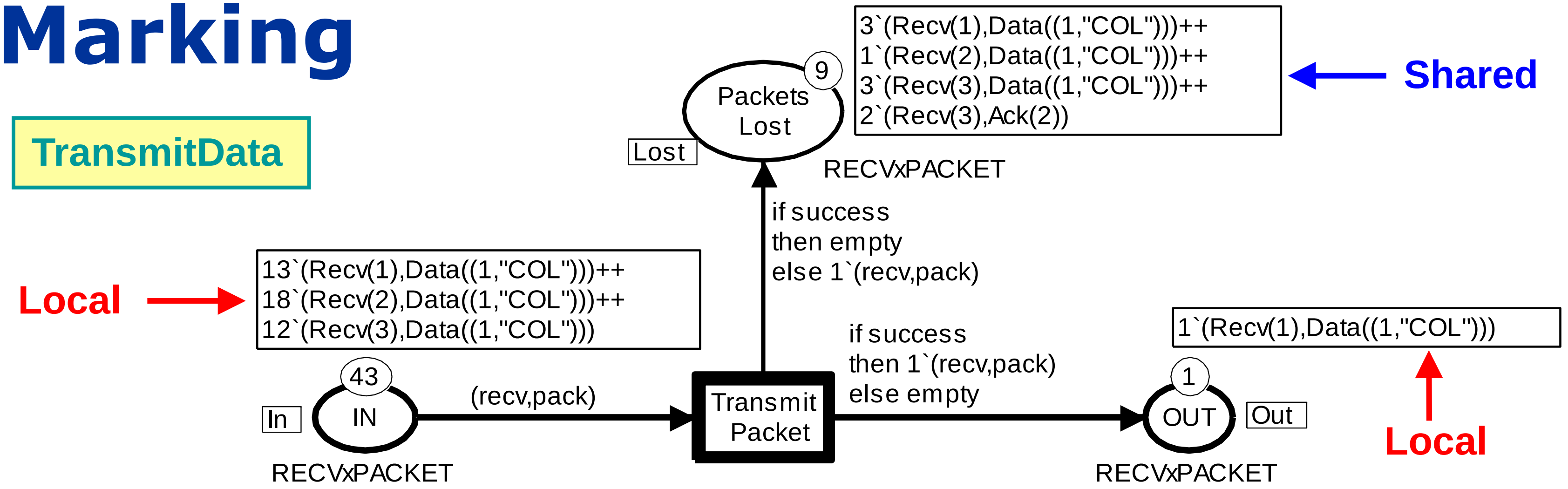
Collecting at a single place

- All instances of the place PacketsLost are “glued together” to become a single compound place – sharing the same marking.

Member of fusion set



Marking



TransmitData

Local

13`(Recv(1),Data((1,"COL")))++
18`(Recv(2),Data((1,"COL")))++
12`(Recv(3),Data((1,"COL")))

43

In

IN

RECVxPACKET

(recv,pack)

Transmit Packet

if success
then empty
else 1`(recv,pack)

Packets Lost

9

Lost

RECVxPACKET

3`(Recv(1),Data((1,"COL")))++
1`(Recv(2),Data((1,"COL")))++
3`(Recv(3),Data((1,"COL")))++
2`(Recv(3),Ack(2))

Shared

if success
then 1`(recv,pack)
else empty

1

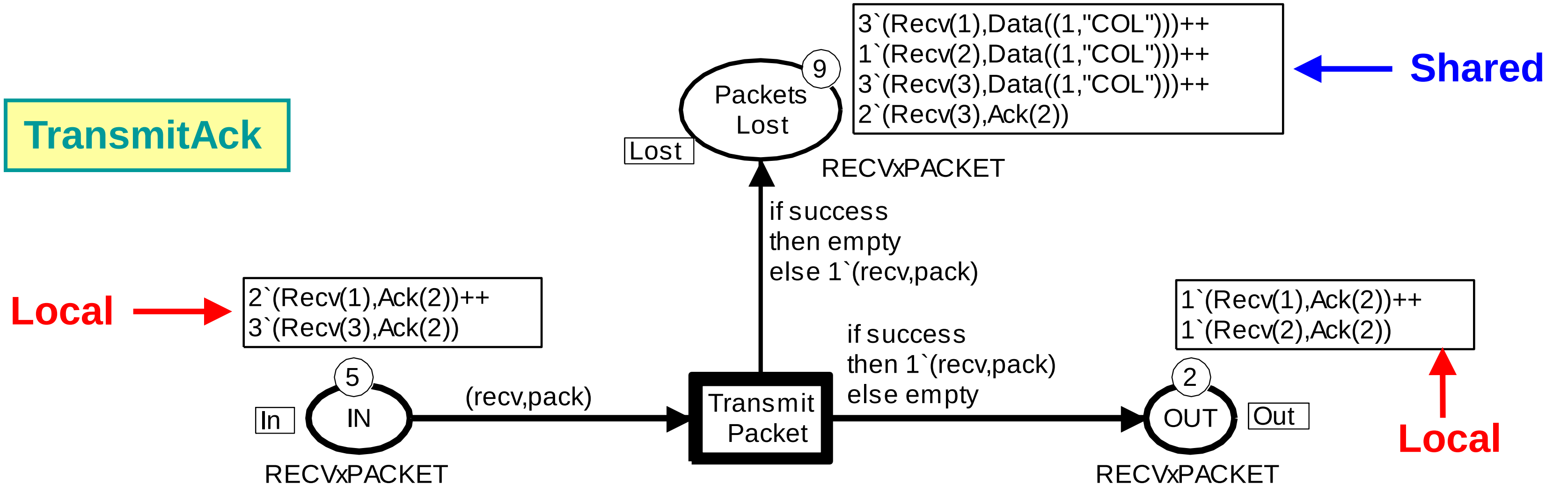
OUT

Out

RECVxPACKET

1`(Recv(1),Data((1,"COL")))

Local



TransmitAck

Local

2`(Recv(1),Ack(2))++
3`(Recv(3),Ack(2))

5

In

IN

RECVxPACKET

(recv,pack)

Transmit Packet

if success
then empty
else 1`(recv,pack)

Packets Lost

9

Lost

RECVxPACKET

3`(Recv(1),Data((1,"COL")))++
1`(Recv(2),Data((1,"COL")))++
3`(Recv(3),Data((1,"COL")))++
2`(Recv(3),Ack(2))

Shared

if success
then 1`(recv,pack)
else empty

2

OUT

Out

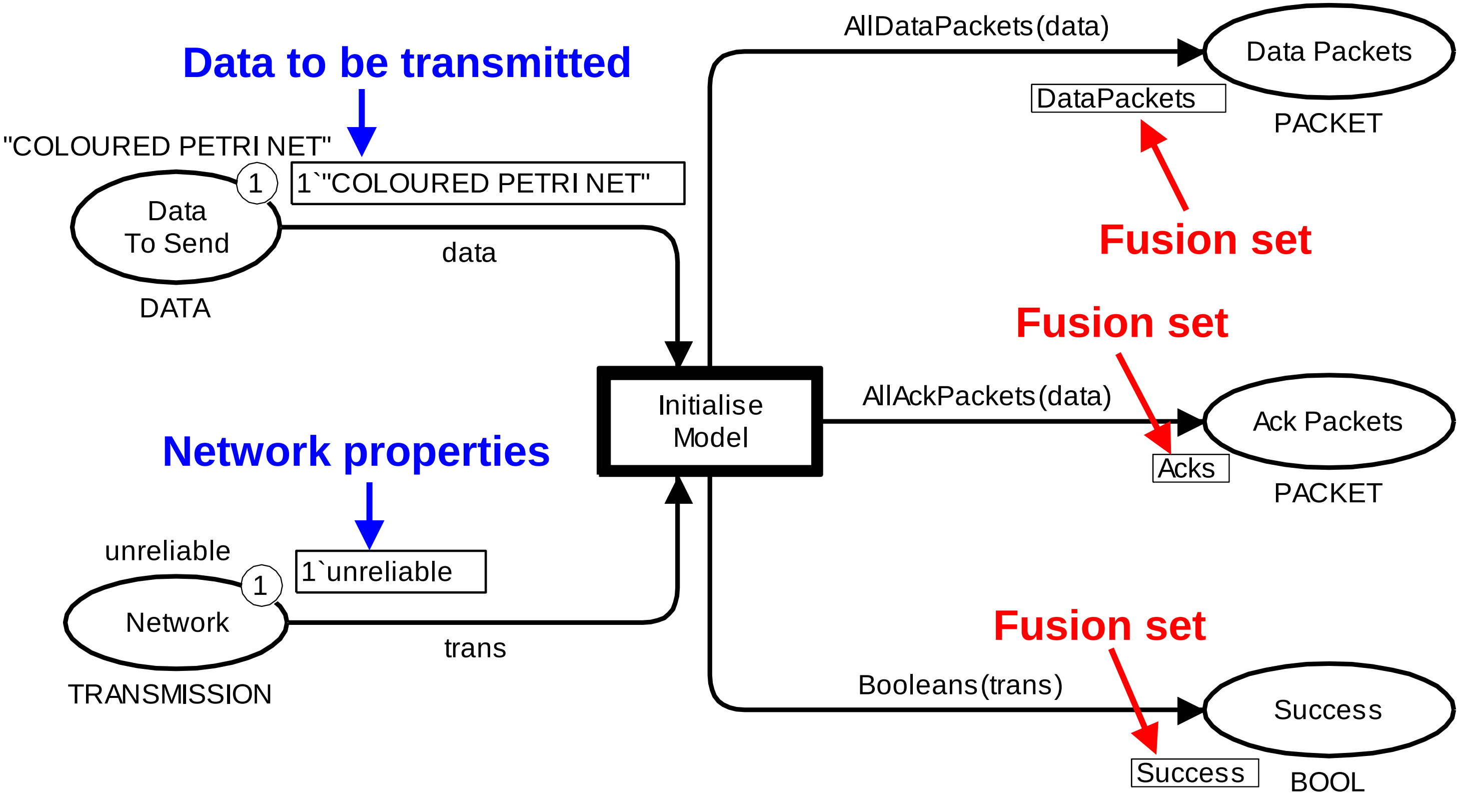
RECVxPACKET

1`(Recv(1),Ack(2))++
1`(Recv(2),Ack(2))

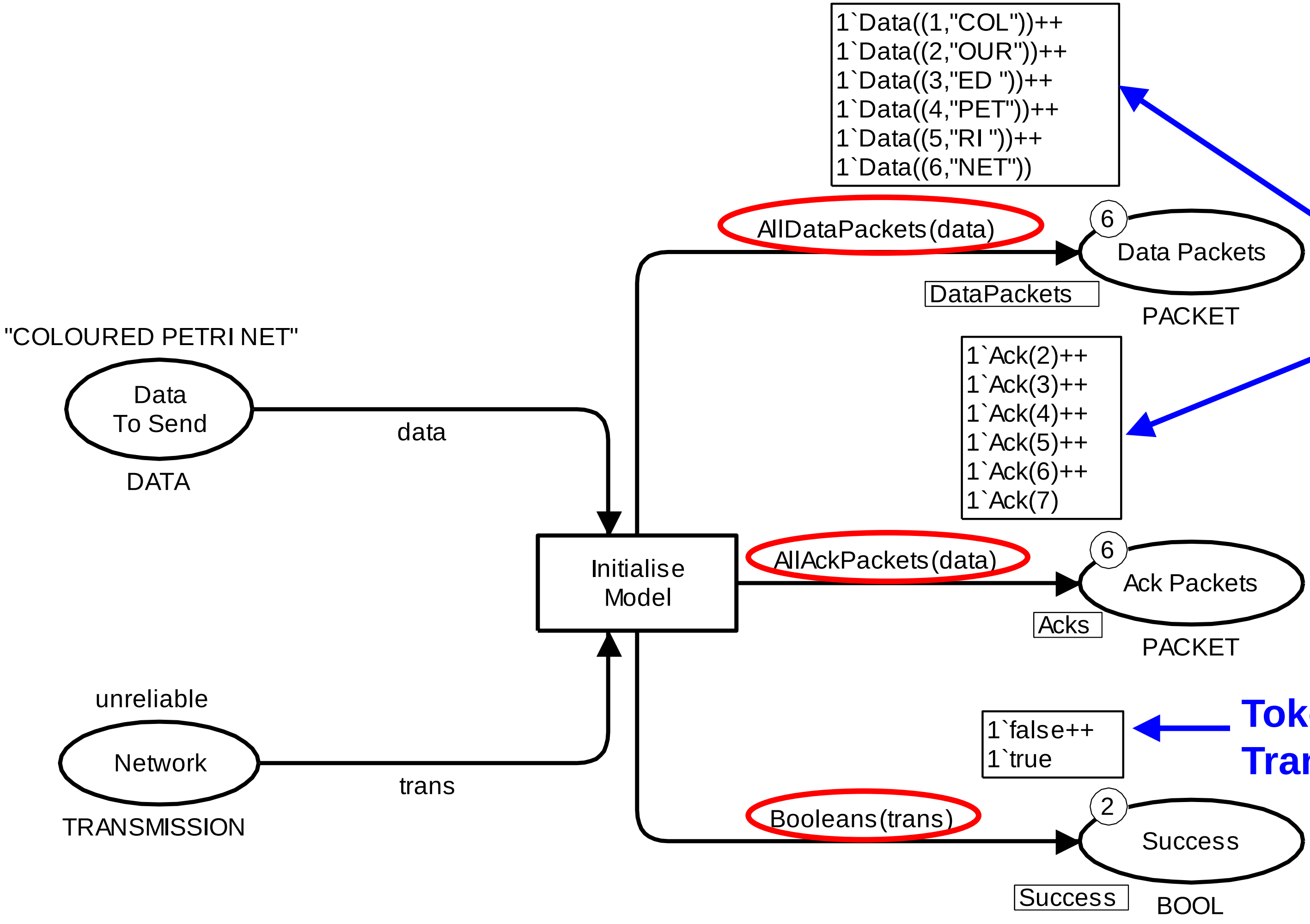
Local

Initialisation module

- Initialisation of the CPN model is done in a specific module.
- Tokens are distributed to the other modules via fusion sets.



Marking after initialisation



Tokens to
Sender module

Tokens to
Transmit module

SplitData

```
val PacketLength = 3;
```

Symbolic constant: max length of each packet

```
fun SplitData (data) =
   let
      val pl = PacketLength;
      fun splitdata (n,data) =
      let
         val dl = String.size (data)
      in
         if dl <= pl
         then [(n,data)]
         else (n,substring (data,0,pl))::
              splitdata (n+1,substring (data,pl,dl-pl))
         end;
      in
         splitdata (1,data)
      end;
```

Local symbolic constant: Max length of each packet

Local auxiliary function

AllDataPackets and AllAckPackets

- Function to initialise place PacketsToSend in the Sender module:

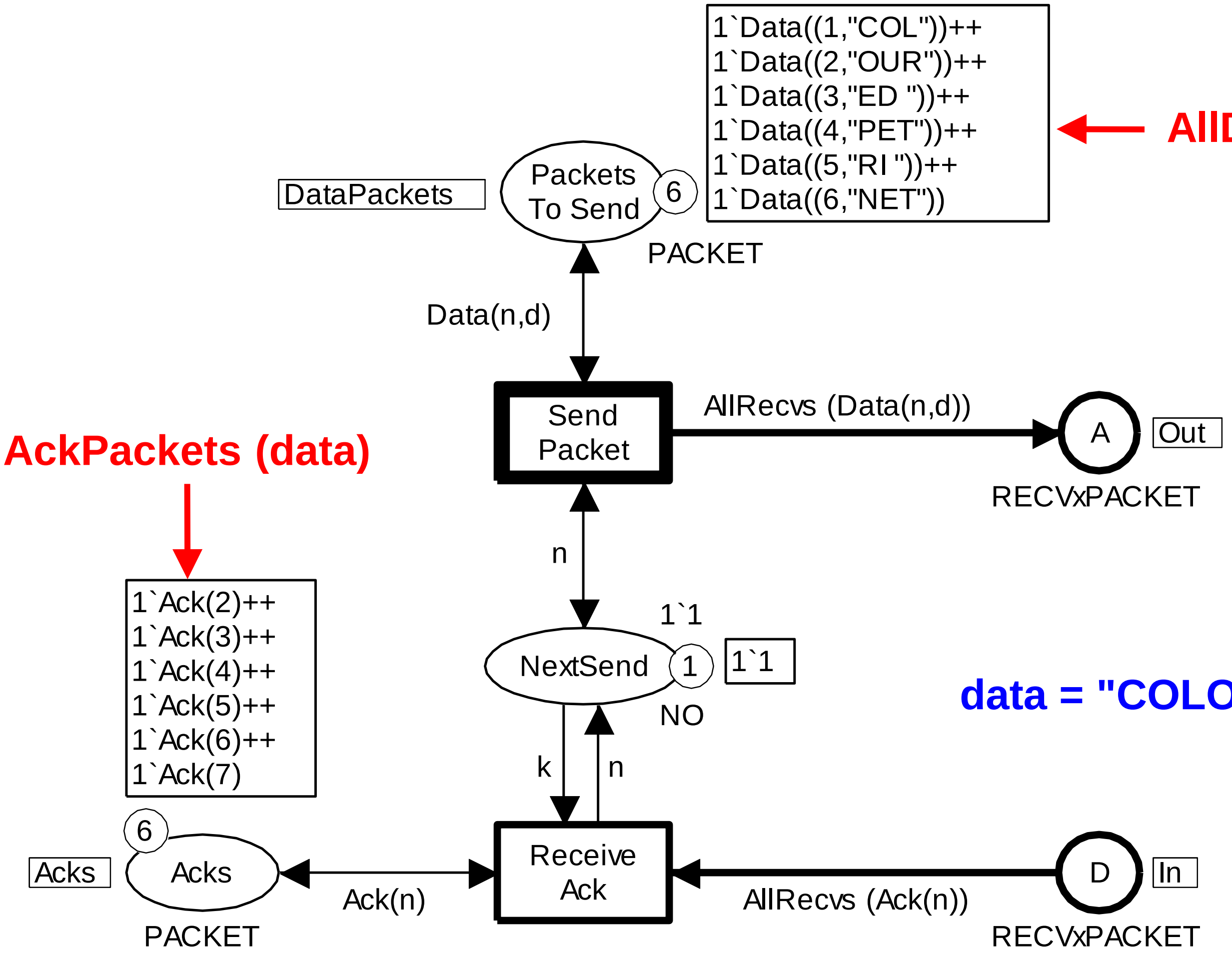
```
fun AllDataPackets (data) =
          (List.map (fn (n,d) => Data(n,d)) (SplitData (data)));
```

```
[(1,"COL"),(2,"OUR"),(3,"ED "),(4,"PET"),(5,"RI "),(6,"NET")]
```

- Function to initialise place Acks in the Sender module:

```
fun AllAckPackets (data) =
          (List.map (fn (n,_) => Ack(n+1)) (SplitData (data)));
```

Marking of Sender module after initialisation



AllDataPackets (data)

AllAckPackets (data)

data = "COLOURED PETRI NET"

Reliable/unreliable transmission

- New colour set:

```
colset TRANSMISSION = with reliable | unreliable;
```

No packets are lost: success can only be bound to true

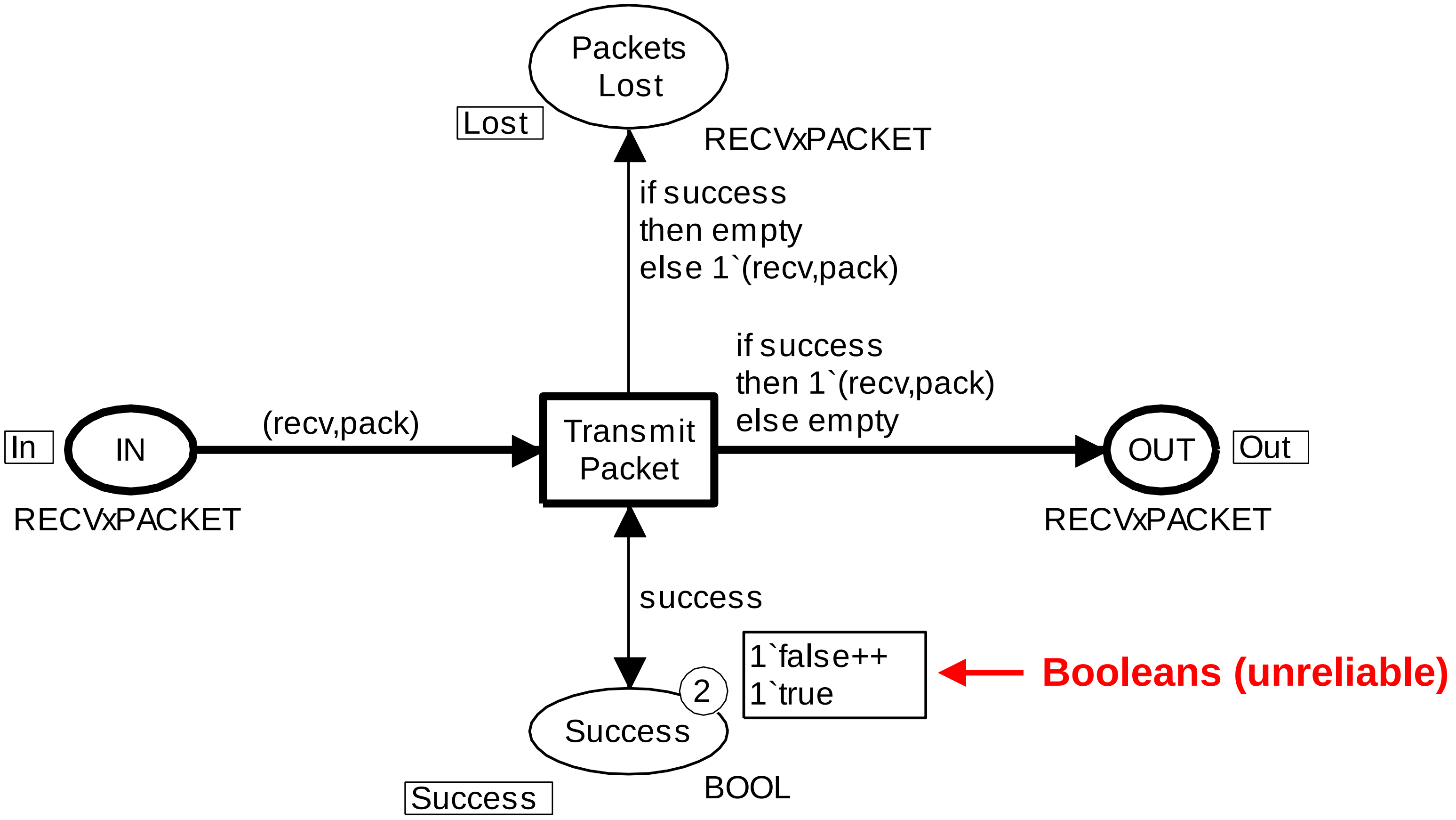
Packets may be lost: success can be bound to either true or false

- Function to initialise place Success in the Transmit module:

```
(* Produces the boolean values to which the
   variable success can be bound *)

fun Booleans reliable   = 1`true
 |  Booleans unreliable = 1`true ++ 1`false;
```

Marking of Transmit module after initialisation

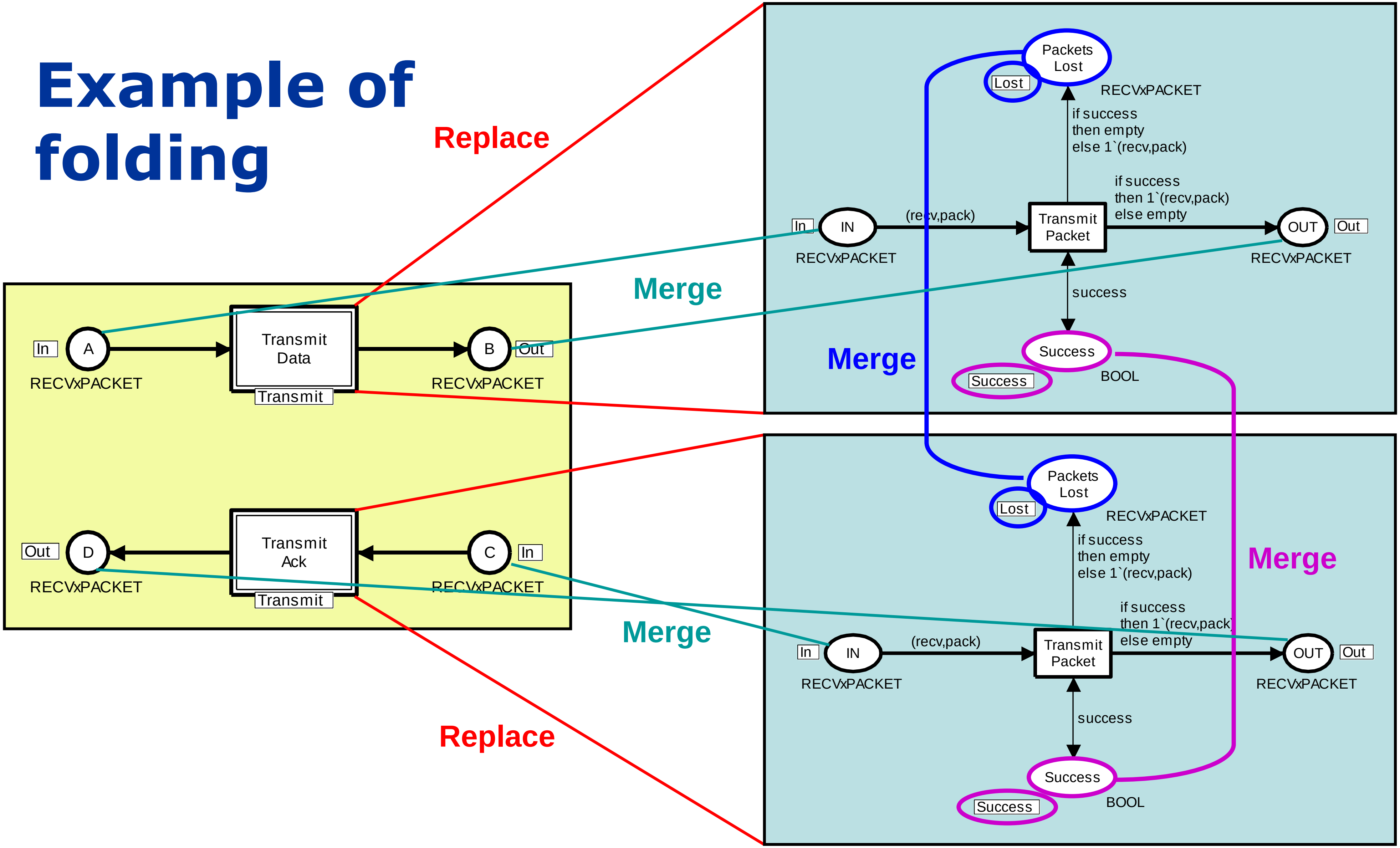


Unfolding hierarchical CPN models

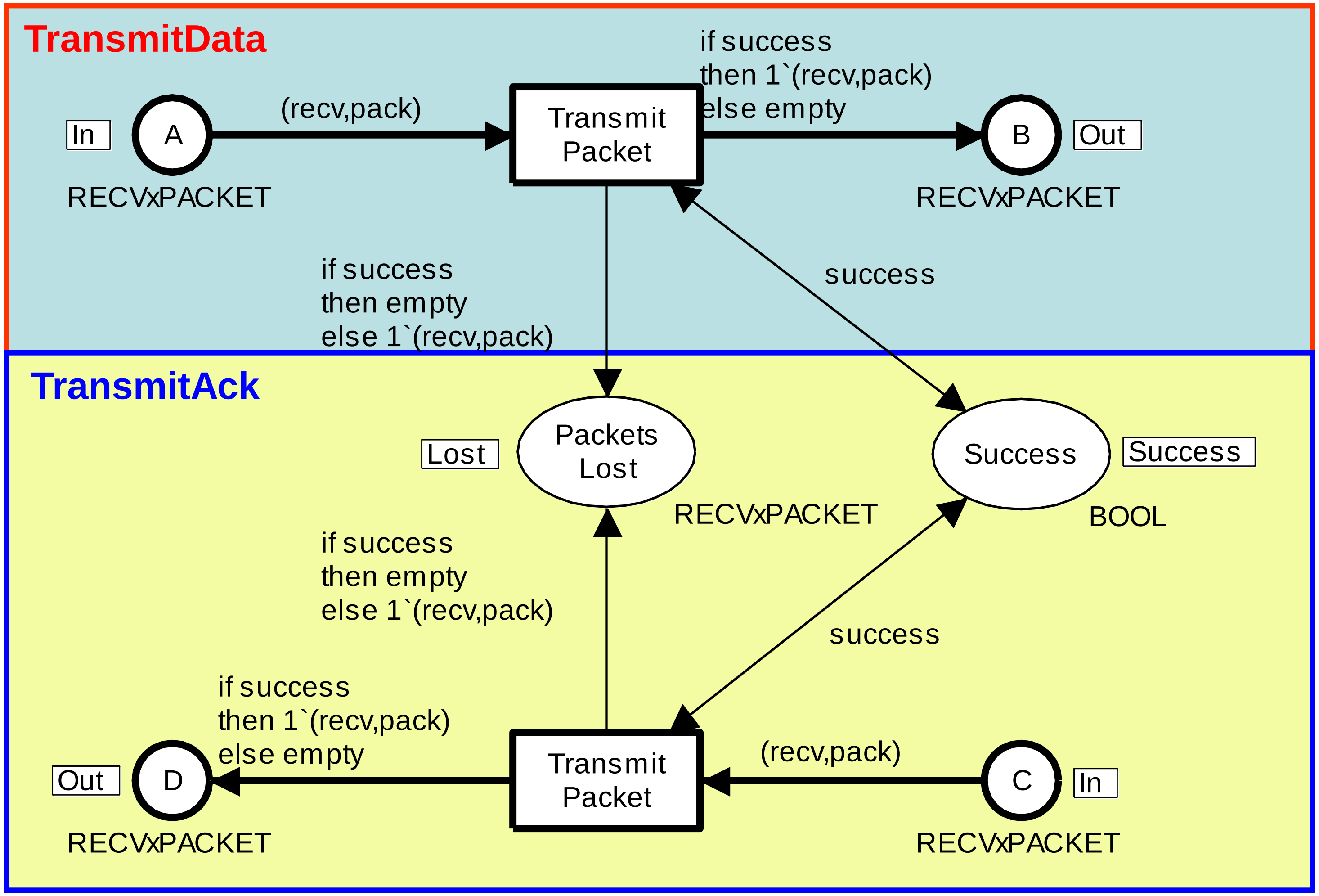
- A hierarchical CPN model can always be unfolded to an equivalent non-hierarchical CPN model

- Three steps:
- Replace each substitution transition with the content of its submodule (related port and socket places are merged into a single place).
- Collect the contents of all prime modules in a single module.
- Merge places which belong to the same fusion set.

Example of folding



Non-hierarchical CPN model



Hierarchical versus non-hierarchical nets

- A hierarchical CPN model can always be transformed into an equivalent non-hierarchical CPN model.
  - In theory, this implies that the hierarchy constructs of CP-nets do not add expressive power to the modelling language.
  - Any system that can be modelled with a hierarchical CPN model can also be modelled with a non-hierarchical CPN model.
  - In practice, the hierarchy constructs are very important.
  - They make it possible to structure large models and thereby cope with the complexity of large systems.
  - Hence they add modelling power.

Questions