

U O W

Software Requirements, Specifications and Formal Methods

Dr. Shixun Huang



UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Coloured Petri Net



UNIVERSITY
OF WOLLONGONG
AUSTRALIA

History: birth of high-level nets

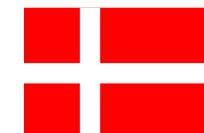
- The first successful kind of **high-level Petri nets** was called **Predicate/Transition Nets** (Pr/T-nets).
- PrT-nets was developed by **Hartmann Genrich** and **Kurt Lautenbach**.
 - The first paper was presented at a conference on Semantics of Concurrent Computation, 1979.
 - The work was partly based on earlier work:
 - Transition nets with coloured tokens,
Kurt Lautenbach & M. Schifflers 1977.
 - Transition nets with complex conditions,
Robert Shapiro 1979.

High-level Petri nets

- The invention of **Pr/T-nets** was the first step towards the kind of **high-level Petri nets** that we know today:
 - **Tokens** can be **distinguished** from each other and hence they are said to be **coloured**.
 - **Transitions** can occur in many different ways depending on the **token colours** of the available input tokens.
 - **Arc expressions** and **guards** are used to specify **enabling conditions** and the effects of **transition occurrences**.

What is a Coloured Petri Net?

- *Modelling language* for systems where **synchronisation**, **communication**, and **resource sharing** are important.
- Combination of *Petri Nets* and *Programming Language*.
 - *Control structures*, **synchronisation**, **communication**, and **resource sharing** are described by *Petri Nets*.
 - *Data* and **data manipulations** are described by *functional programming language*.
- CPN models are **validated** by means of **simulation** and **verified** by means of **state spaces** and **place invariants**.
- Coloured Petri Nets is developed at *University of Aarhus, Denmark* over the last 25 years.



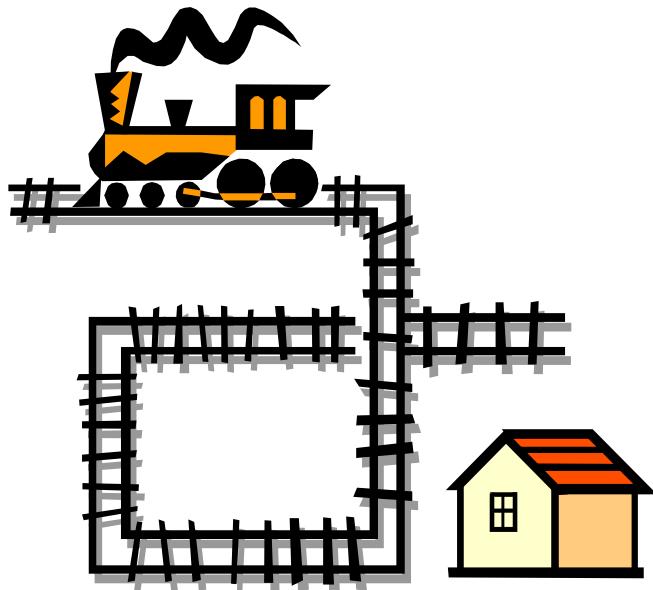
High-level Petri nets

- The relationship between *CP-nets* and *ordinary Petri nets* (PT-nets) is *analogous* to the relationship between *high-level programming languages* and *assembly code*.
 - In *theory*, the two levels have exactly the same *computational power*.
 - In *practice*, high-level languages have much more *modelling power* – because they have better structuring facilities, e.g., *types* and *modules*.
- Several other kinds of *high-level Petri Nets* exist. However, *Coloured Petri Nets* is the most widely used – in particular for *practical work*.

What are models used for?

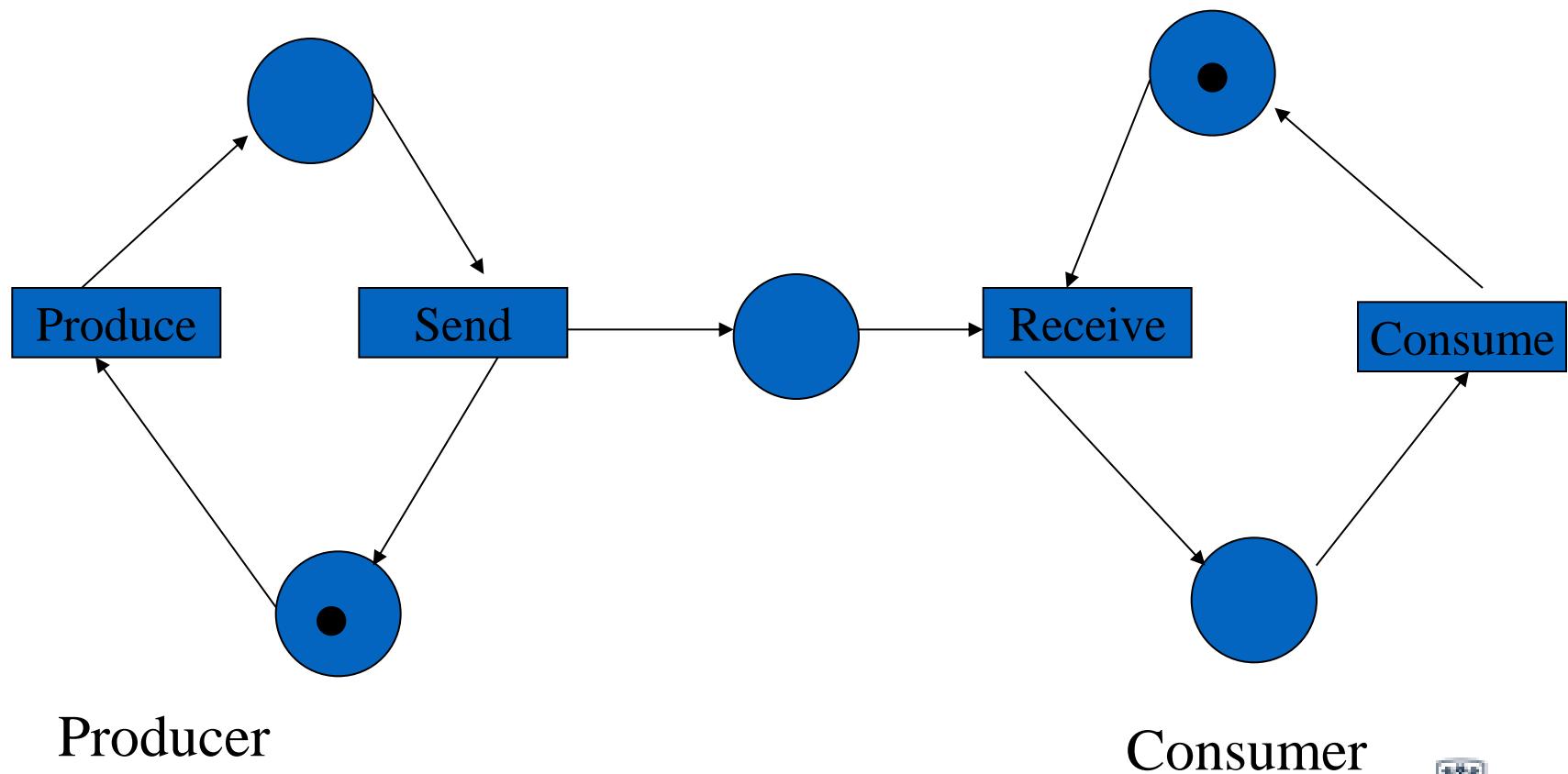
- Static Model
 - Modular architecture of systems
 - Data model
 - Class diagrams
- Dynamic Models
 - Behaviour of systems
 - Exchange of messages
 - State changes

Why do we make models?

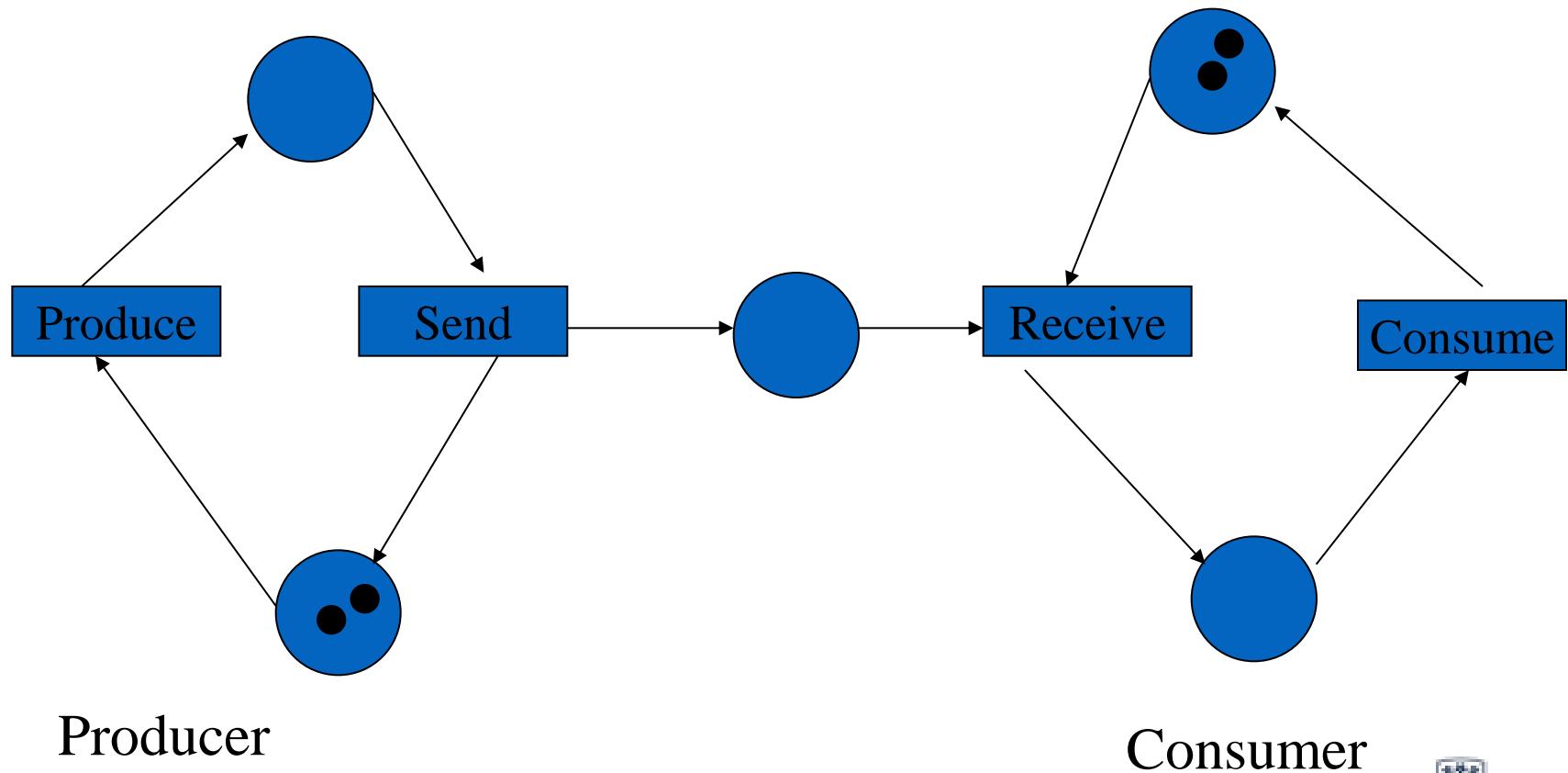


- We make *models* to:
 - *Learn new things* about a system.
 - To check that the system design has certain *expected properties*.
- *CPN models* are *dynamic*:
 - They can be *executed* on a *computer*.
 - This allows us to play and *investigate* different *scenarios*.

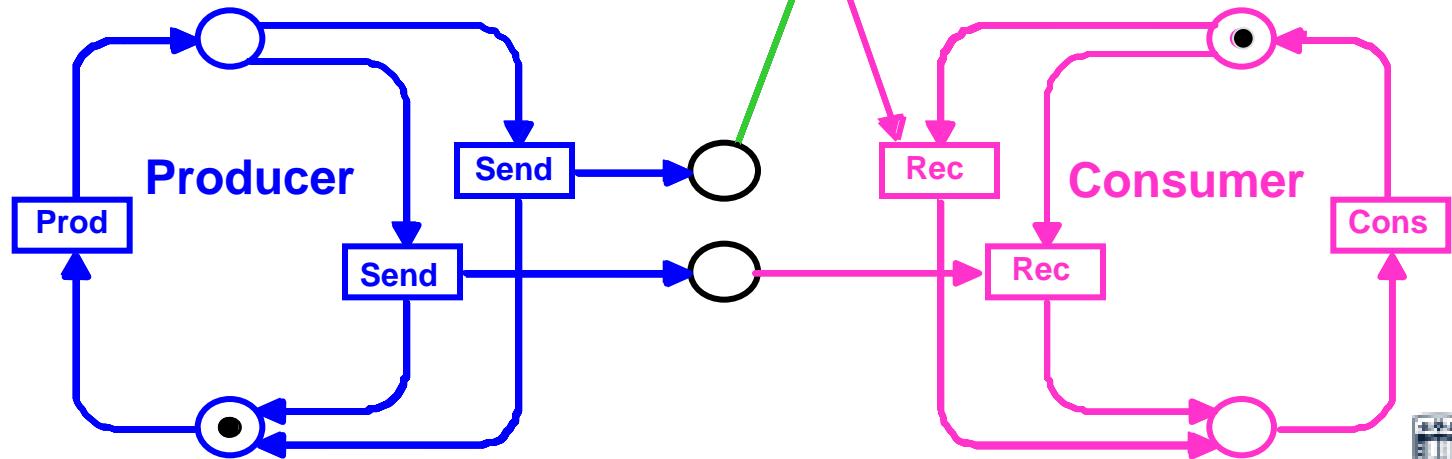
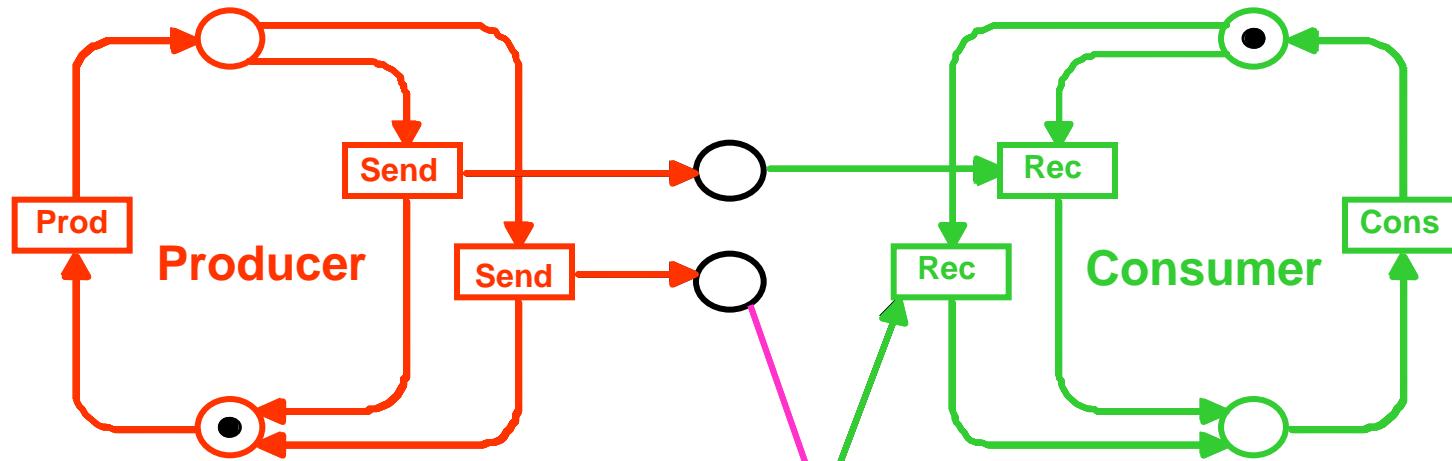
Produce/Consumer Problem (1 to 1)



Produce/Consumer Problem (2 to 2)



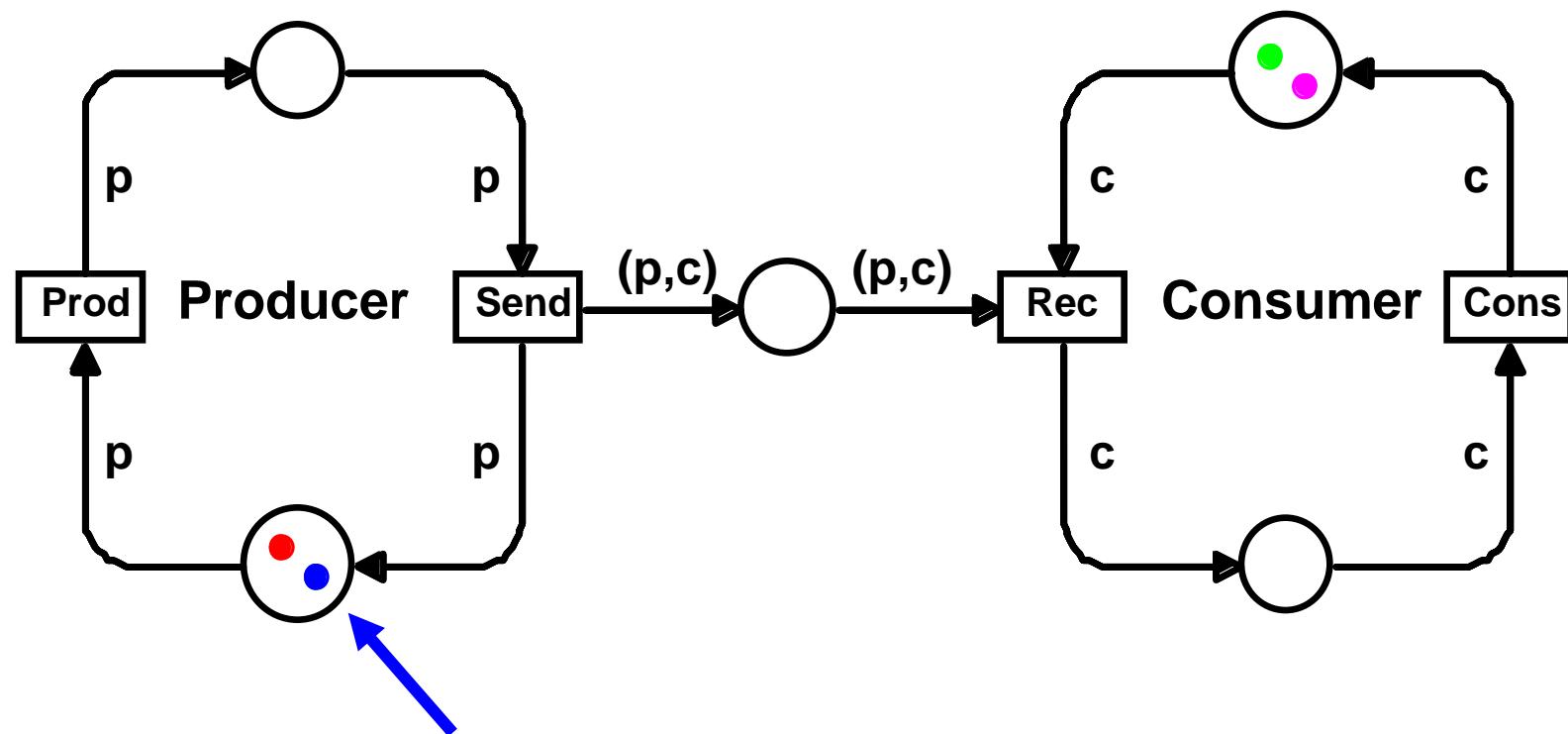
Repeated net structure



Predicate/Transition net

$D = \{ \text{red, blue, green, purple} \}$

var p,c : D

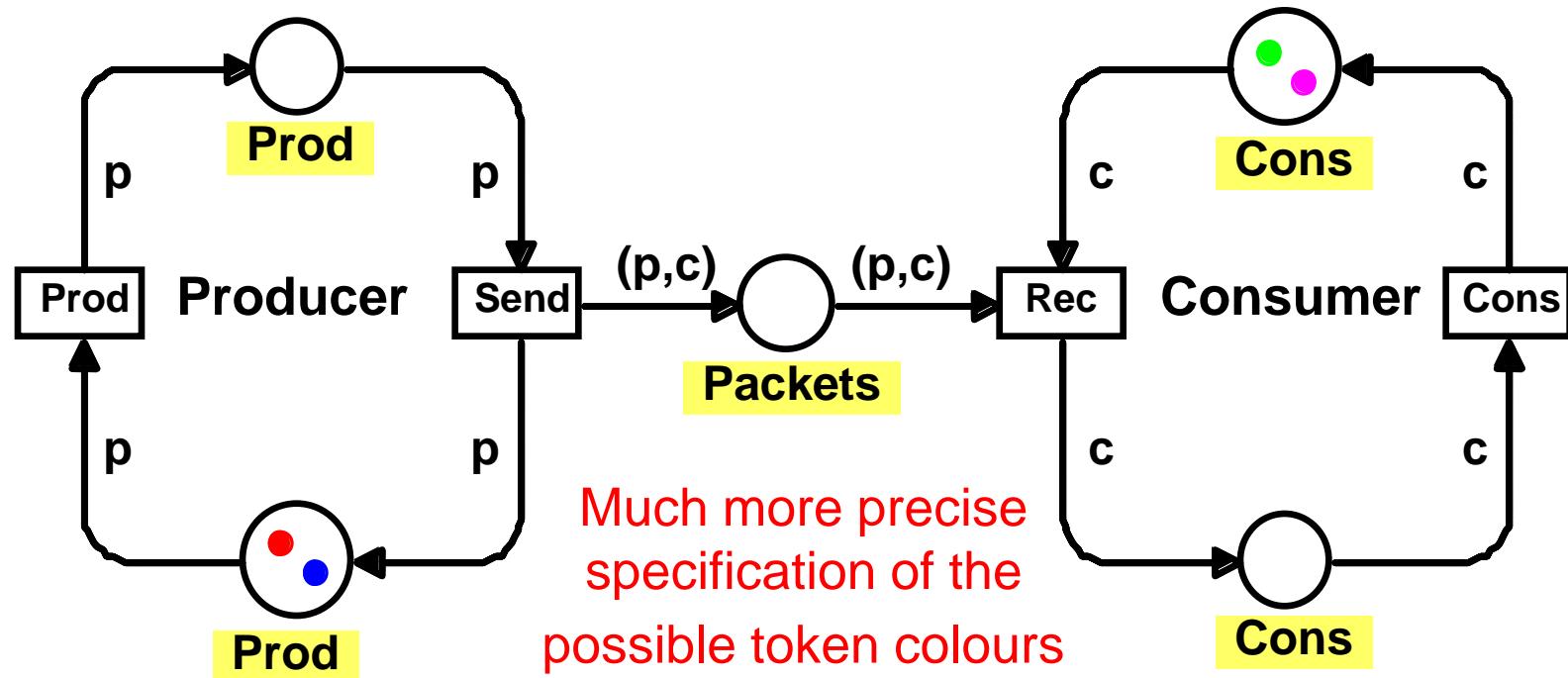


Each token carries a data value
It is coloured !!!

Possible token colours:
 $D, D \times D, D \times D \times D, \dots$



Coloured Petri Nets



colset Prod = { red, blue }

colset Cons = { green, purple }

colset Packets = product Prod * Cons

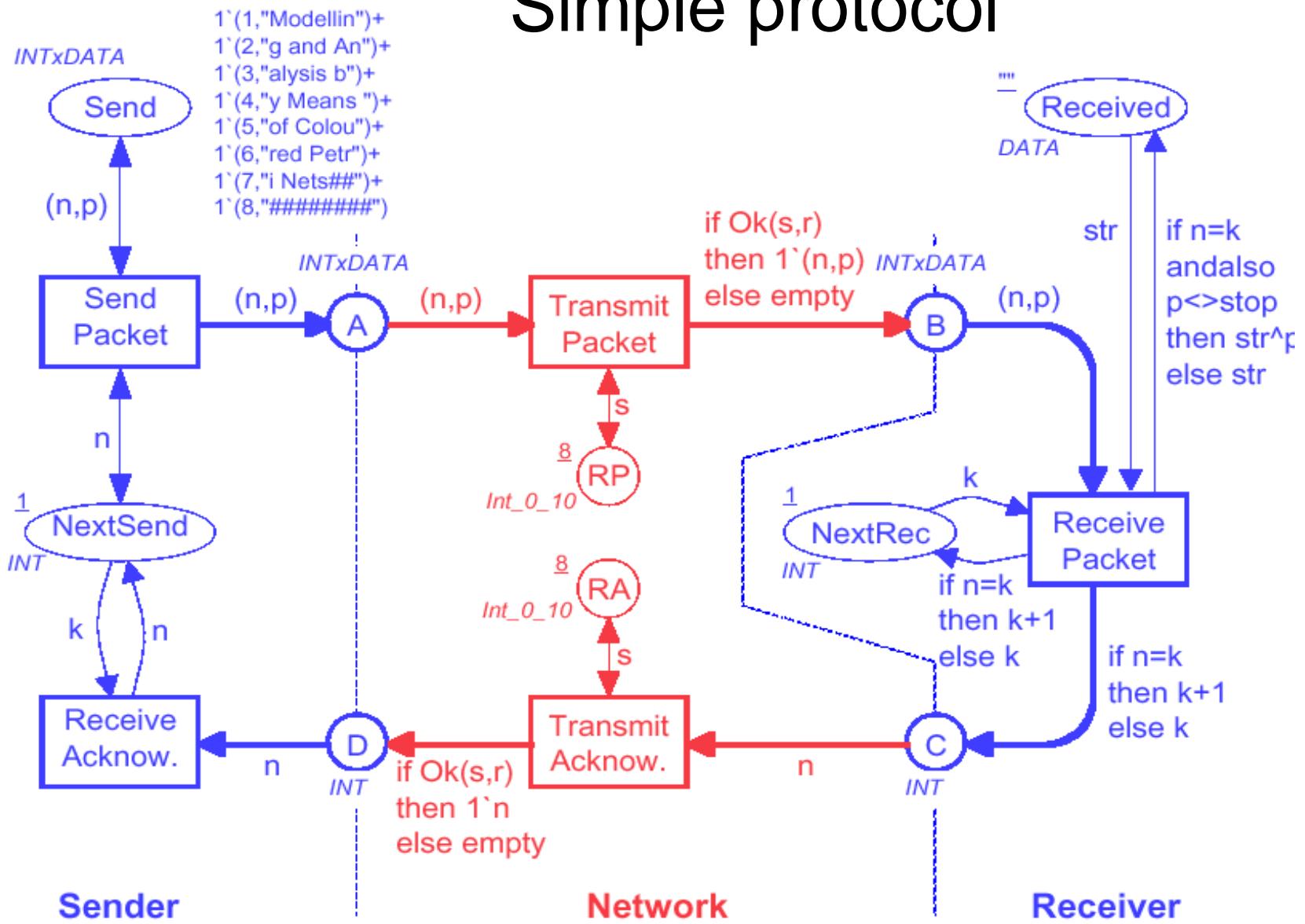
var p : Prod

var c : Cons

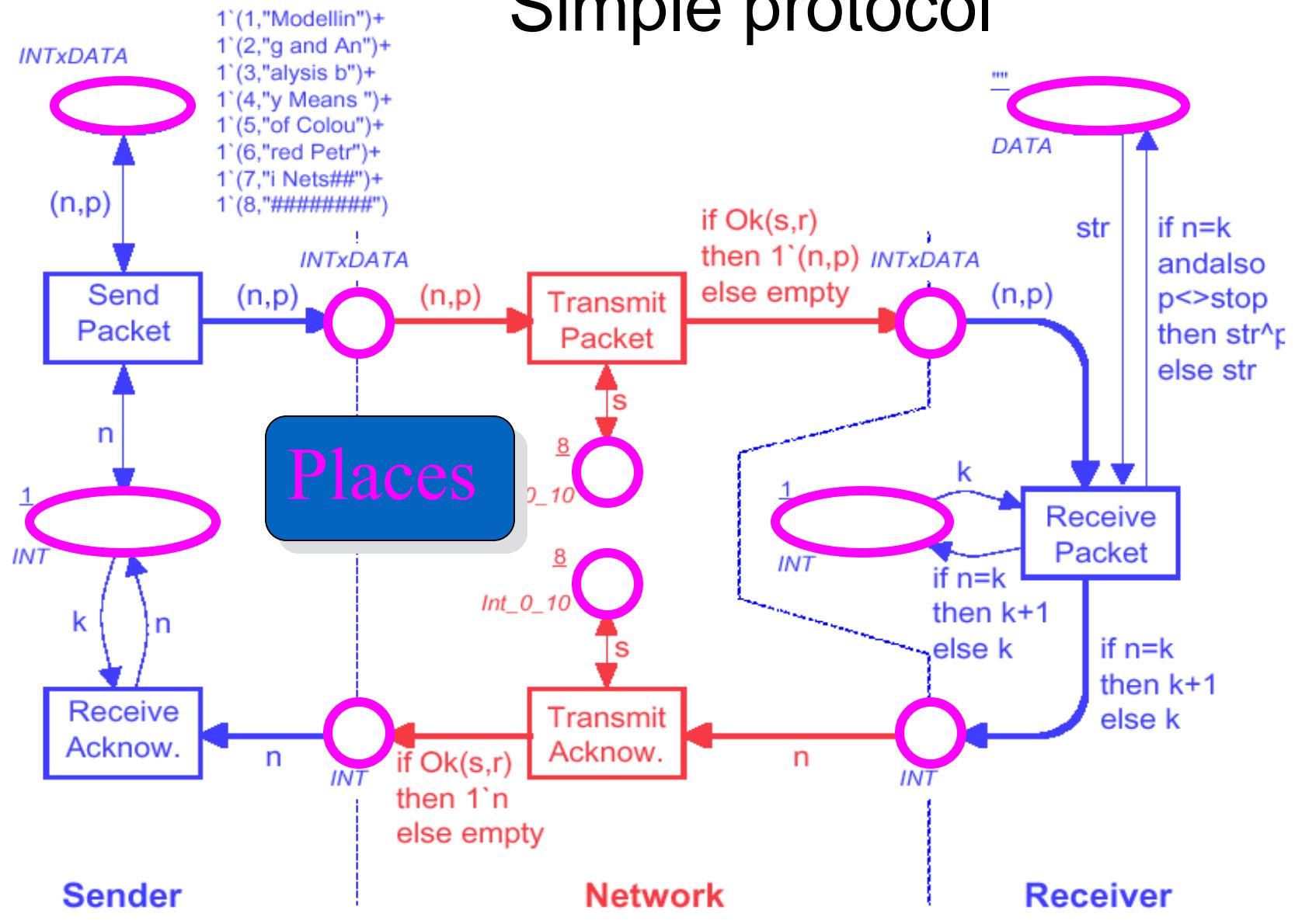
Colour sets = Types

- We use **data types** to specify the **kinds of tokens** which we allow on the **individual places**.
- Types can be **arbitrarily complex**:
 - **Atomic** (e.g., integers, strings, Booleans and enumerations).
 - **Structured** (e.g., products, records, unions, lists, and subsets).
- The use of **types** allows us to make more **readable** descriptions with **mnemonics type names** such as:
 - **PROD, CONS, PACKETS**
- We also get more **correct** descriptions.
 - Automatic **type checking** of **arc expressions**.

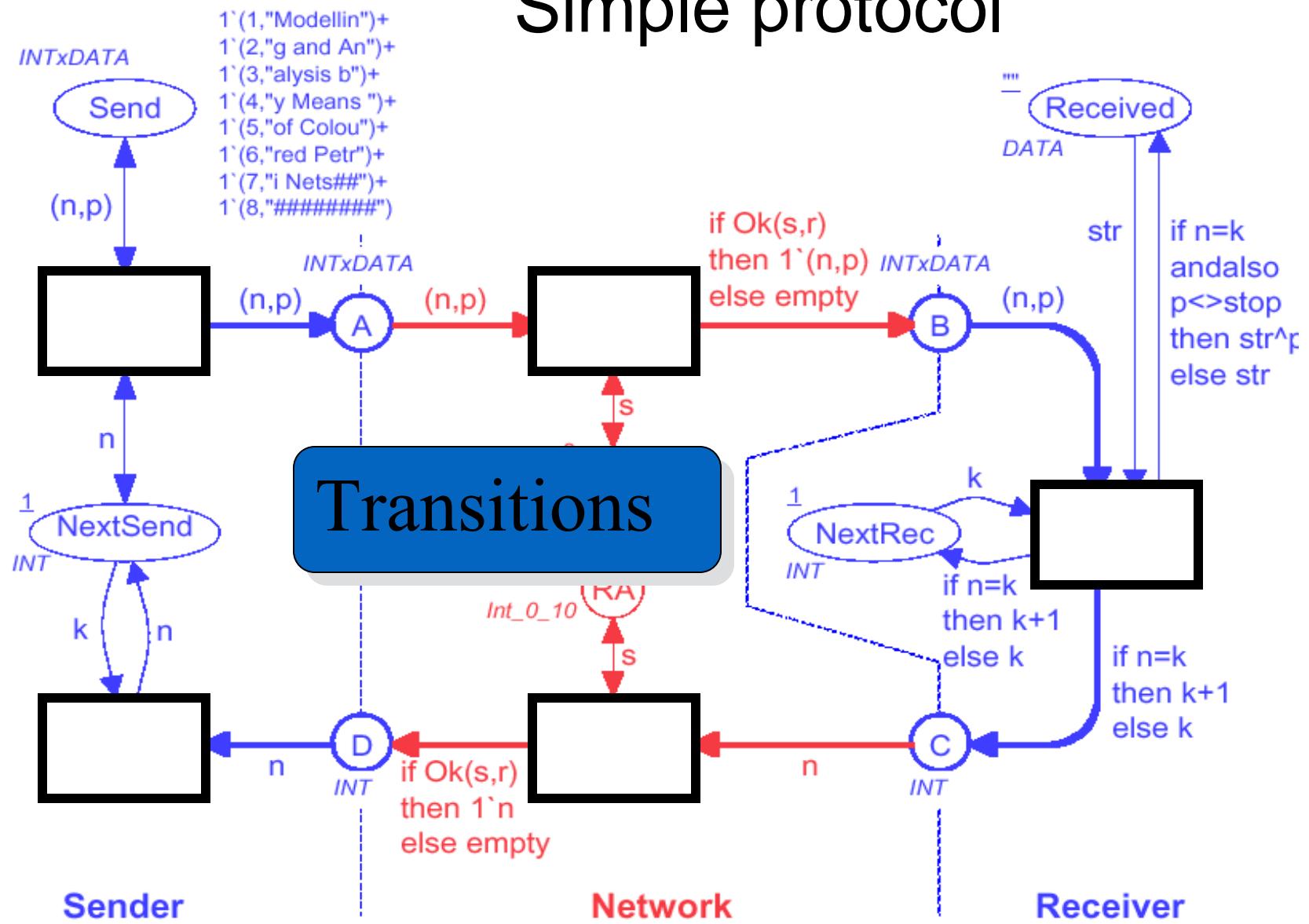
Simple protocol



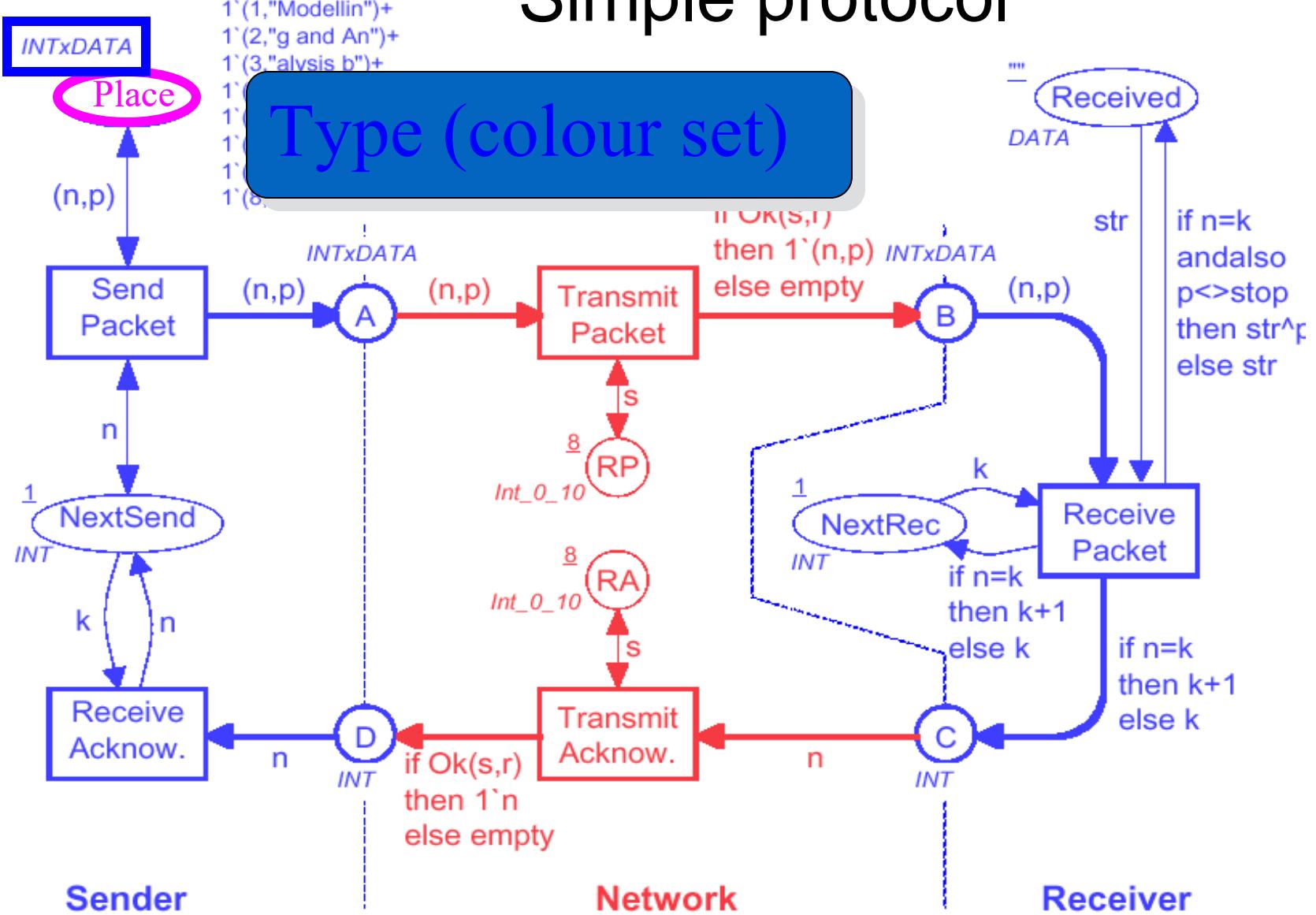
Simple protocol



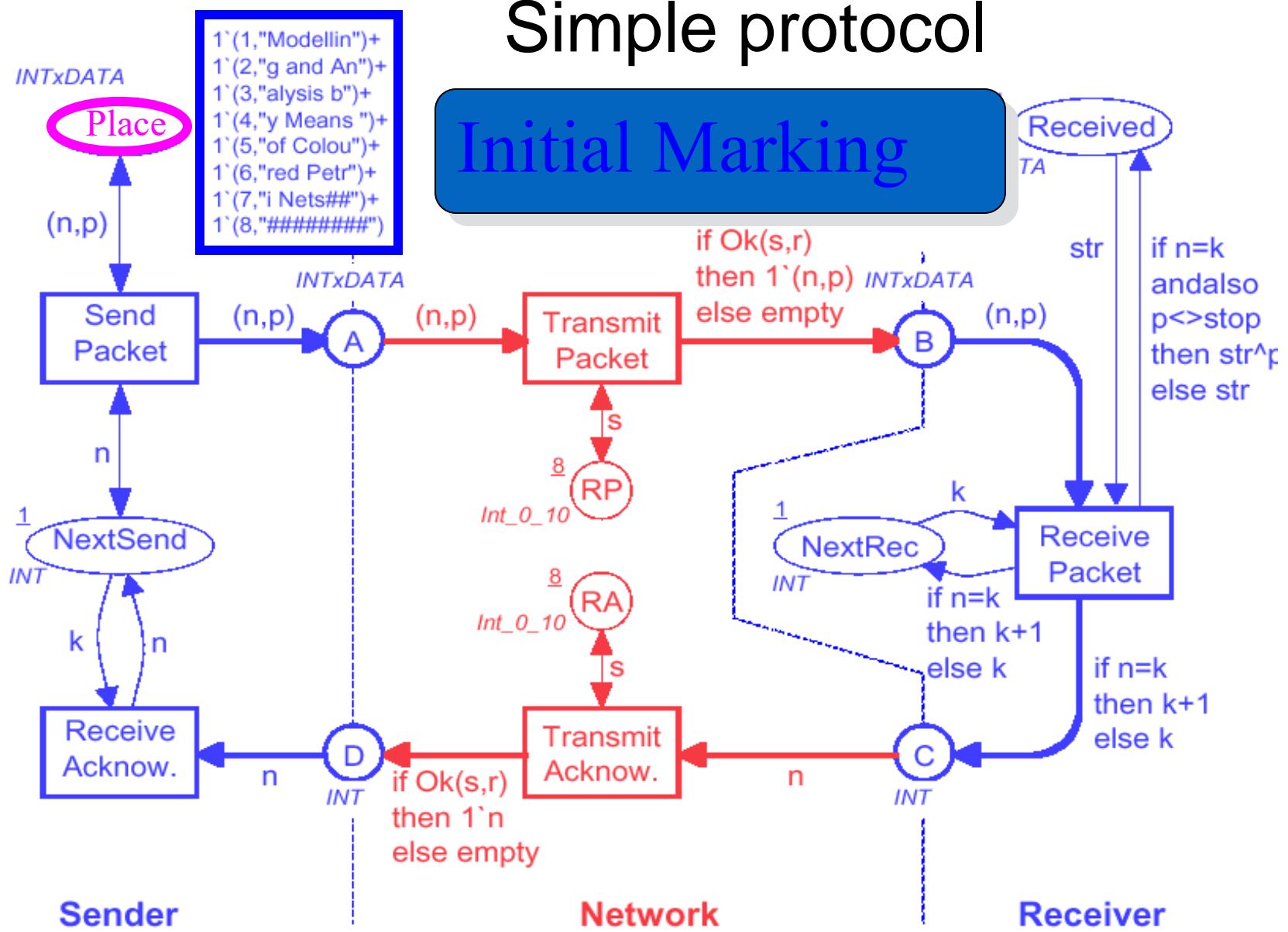
Simple protocol



Simple protocol

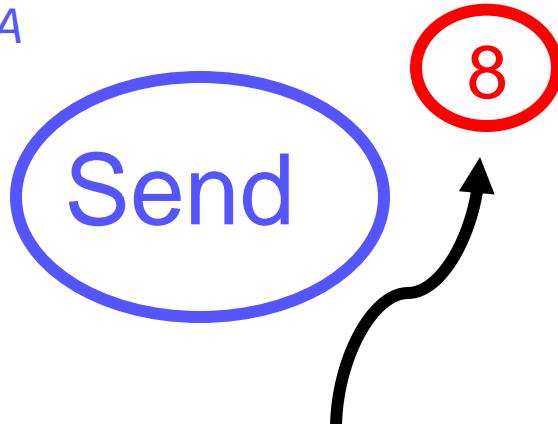


Simple protocol



Marking of Send

INTxDATA

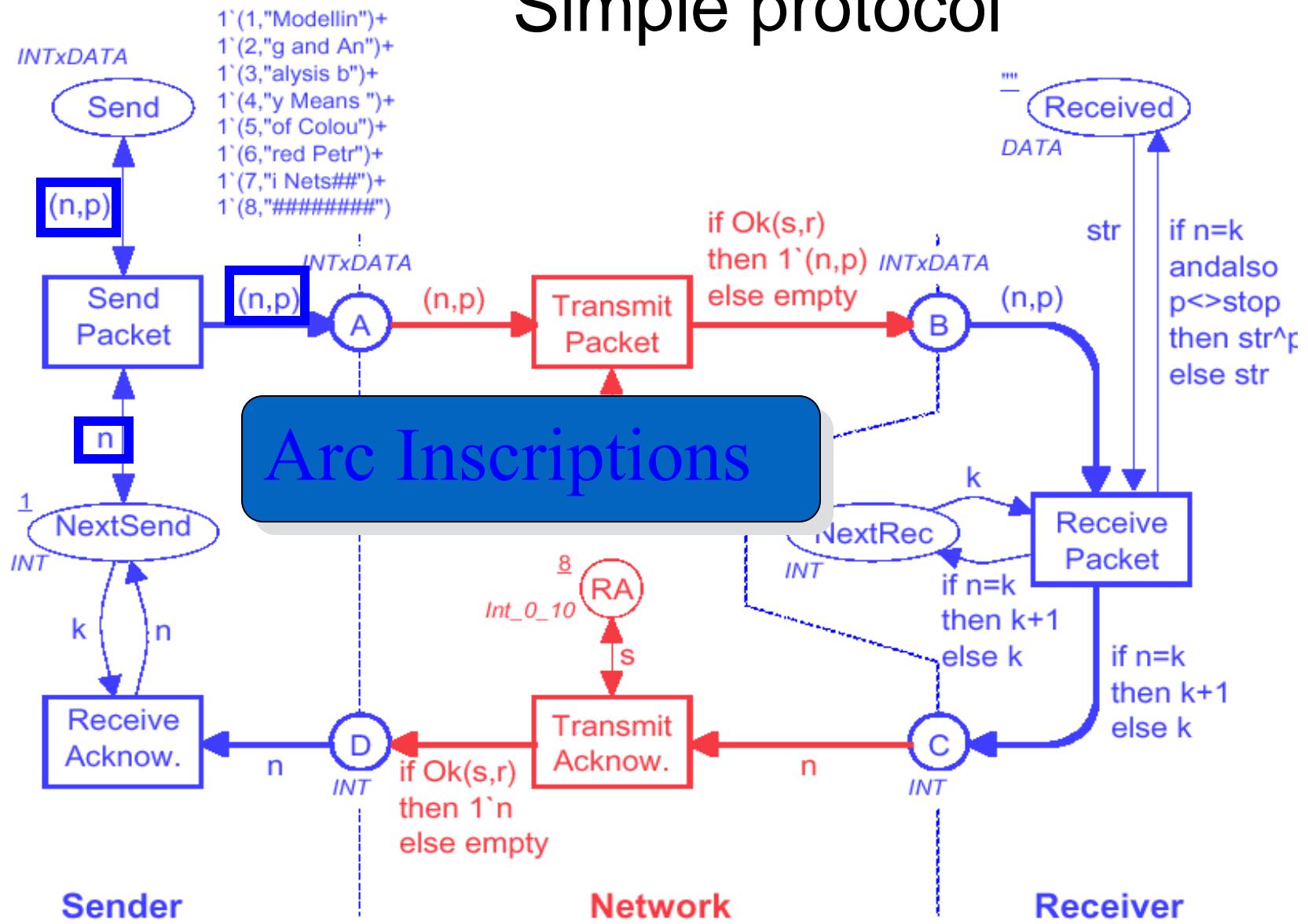


Number of tokens

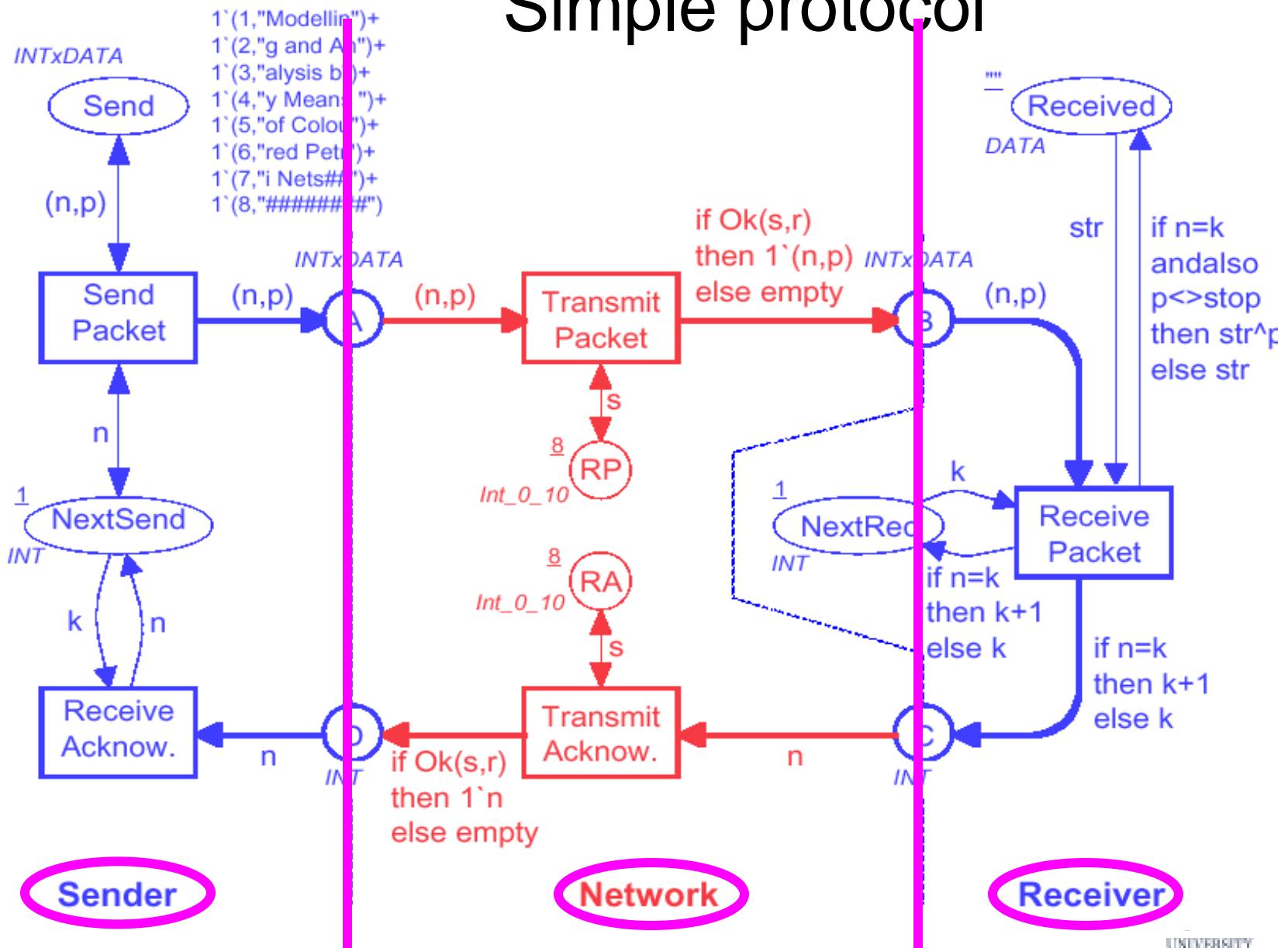
```
1 ` (1,"Modellin") +
1 ` (2,"g and An") +
1 ` (3,"alysis b") +
1 ` (4,"y Means ") +
1 ` (5,"of Colou") +
1 ` (6,"red Petr") +
1 ` (7,"i Nets##") +
1 ` (8,"#####")
```

Multi-set of token colours

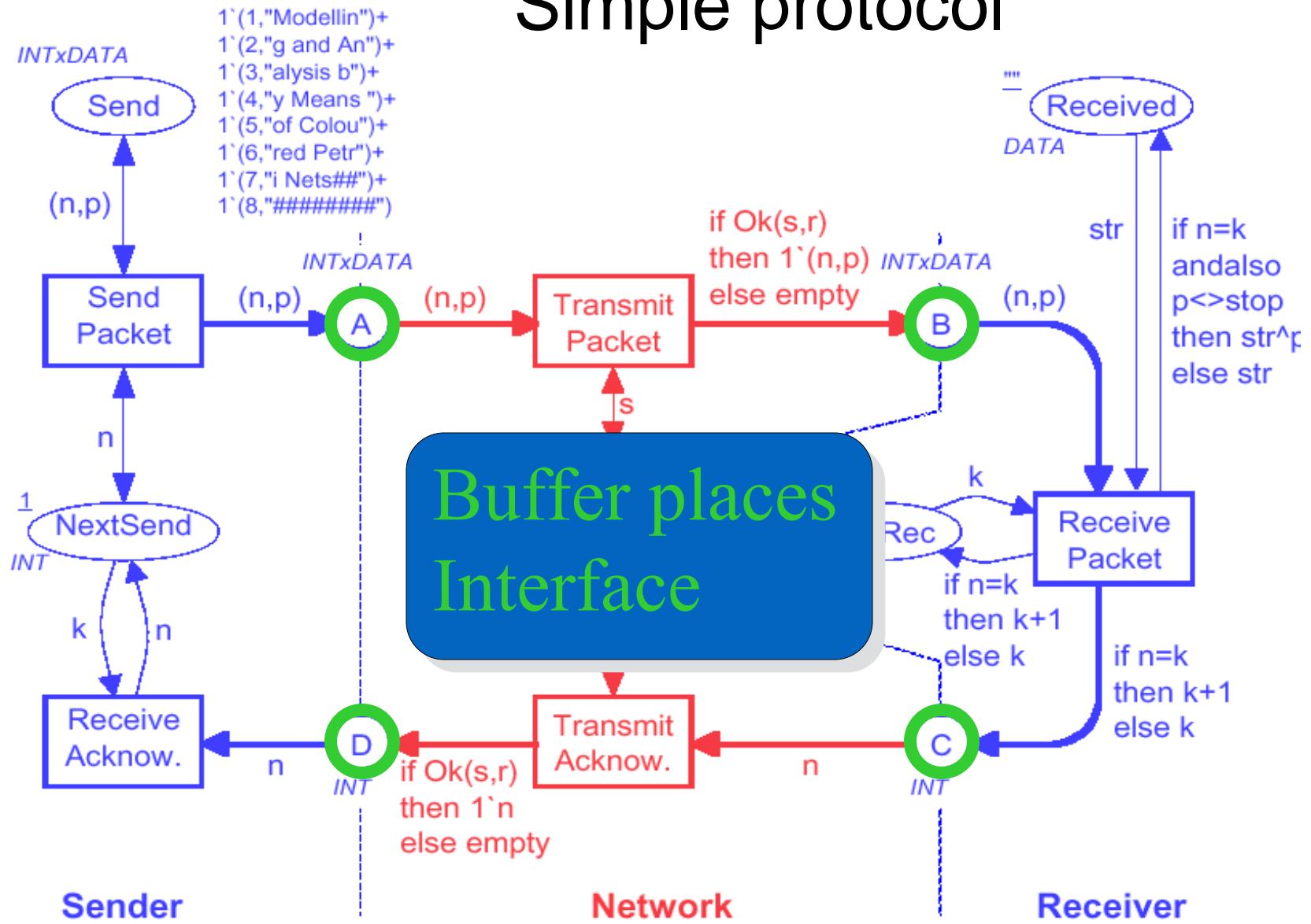
Simple protocol



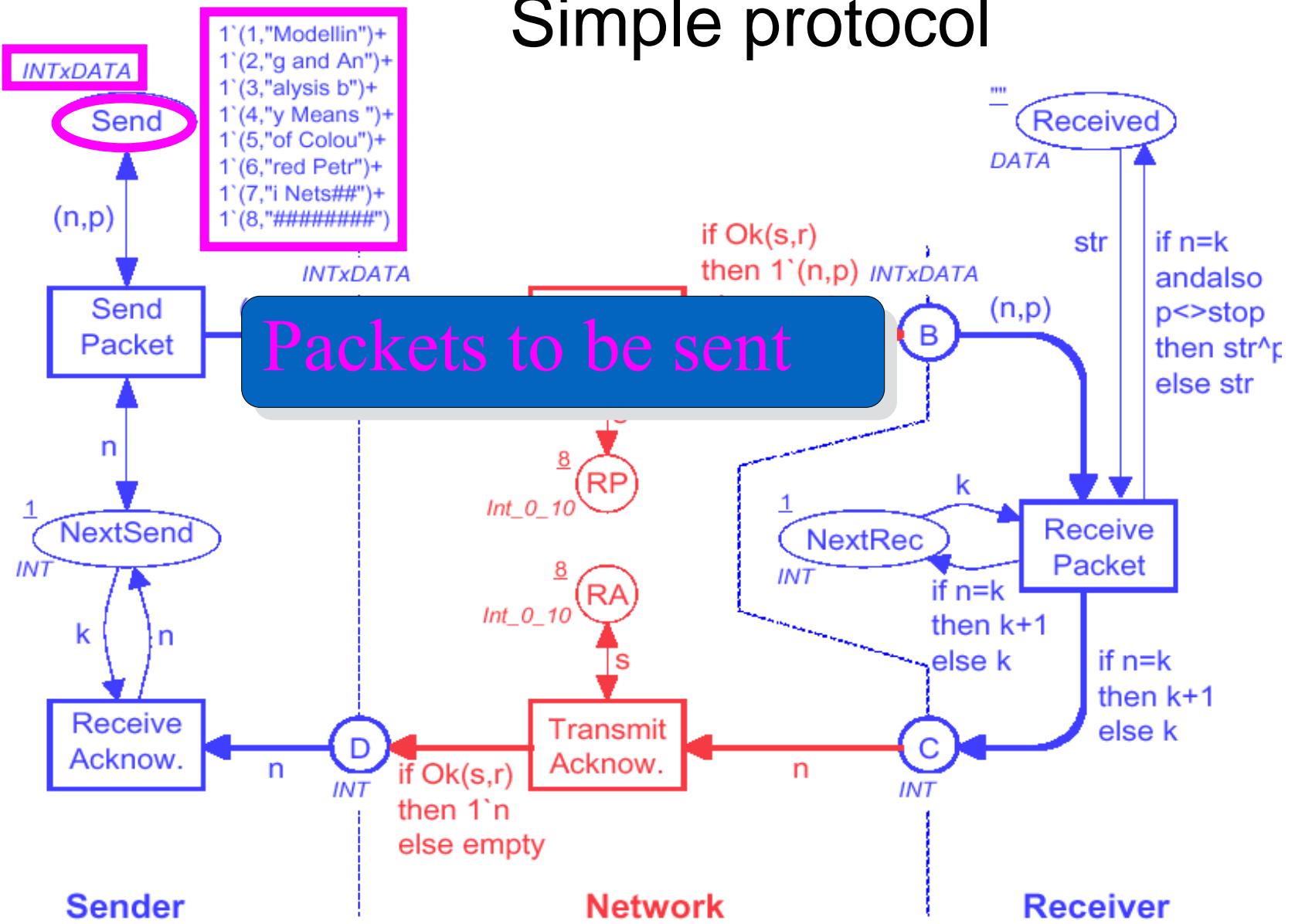
Simple protocol



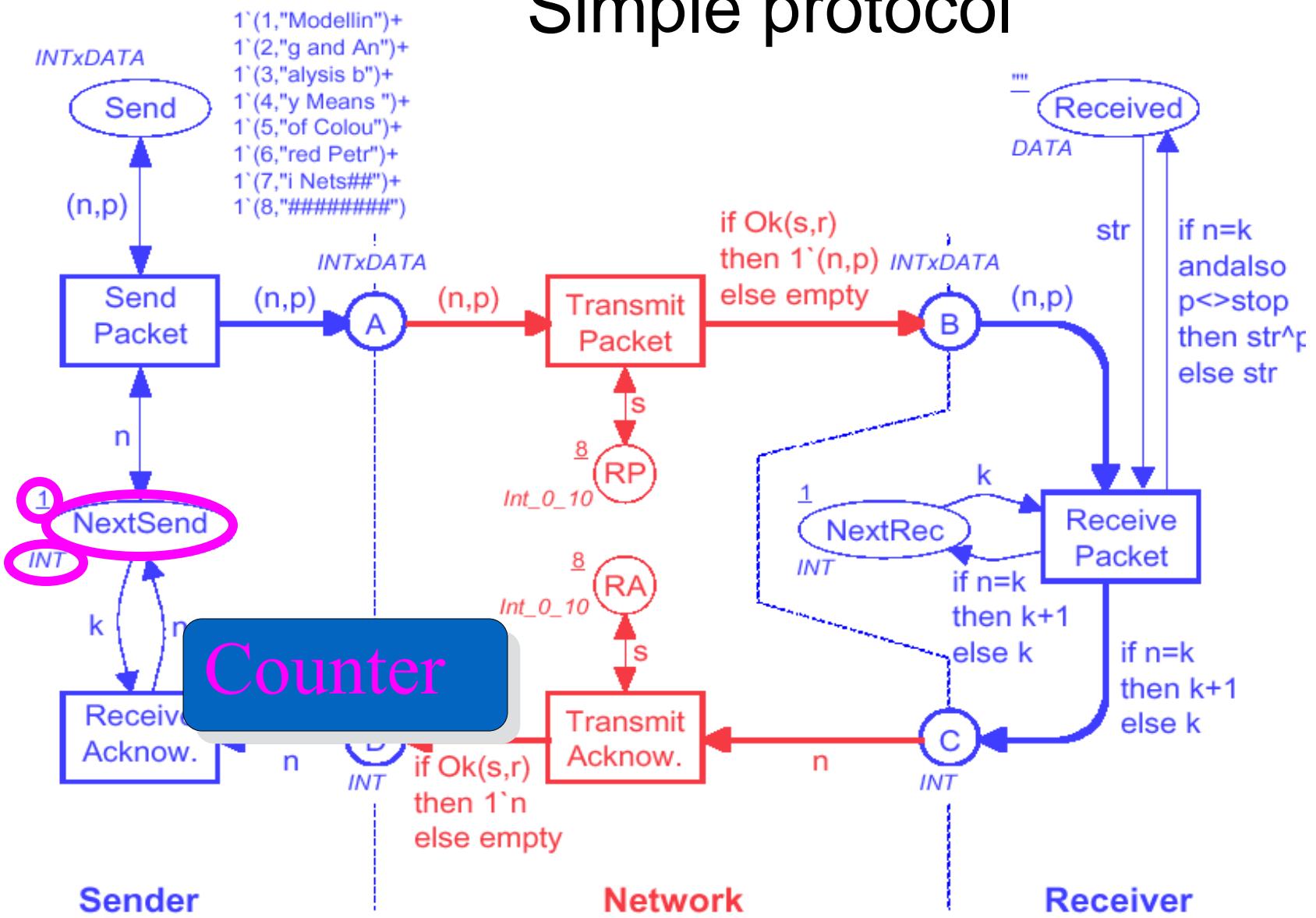
Simple protocol



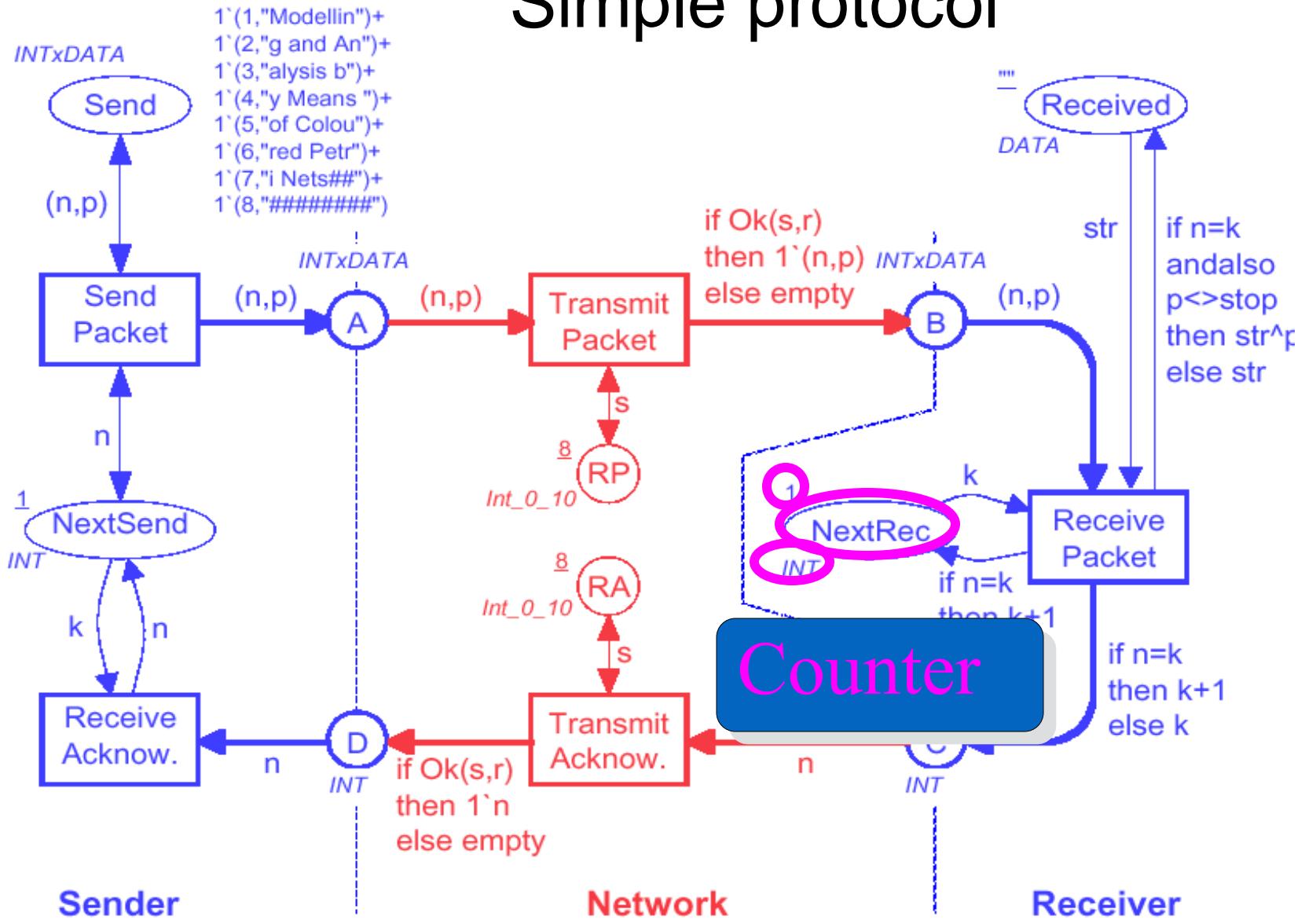
Simple protocol



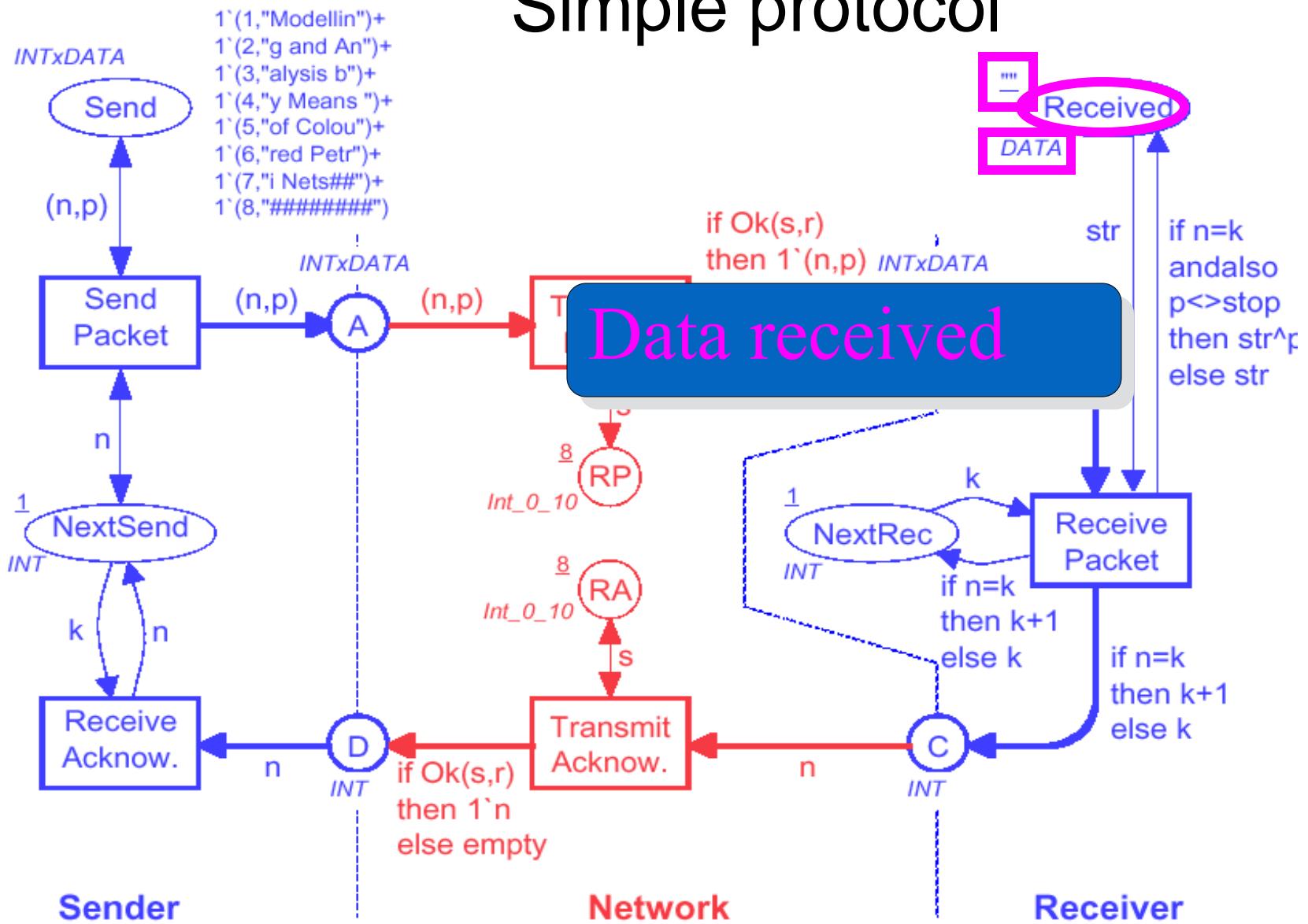
Simple protocol



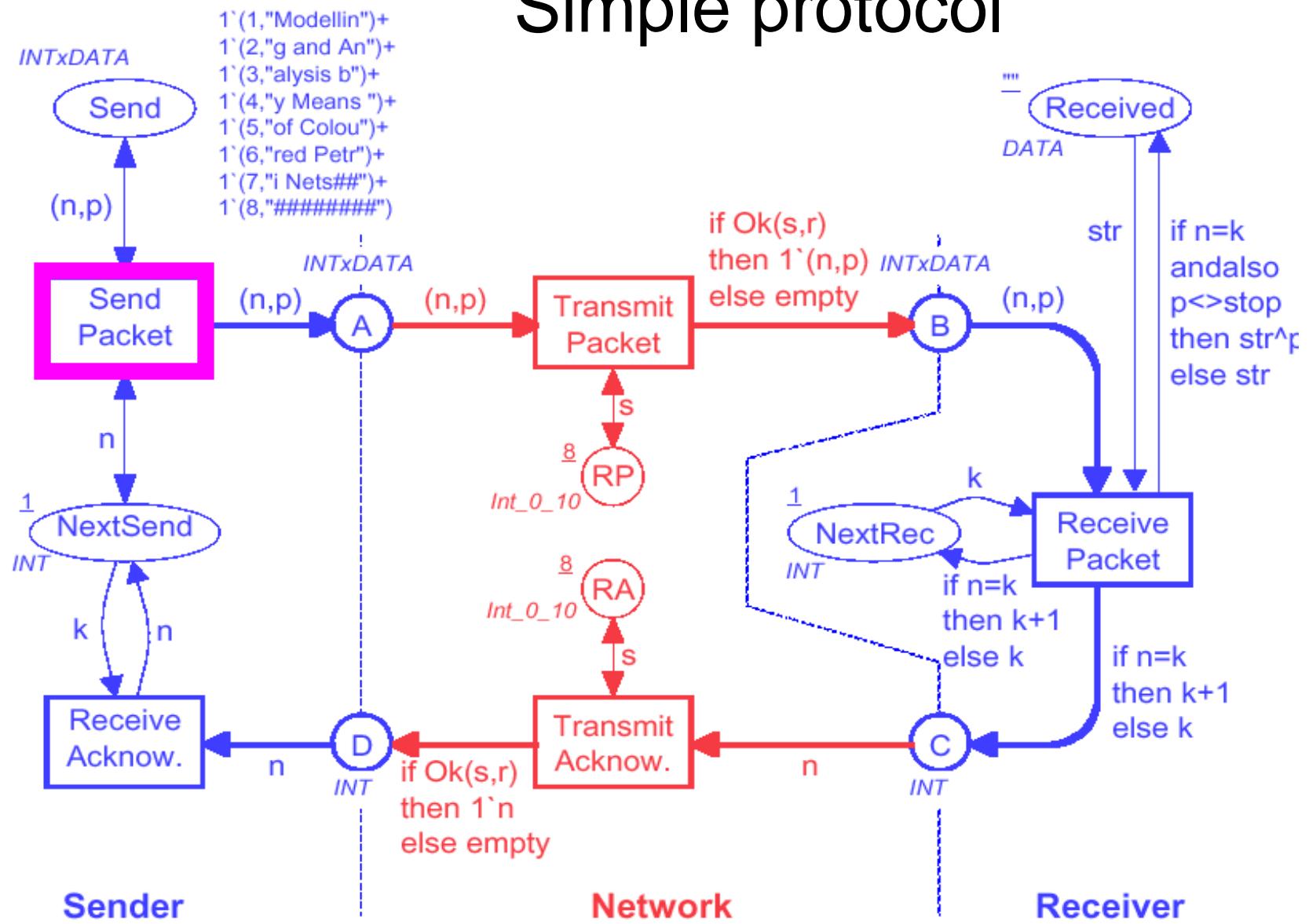
Simple protocol



Simple protocol



Simple protocol



Send packet

- The binding

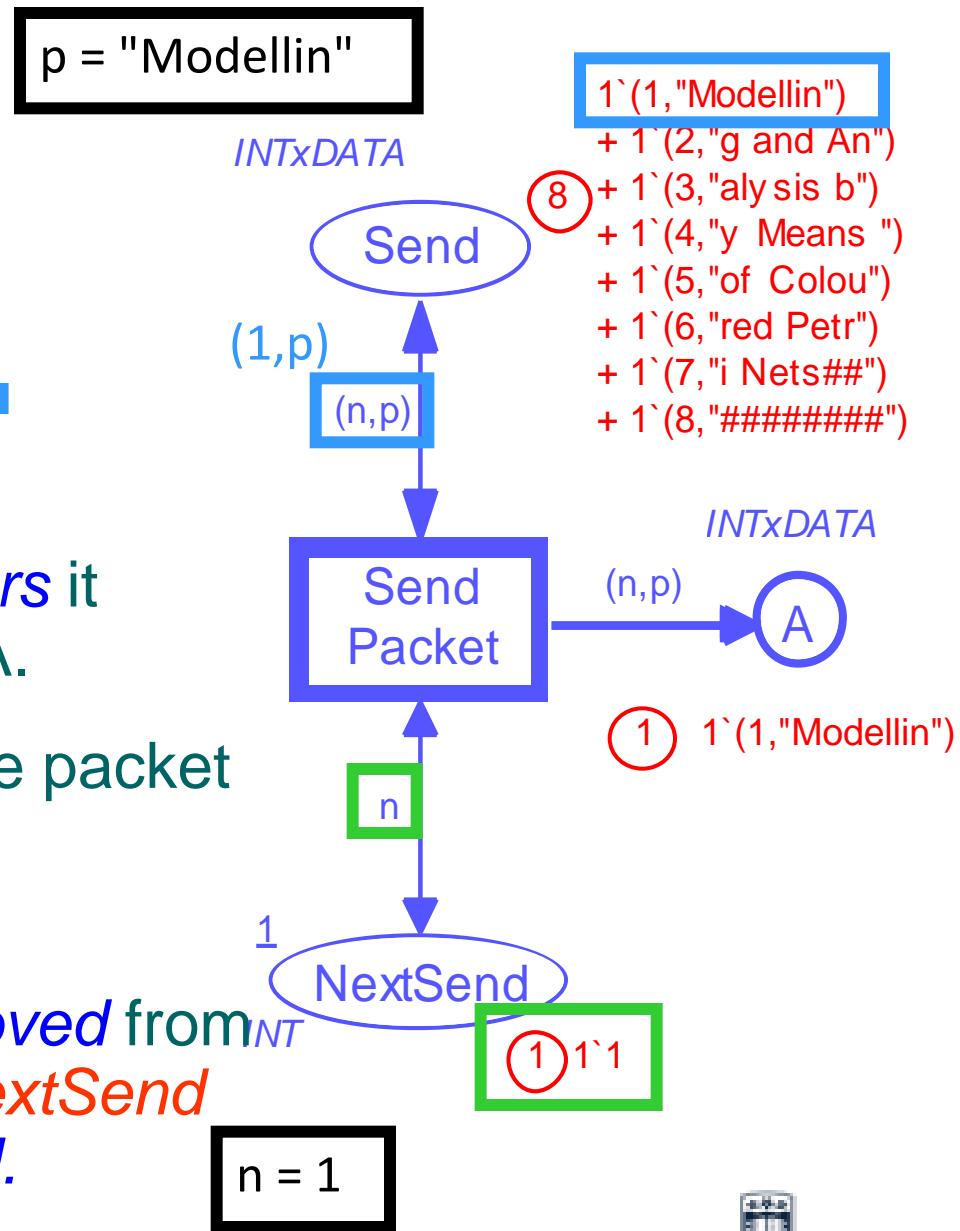
`<n=1,p="Modellin">`

is *enabled*.

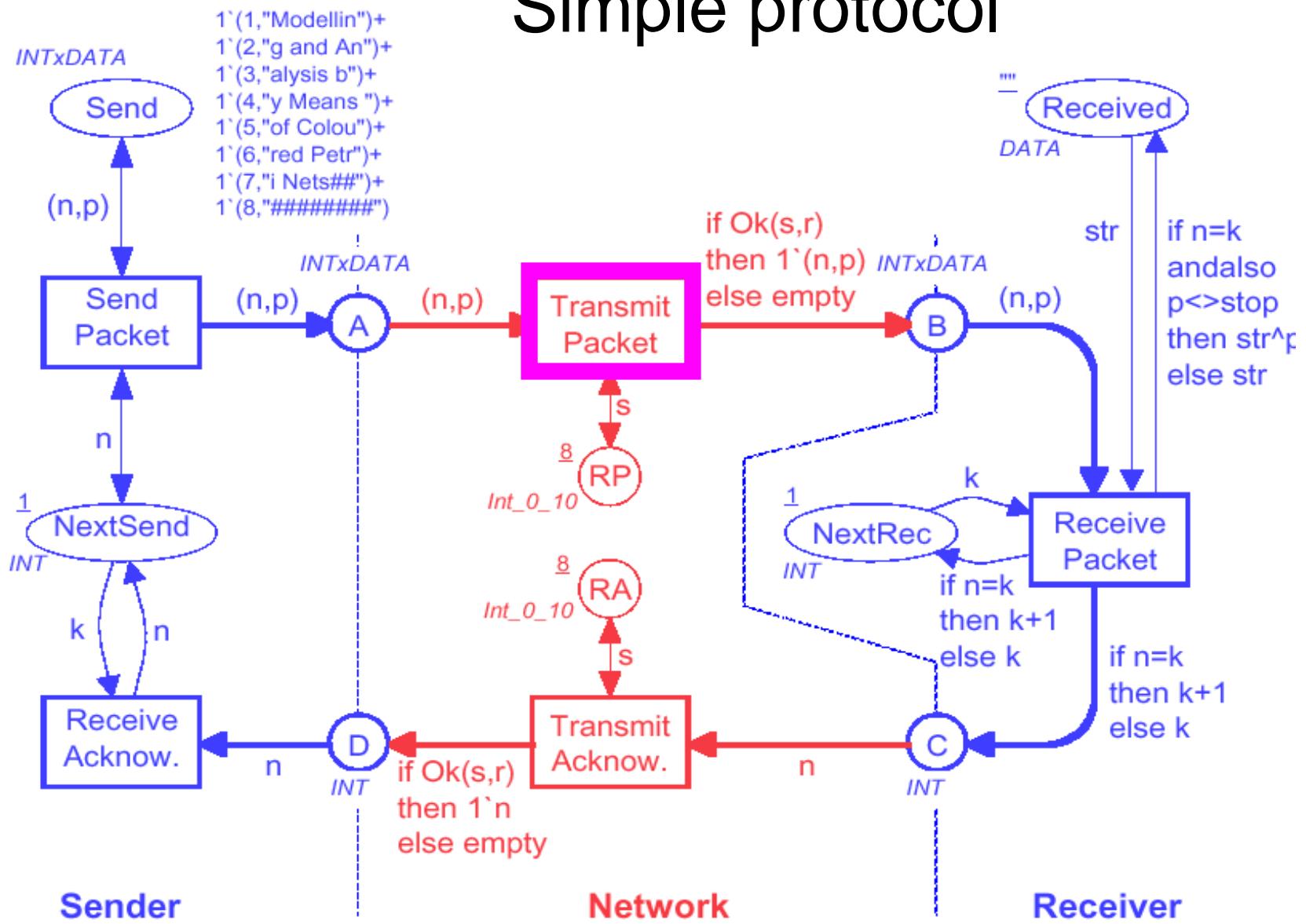
When the binding *occurs* it
adds a token to place A.

This represents that the packet
`(1,"Modellin")`
is *sent to the network*.

The packet is *not removed* from place *Send* and the *NextSend* counter is *not changed*.

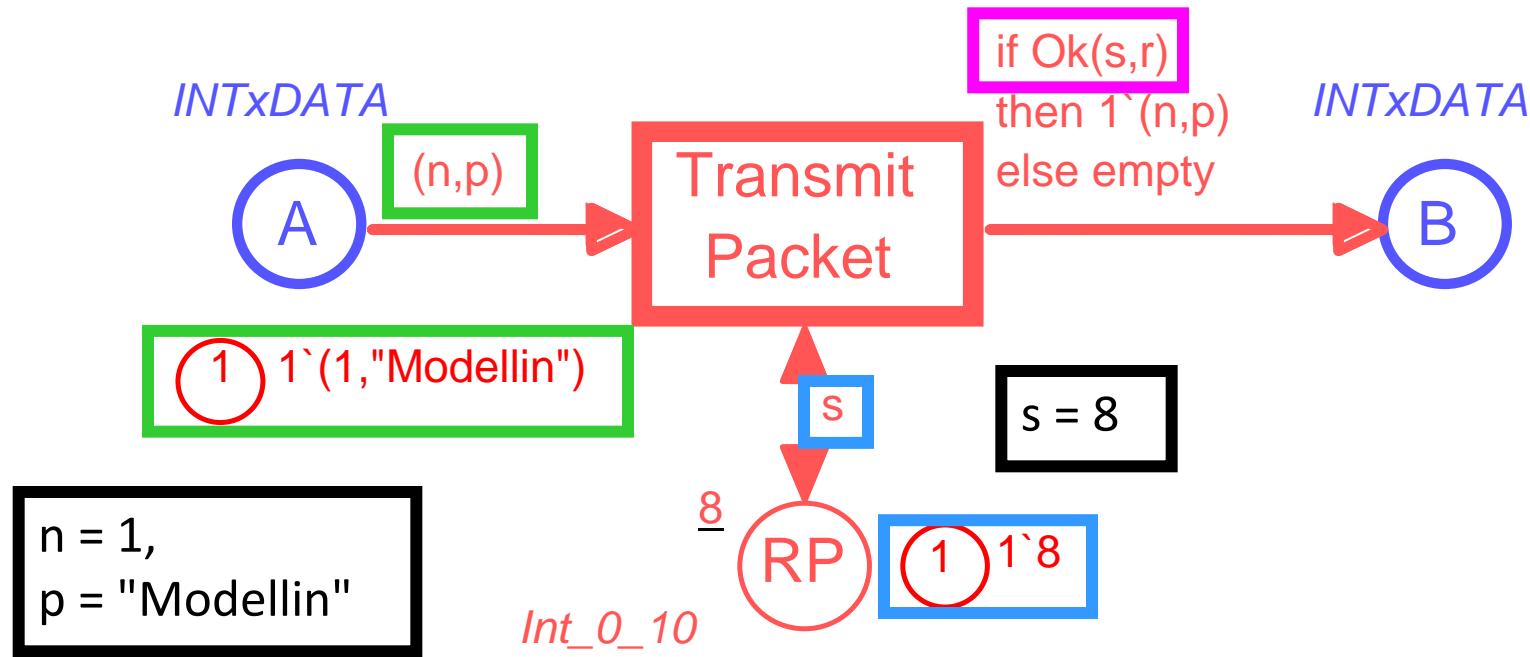


Simple protocol



Transmit packet

$r \in 1..10$



- All *enabled bindings* are on the form:
 - $< n=1, p= "Modellin", s=8, r=... >$

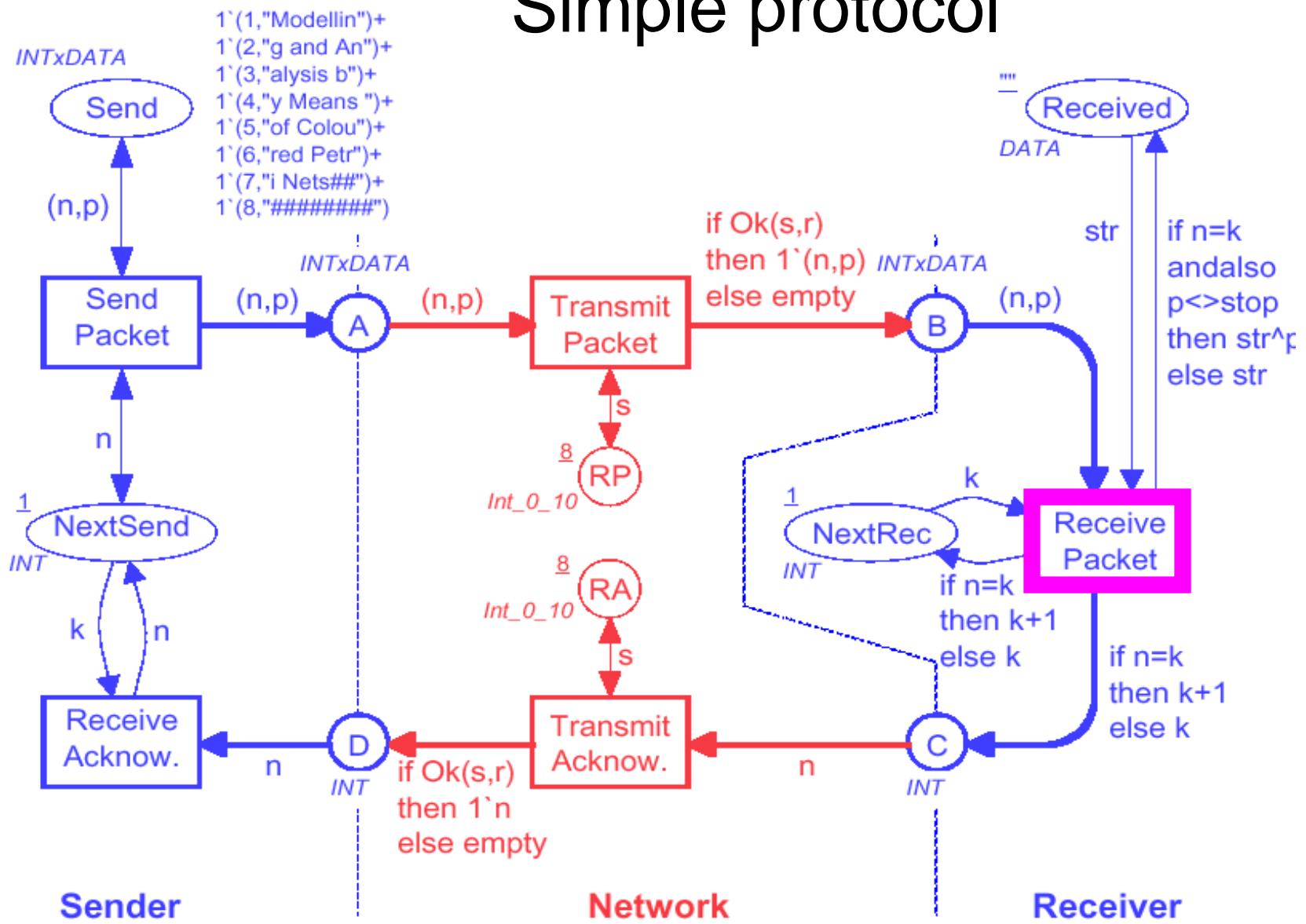
 - where $r \in 1..10$

Loss of packets

```
if Ok(s,r)  
then 1`(n,p)  
else empty
```

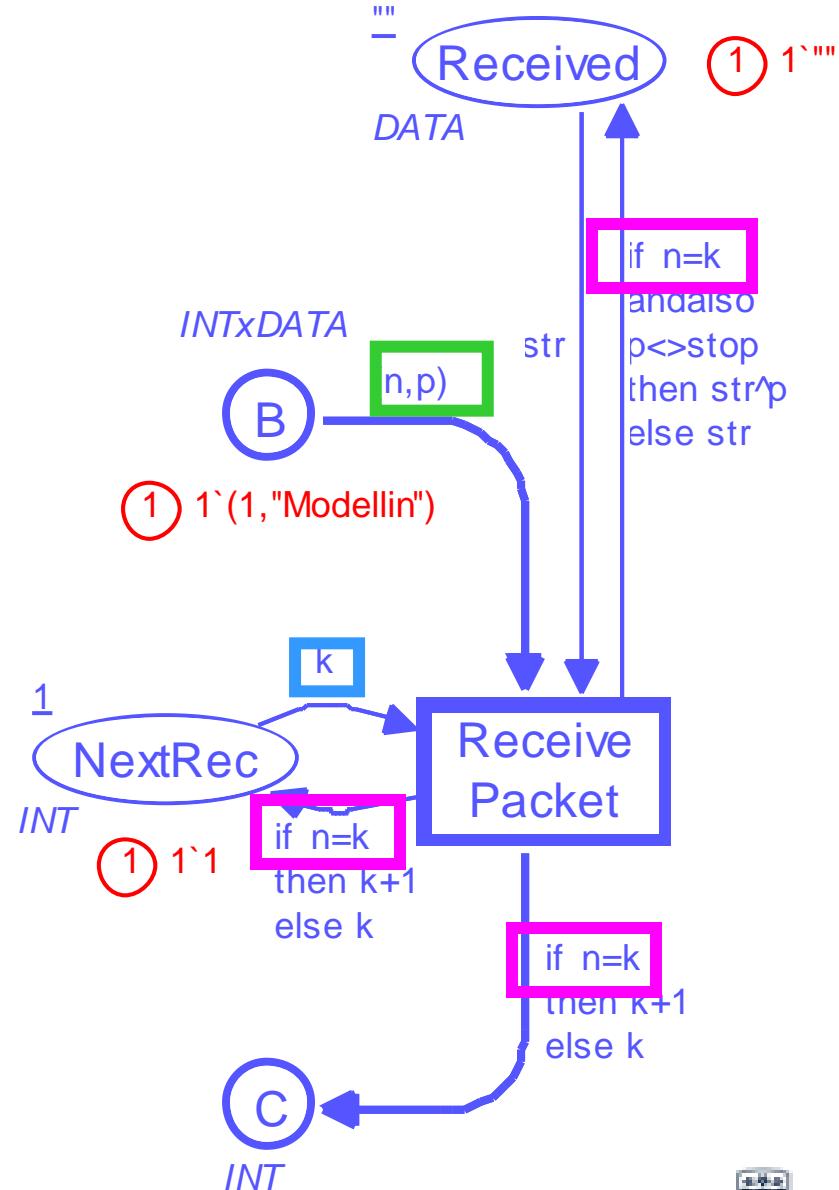
- The *function $Ok(s,r)$* checks whether $r \leq s$.
 - For $r \in 1..8$, $Ok(s,r)=\text{true}$.
The token is moved from A to B. This means that the packet is *successfully transmitted* over the network.
 - For $r \in 9..10$, $Ok(s,r)=\text{false}$.
No token is added to B. This means that the packet is *lost*.
- The CPN simulator makes *random choices* between bindings: 80% chance for successful transfer.

Simple protocol

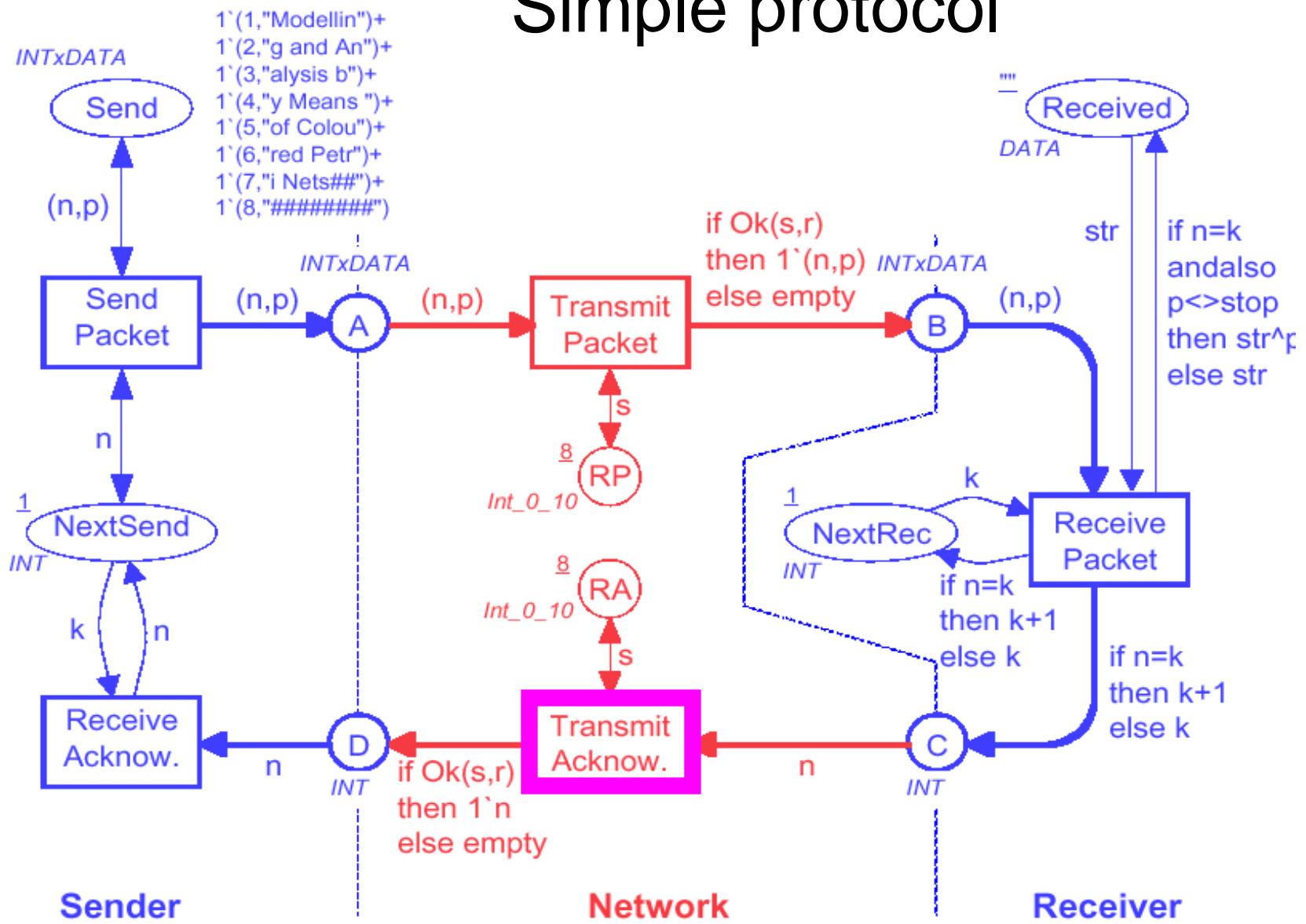


Receive packet

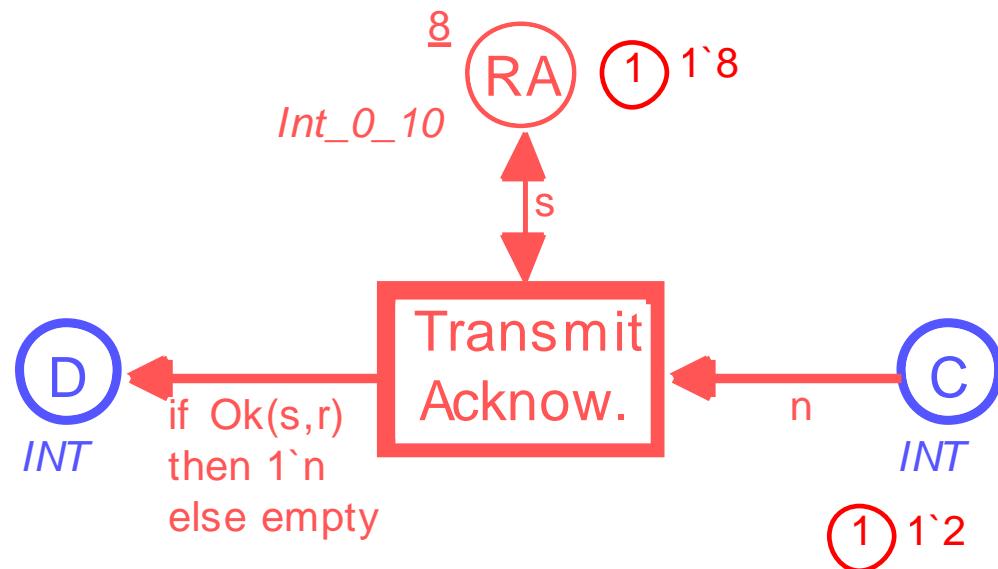
- The number of the *incoming packet n* and the number of the *expected packet k* are *compared*.



Simple protocol

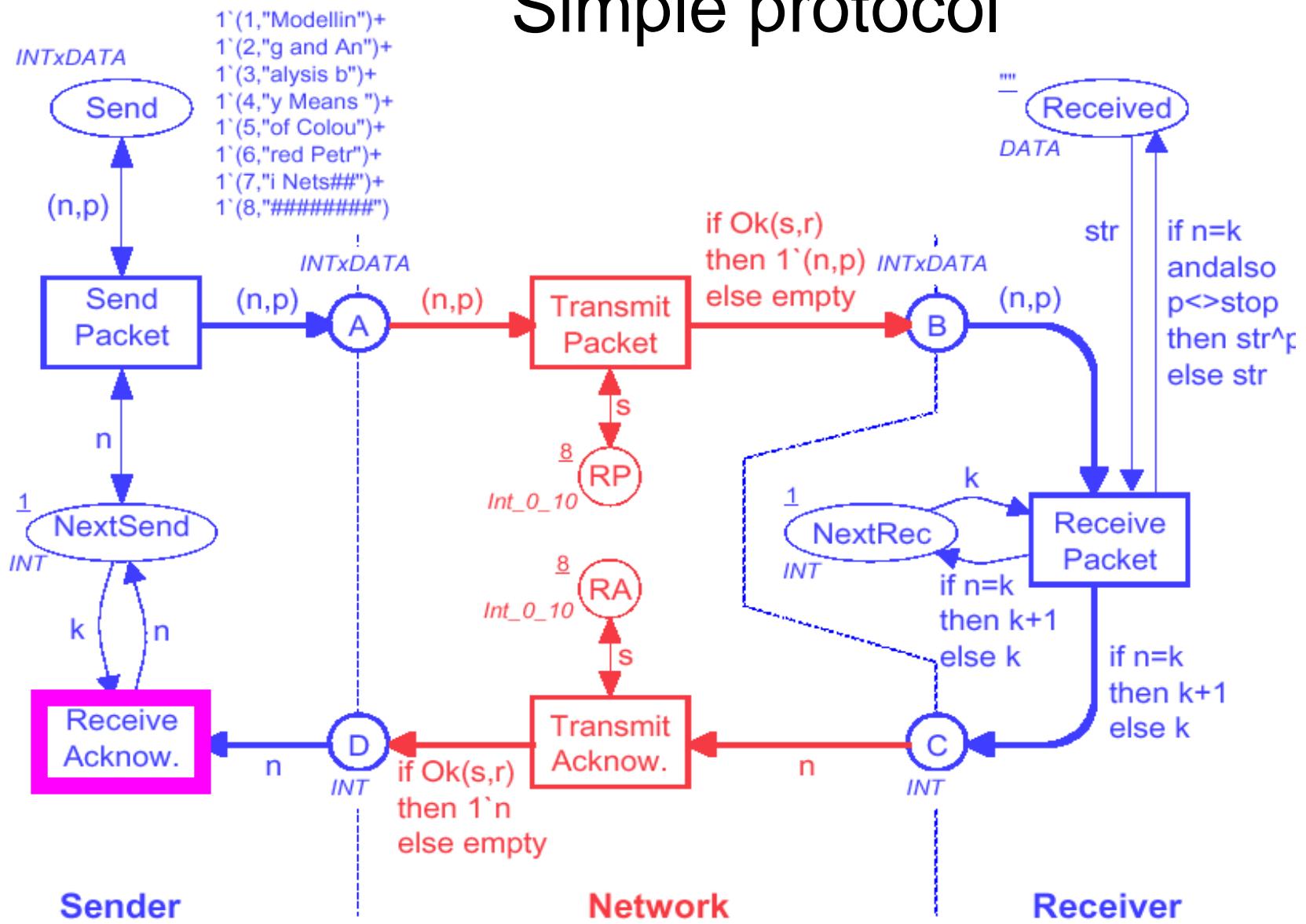


Transmit acknowledgement

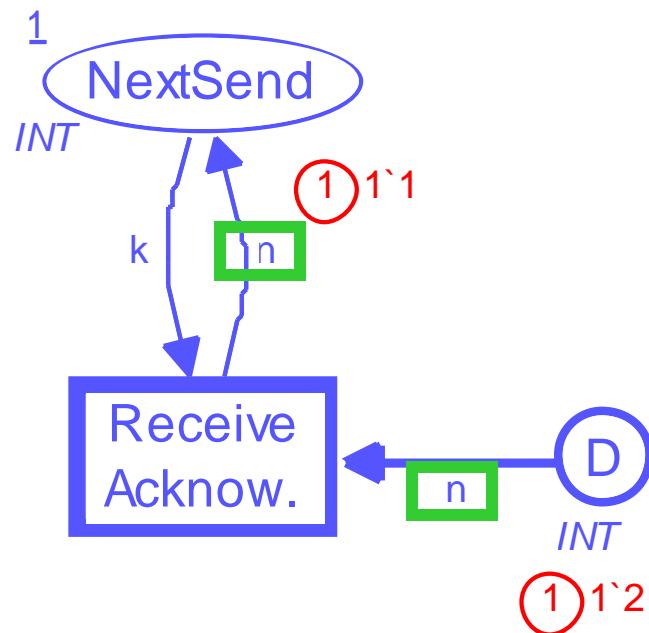


- This transition works in a similar way as *Transmit Packet*.
- The marking of *RA* determines the *success rate*.

Simple protocol

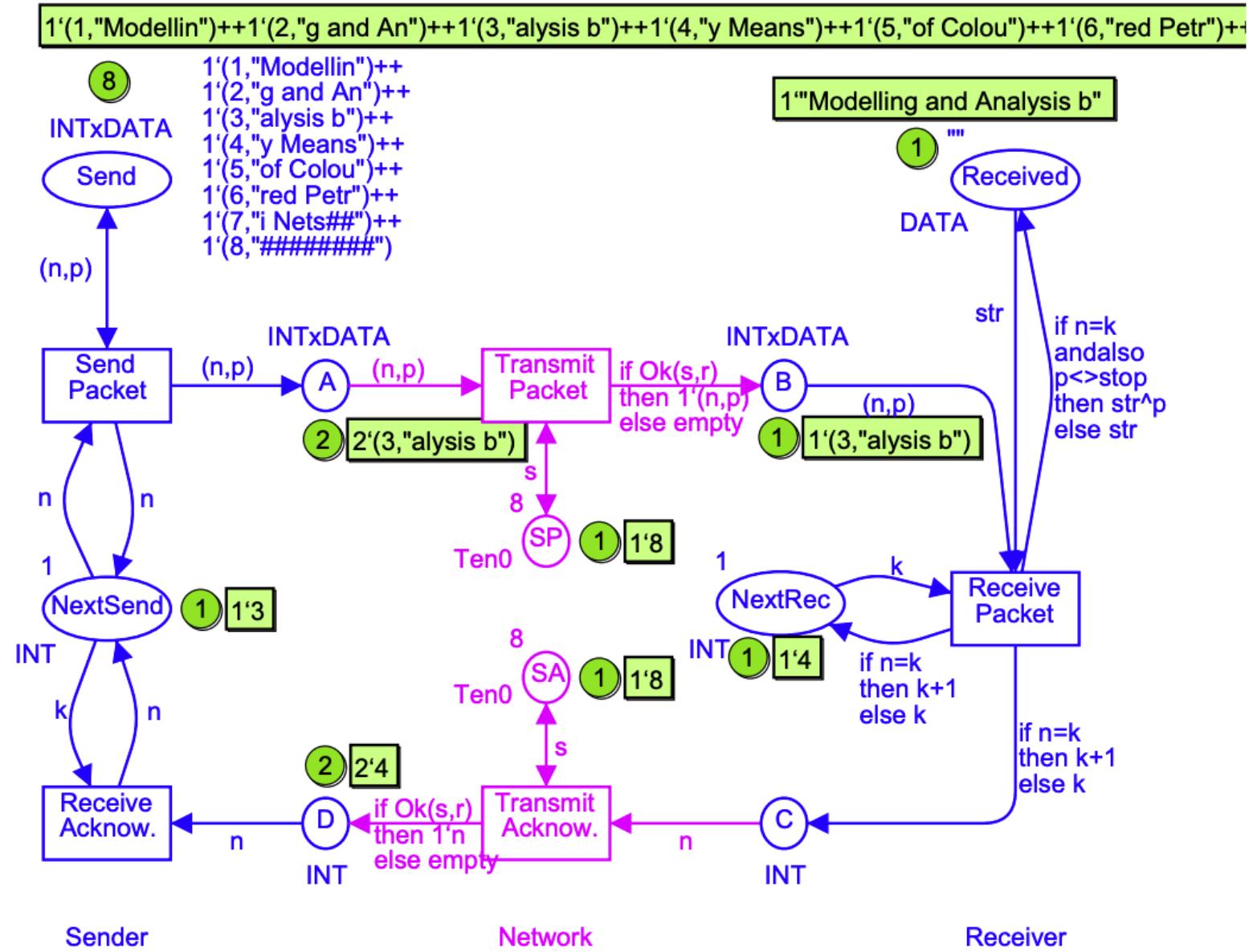


Receive acknowledgement



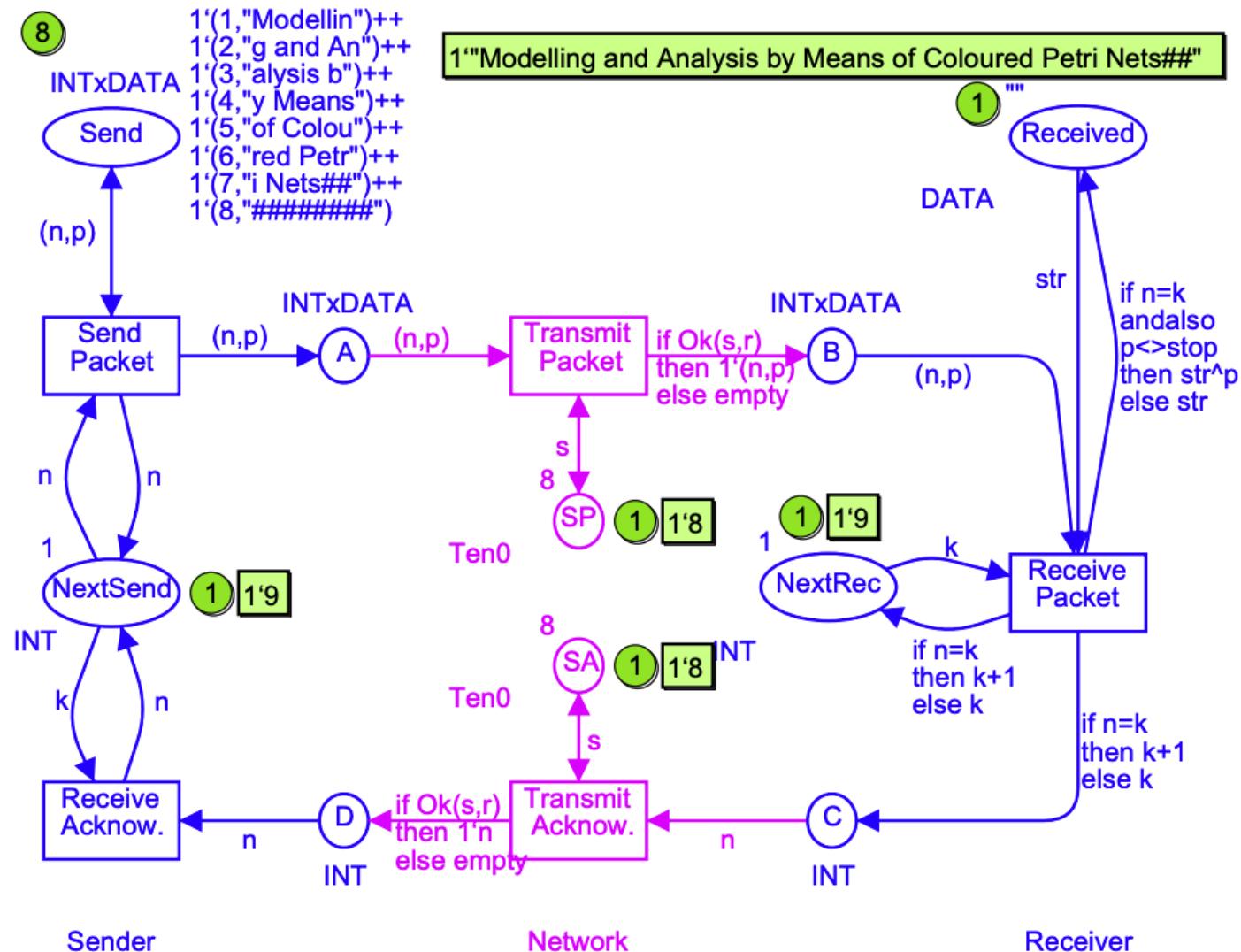
- When an acknowledgement arrives to the *Sender* it is used to update the *NextSend* counter.
 - In this case the counter value becomes 2, and hence the *Sender* will begin to send *packet number 2*.

Intermediate Marking



Final Marking

$1'(1, "Modellin")++1'(2, "g and An")++1'(3, "alysis b")++1'(4, "y Means")++1'(5, "of Colou")++1'(6, "red Petr")++1'(7, "i Nets##")++1'(8, "#####")$



Computer tools

Design/CPN was developed in the late 80'ies and early 90'ies.

- Today it is the *most widely used* Petri net package.
- *750 different organisations* in *50 countries*
- including *200 commercial companies*.

CPN Tools/IDE is the next generation of tool support for coloured Petri Nets.

- CPN Tools/IDE is expected to replace *Design/CPN* and obtain the same number of users.
- <https://cpnide.org/>

CPN Tool Simulation



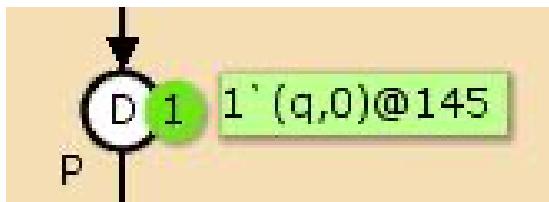
UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Timed CPN

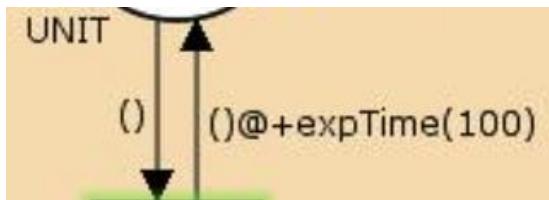
- Time-related information are required.
- Can CPN measure time?
- Can transitions be fired by considering a time stamp?

The answer is YES.

- We can put time stamps on the tokens, and

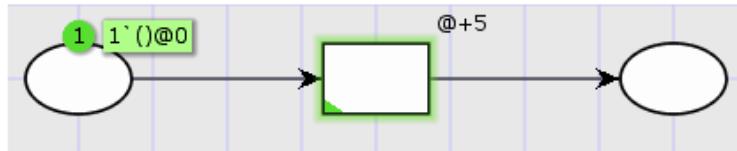


- We can modify the transitions and add arc functions to consider the time stamp.

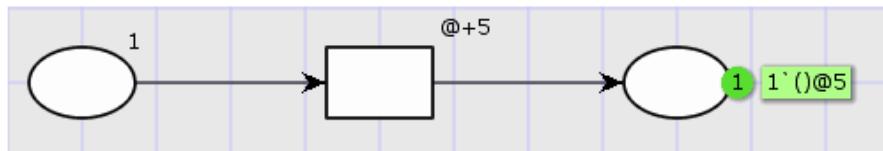


Time Intervals

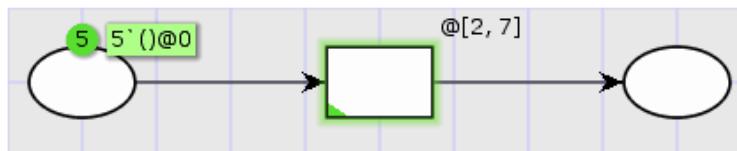
Timed models allow us to specify a duration for each transition, like this:



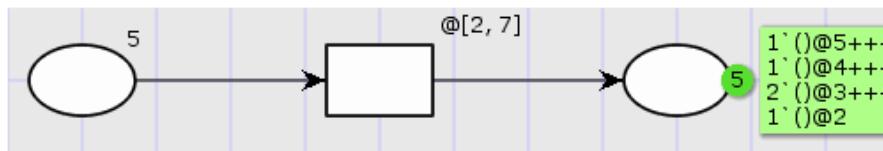
When we execute the transition, the produced token will have a time-stamp that is 5 time units into the future, indicating we cannot consume the token before time 5:



In CPN Tools 4, we can also specify an interval, indicating that the actual value of the time stamp should be taken at random from the interval:

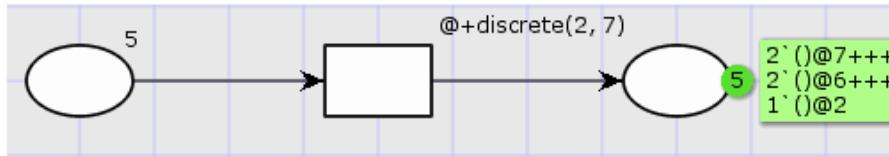


Now, if we execute the transition (5 times), we get this state:

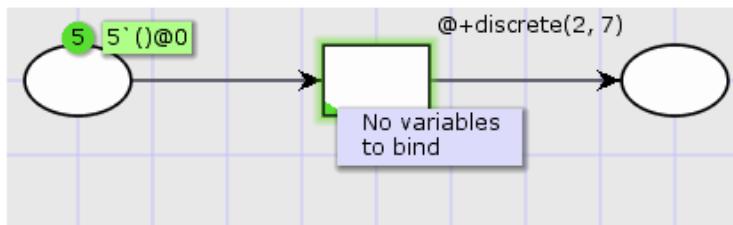


We see that the tokens have different time stamps; they are chosen at random (uniformly) from the interval specified. We could of course achieve something similar using one of the **probability distribution functions**:

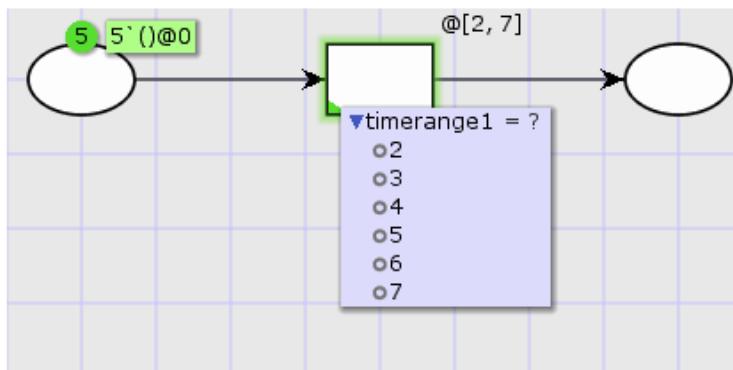
Time Intervals



We can still do that, and if we want another distribution than the uniform distribution we have to do that. The difference between then and now is that if we look at the possible bindings of the transition in the initial state, as expected, we get:

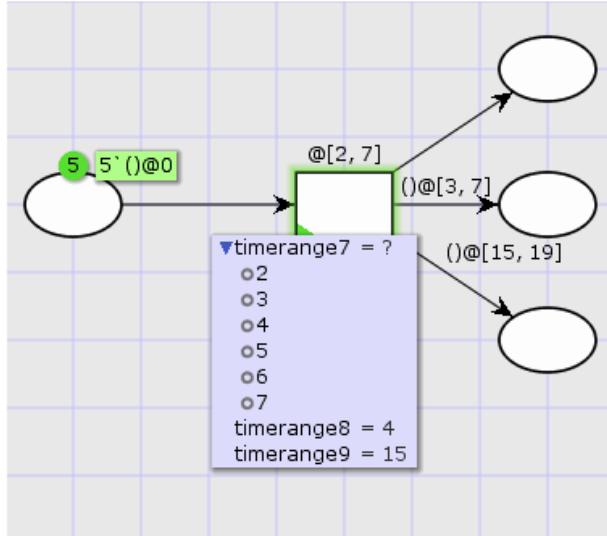


The variable has no variables, so trying to manually select them would not work. However, using the new syntax we get:

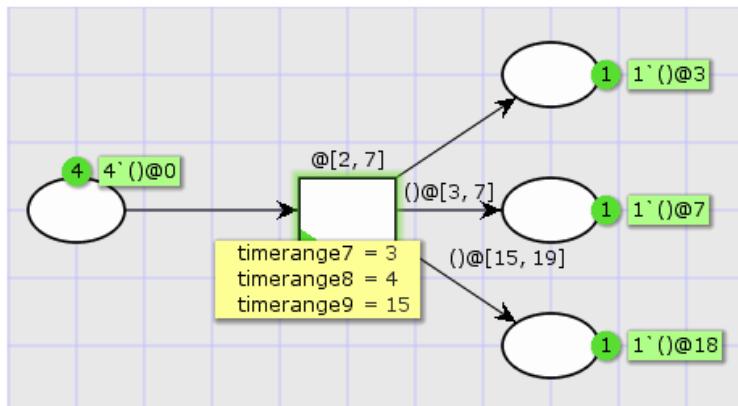


As we see, we can actually pick the time stamp for the token here¹. Time intervals can be used in time regions (as above) and on output arcs:

Time Intervals

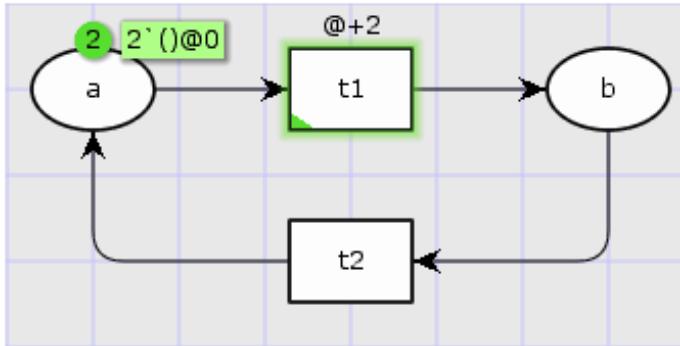


The values are added, so the token on the middle place would arrive 4 time units later than the top one, and the bottom one another 11 time units later in this case:

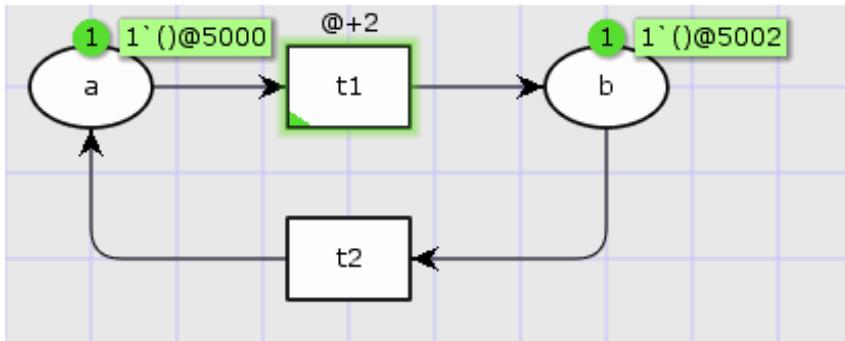


No only can manually select the time ranges, we can also use such models for state-space analysis. This would not be possible using random distribution functions as the tool cannot in that case determine how to do exhaustive analysis.

Time Reduction

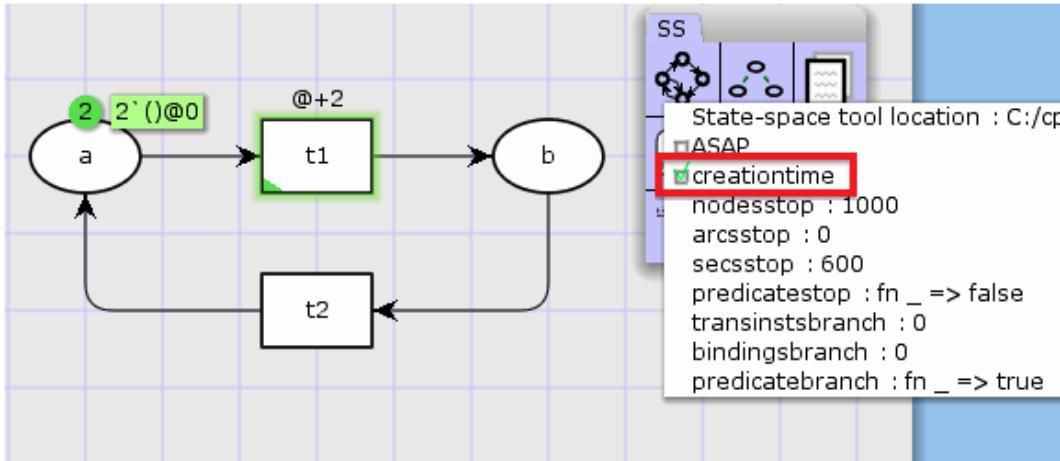


This model has an infinite state-space even though the behavior is very simple: we just have two cycles with t_1 and t_2 , moving tokens between a and b . Executing "a lot" of steps reveals why the state-space is infinite:

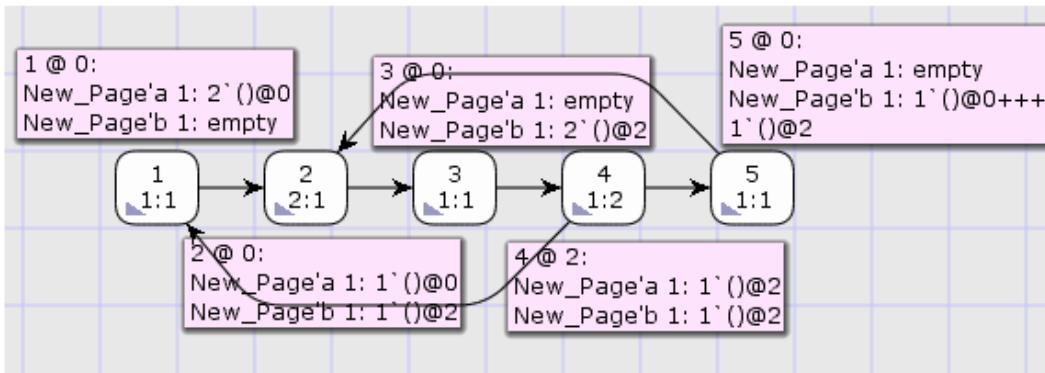


We see that while the behavior is finite, the time-stamps grow unbounded. This model, however, never looks at the absolute value of the time-stamp, but only uses it for delay. Therefore, the actual value of the time-stamp is not so important; we only care about the delay. We can use this to do reduction:

Time Reduction



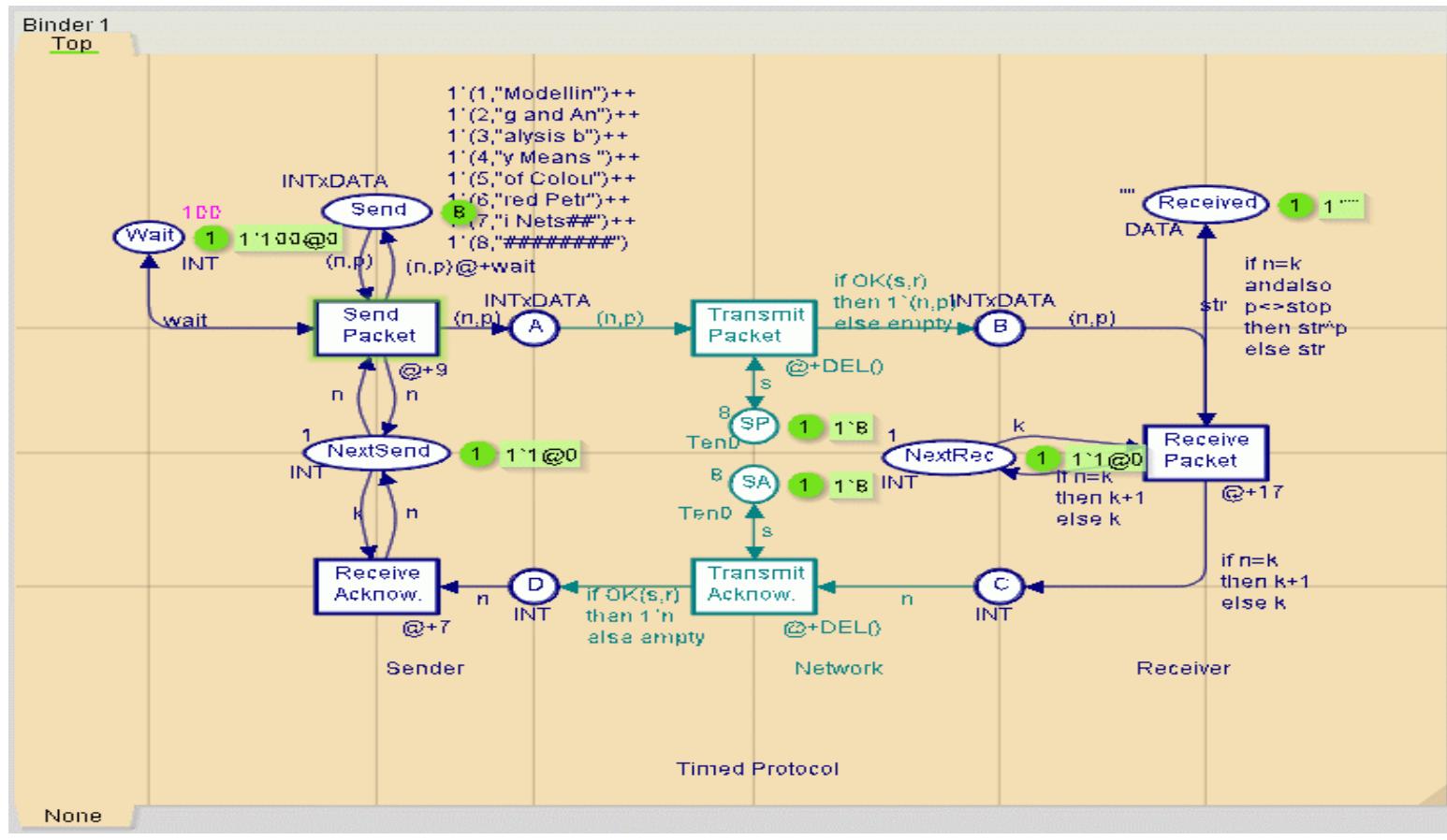
With this option switched on, we obtain a state-space with just 5 nodes:



The idea is that we ignore the actual model time only looks at how far into the future each time-stamp is. This effectively resets the model time after executing each transition, subtracting the model time from each time-stamp. We also do not care about any time-stamps that turn negative this way – if the time stamp is less than or equal to the model time, the token is ready, and the actual value has no influence on this.

Timed Simple Protocol

Showing how long time the individual operations take and how long time the sender should wait before it makes a retransmission.



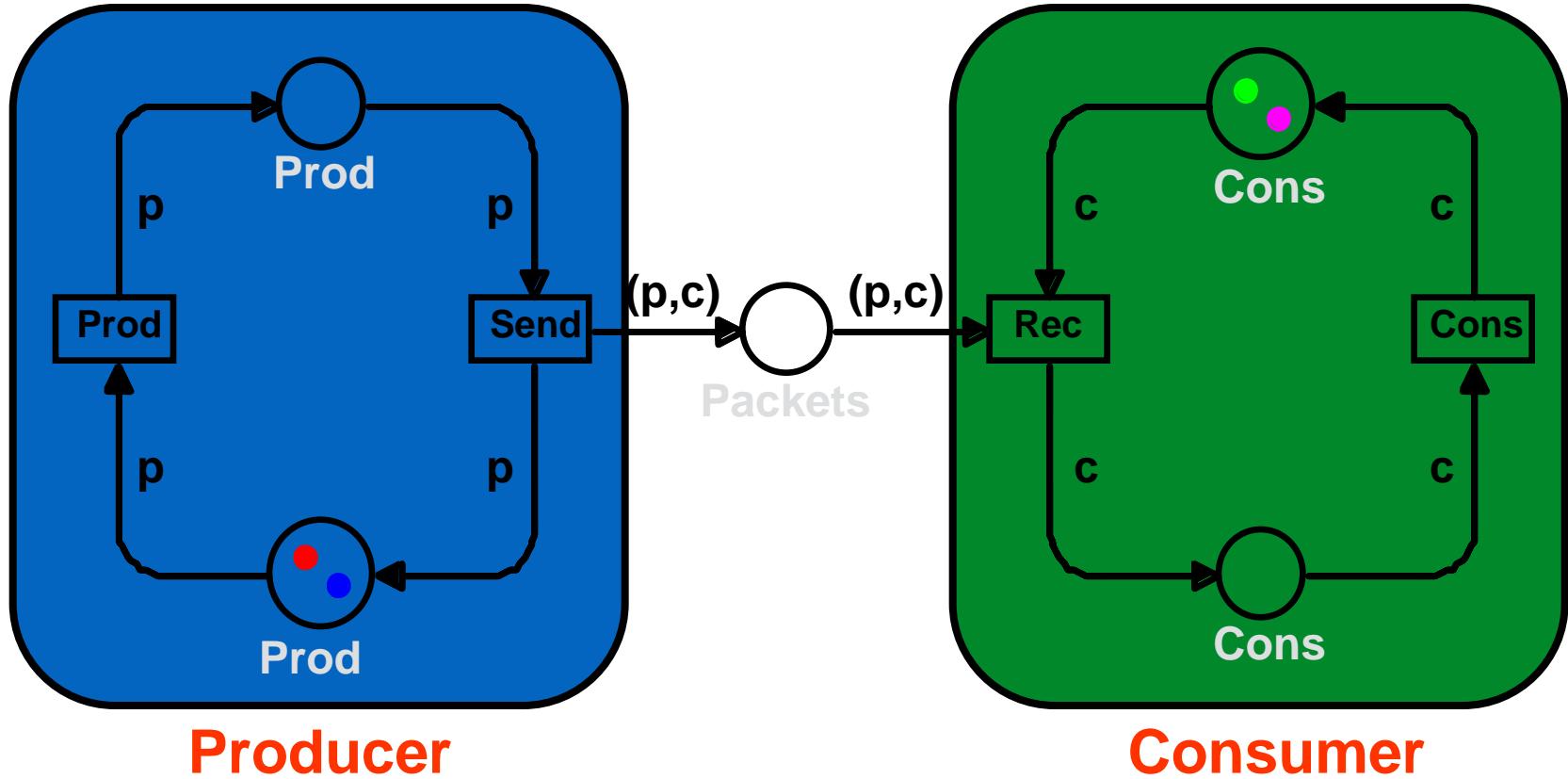
CP-nets are used for large systems

- A CPN model consists of a number of *modules*.
 - Also called *subnets* or *pages*.
 - Well-defined *interfaces*.
- A typical *industrial application* of CP-nets has:
 - 10-200 modules.
 - 50-1000 places and transitions.
 - 10-200 types.
- Industrial applications of this size would be *totally impossible* without:
 - Data types and token values.
 - Modules.
 - Tool support.

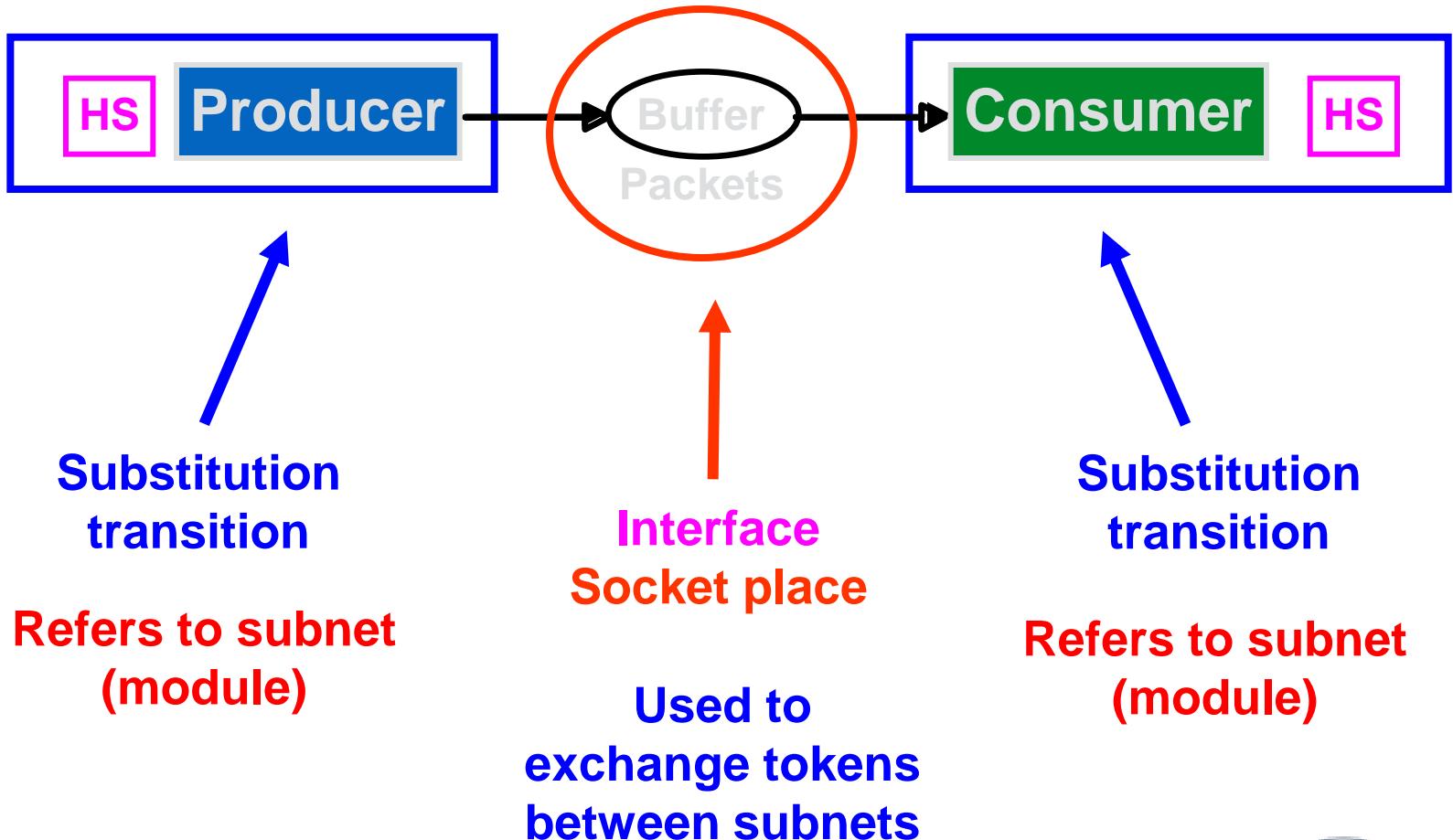
Hierarchical descriptions

- We use *modules* to *structure large* and *complex* descriptions.
- Modules allow us to *hide details* that we do not want to consider at a certain *level of abstraction*.
- Modules have *well-defined interfaces*, consisting of *socket* and *port places*, through which the modules *exchange tokens* with each other.
- Modules can be *reused*.

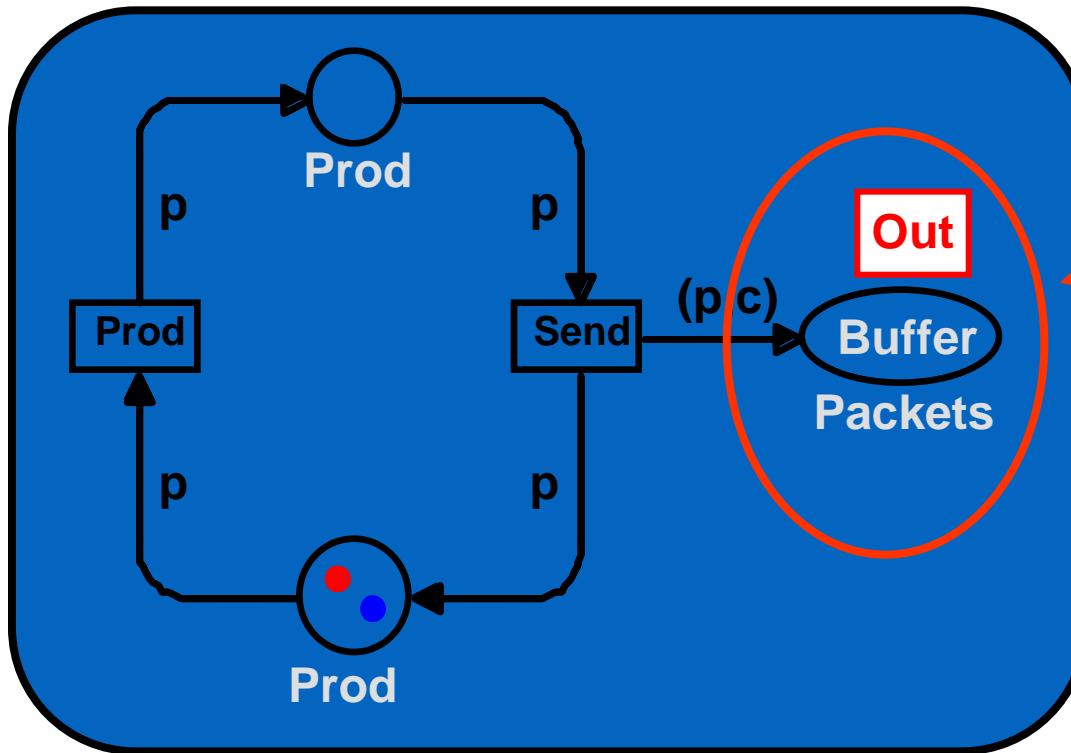
Hierarchical descriptions (modules)



Abstract view



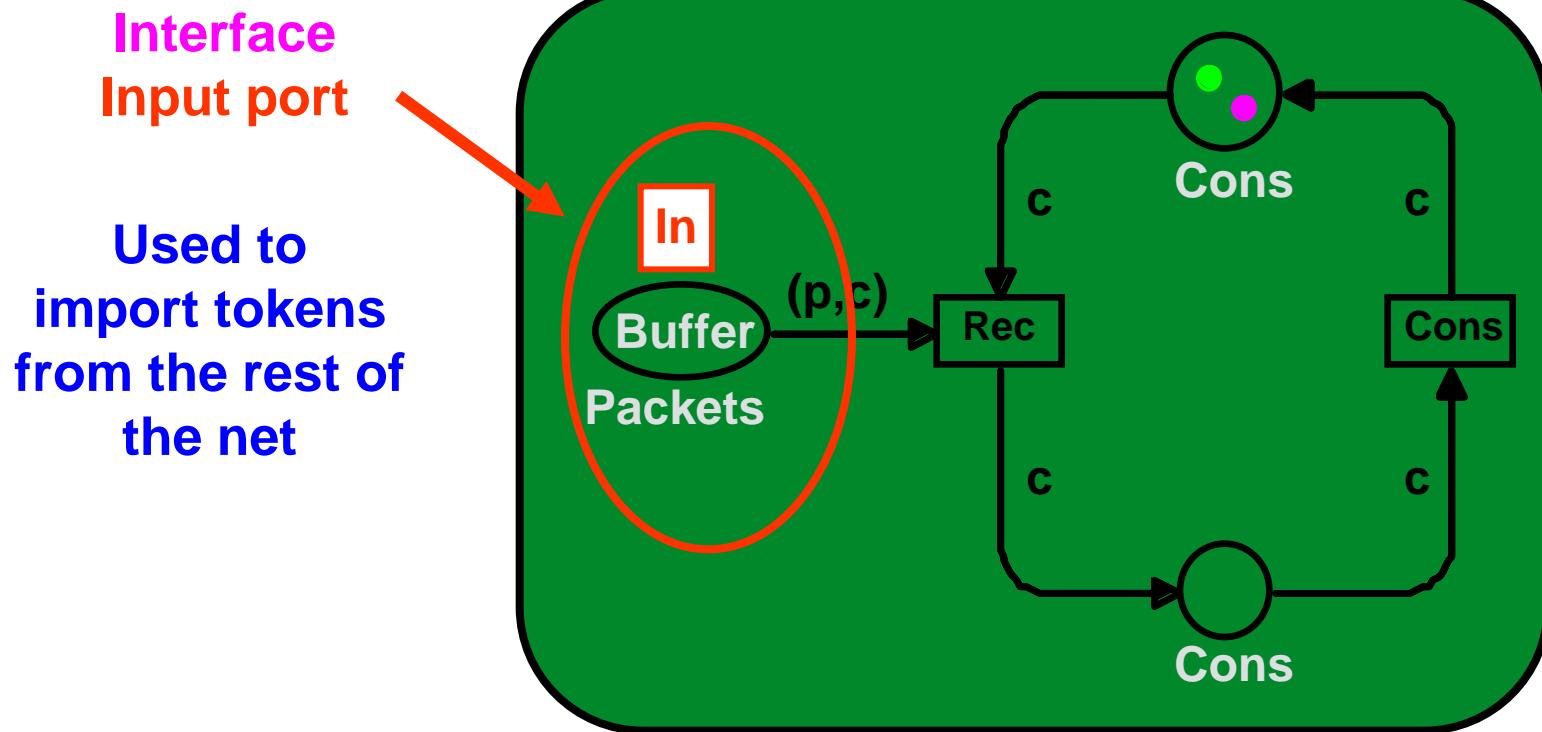
Producer module



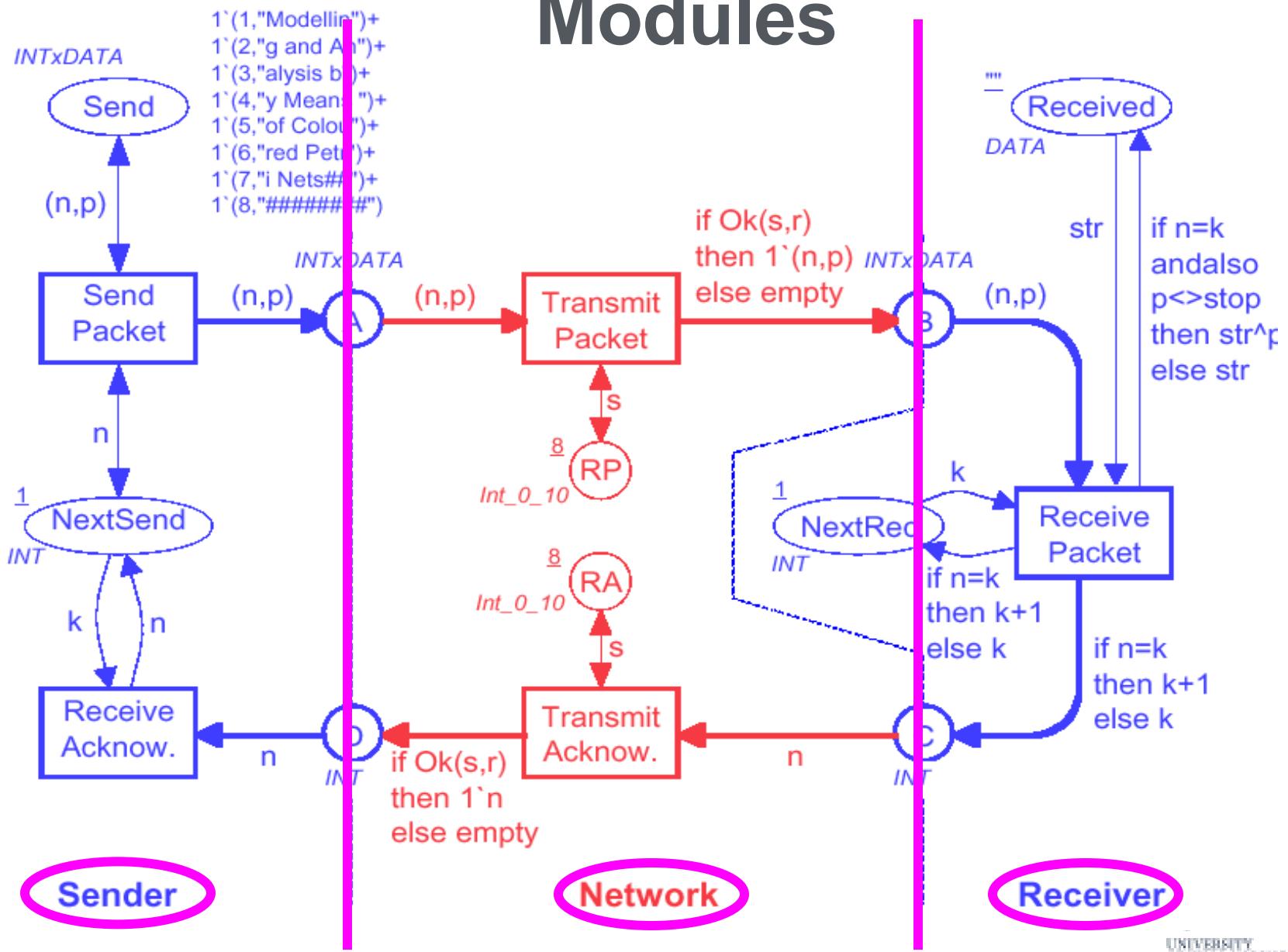
Interface
Output port

Used to
export tokens
to the rest of
the net

Consumer module

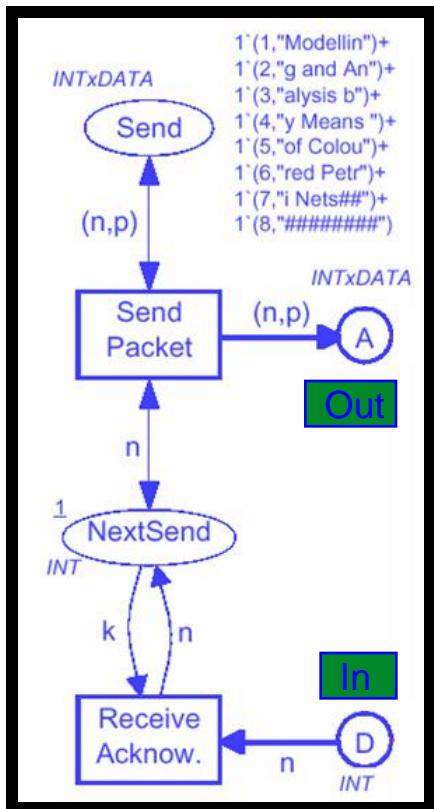


Modules

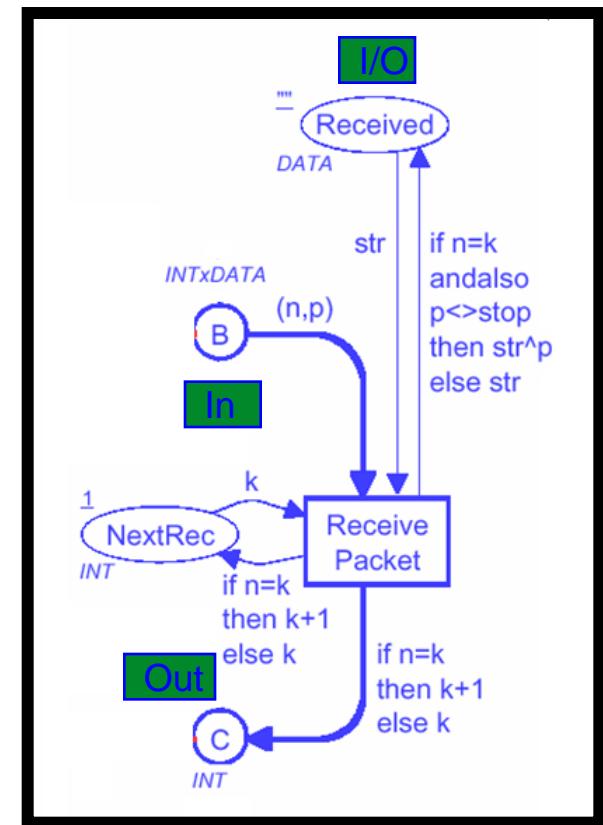


Three different modules

Sender



Receiver

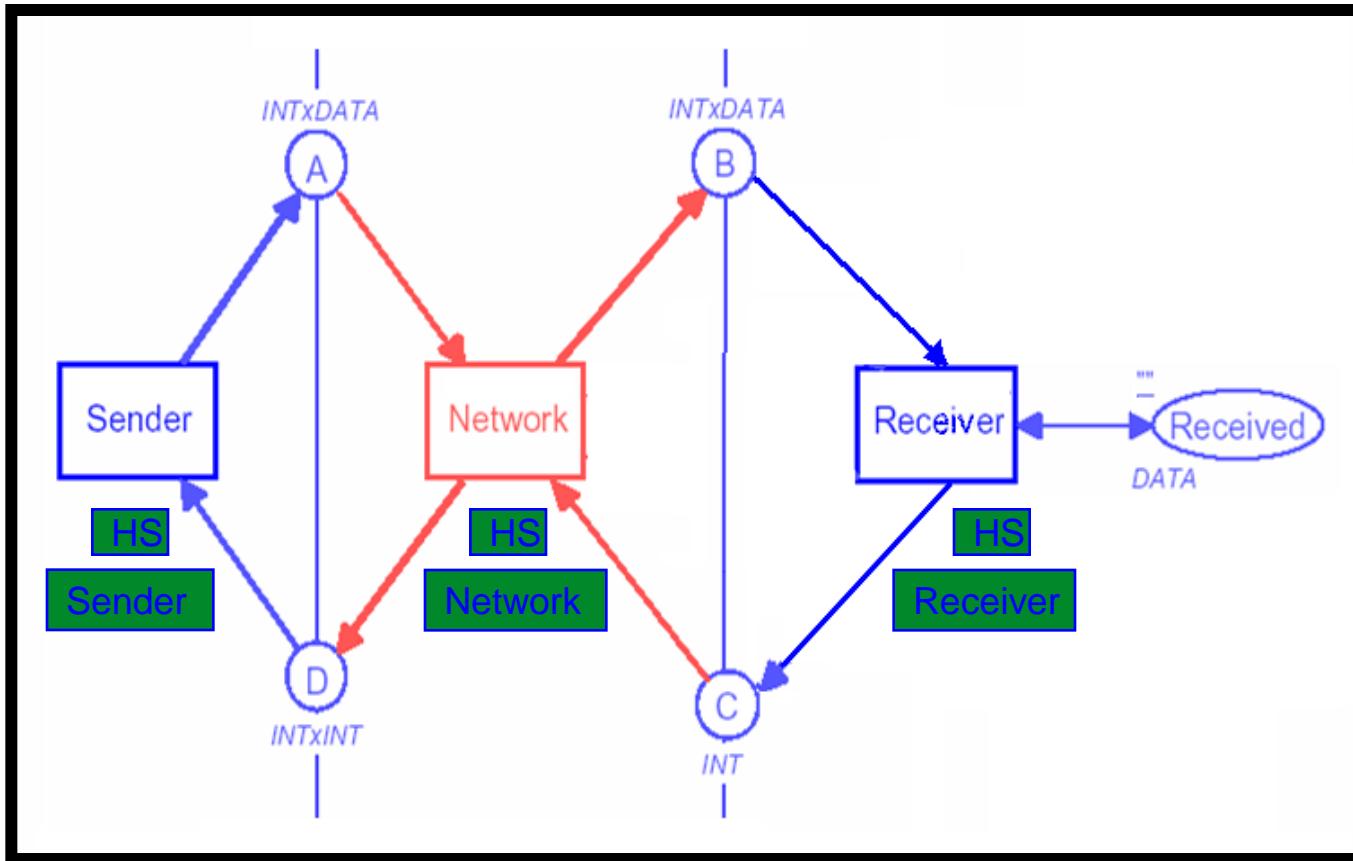


Port places are used to exchange tokens between modules.



Abstract view

Protocol

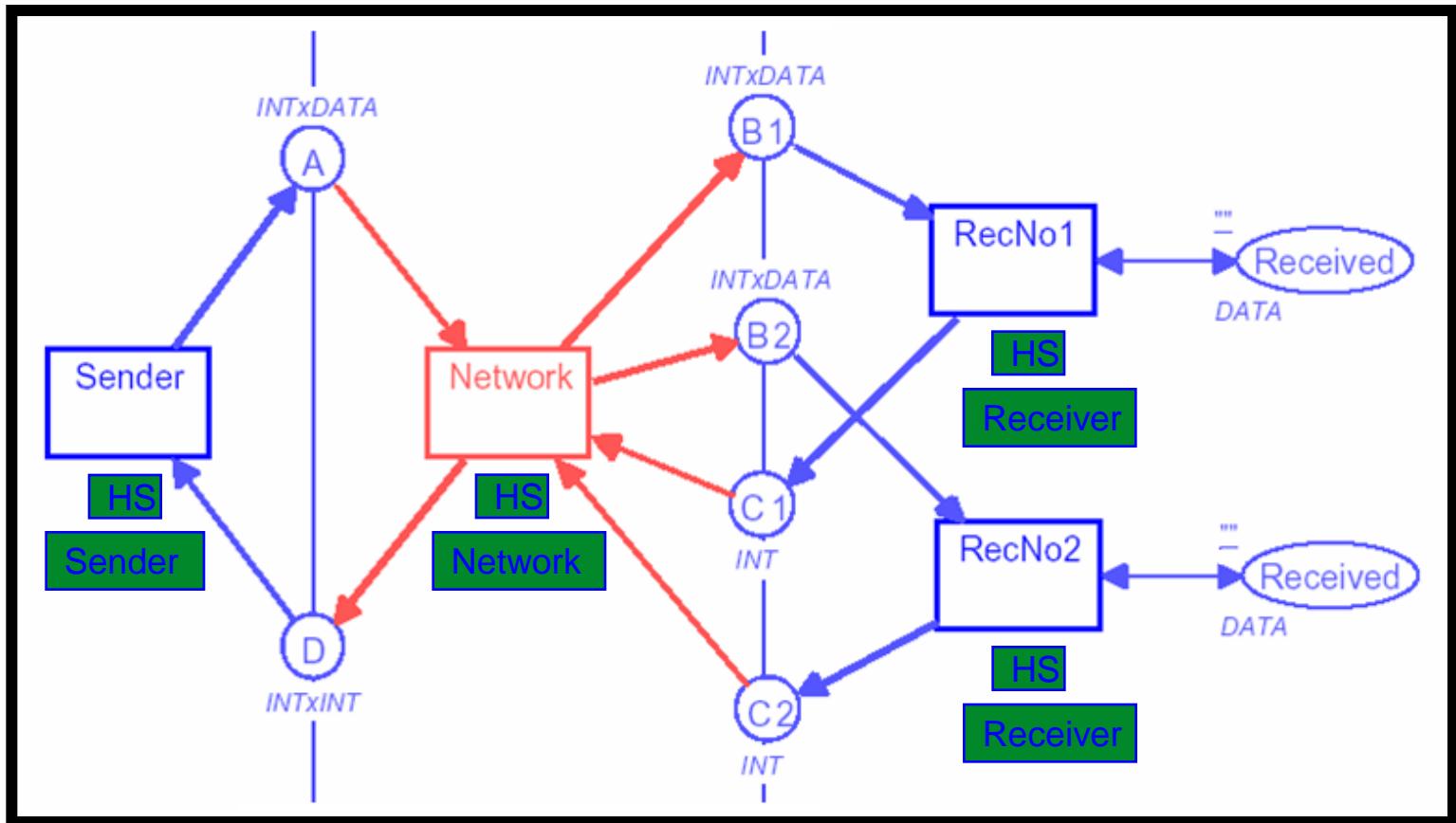


Substitution transitions refer to modules.

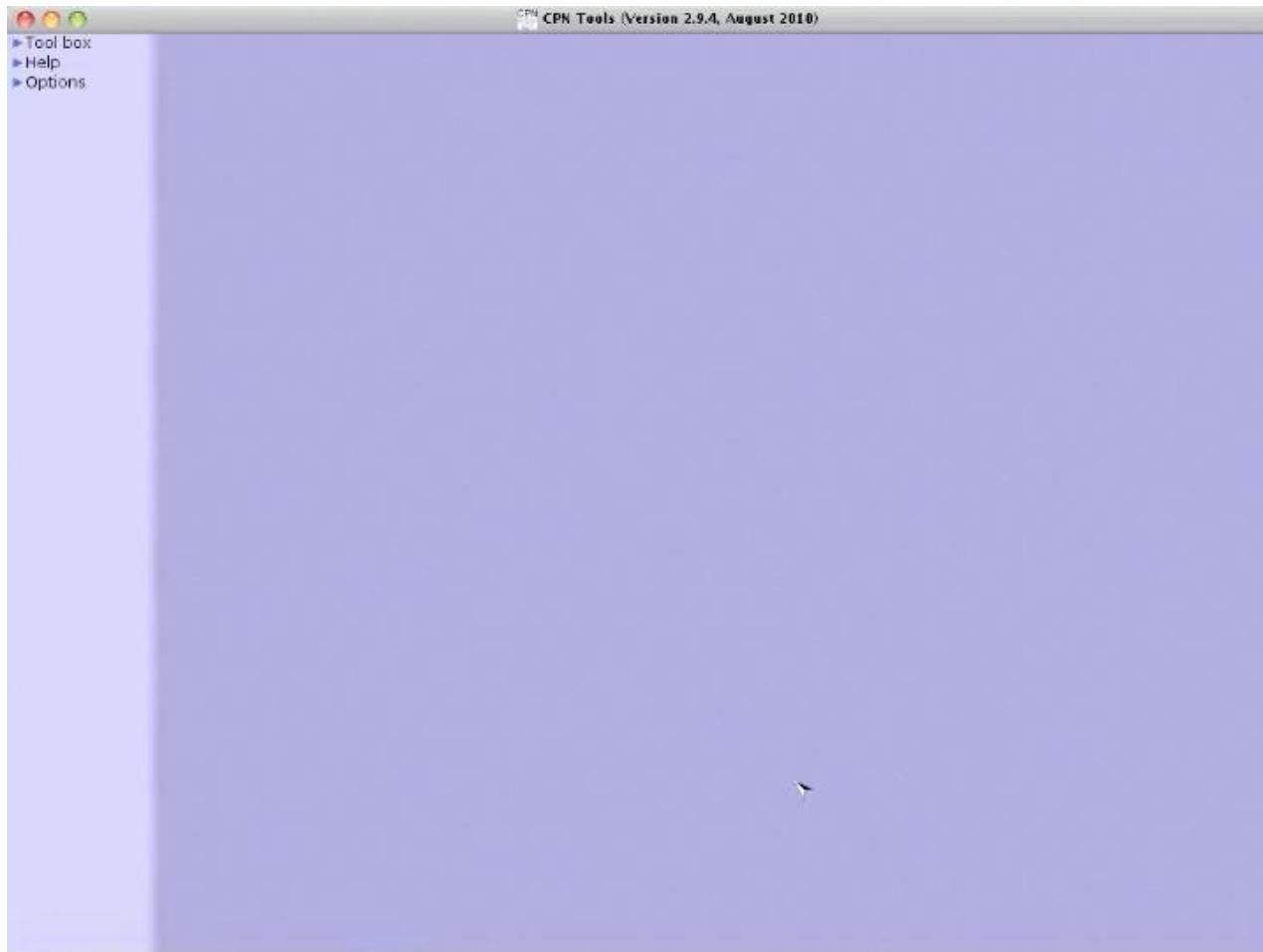
Socket places are related to port places.

Modules can be reused

Protocol



CPN Tool Simulation

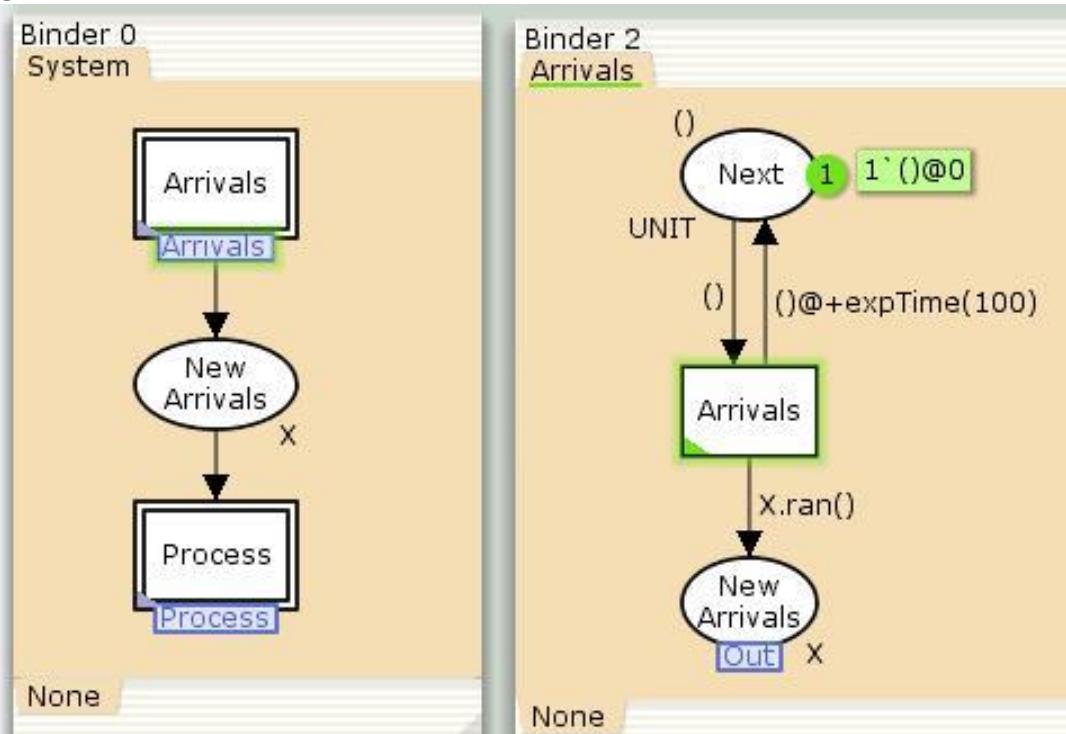


UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Data processing with time stamps

- New objects arrive **regularly** and need to be processed by some kind of resource.

```
Step: 0
Time: 0
▶ Options
▶ History
▼ Declarations
  ▼ colset UNIT = unit timed;
  ▼ colset X = with A | B timed;
  ▼ var x : X;
  ▼ colset RESOURCE = with r timed;
  ▶ fun expTime
  ▶ fun intTime
  ▼ val avg_proc_time = 150;
▶ Monitors
▼ System
  Arrivals
  ▼ Process
    Task (1)
    Task (2)
```

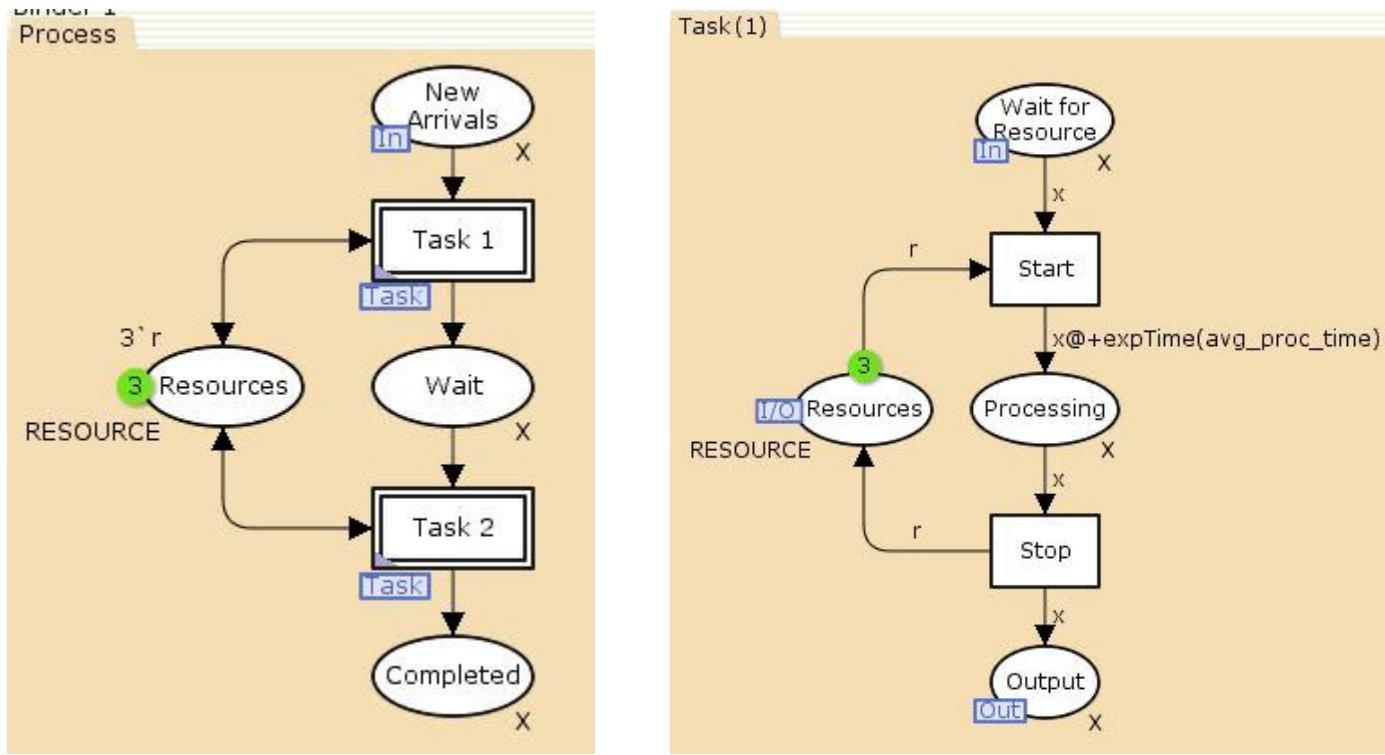


- @0: a token with the time stamp '0'
- expTime(100): a function to generate exponentially distributed inter-arrival times with an average time of 100



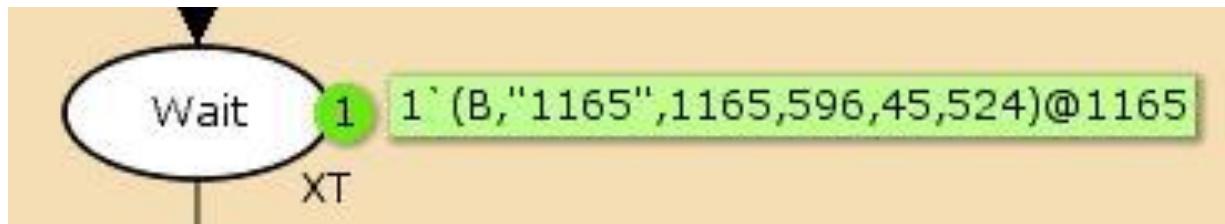
Data processing with time stamps

- The process contains two tasks and the processing time is also considered.
- The time stamps on the tokens representing objects on place Waiting for Resource and resources on place Resource determine when a resource can start processing the object.



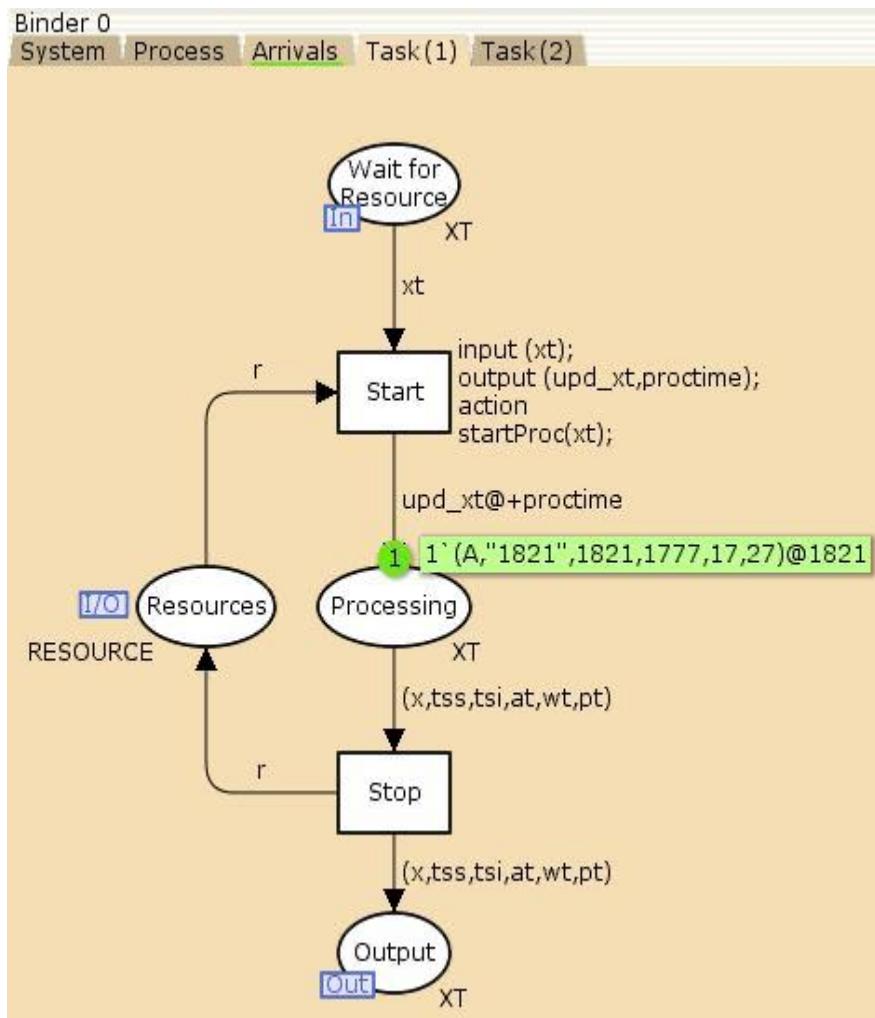
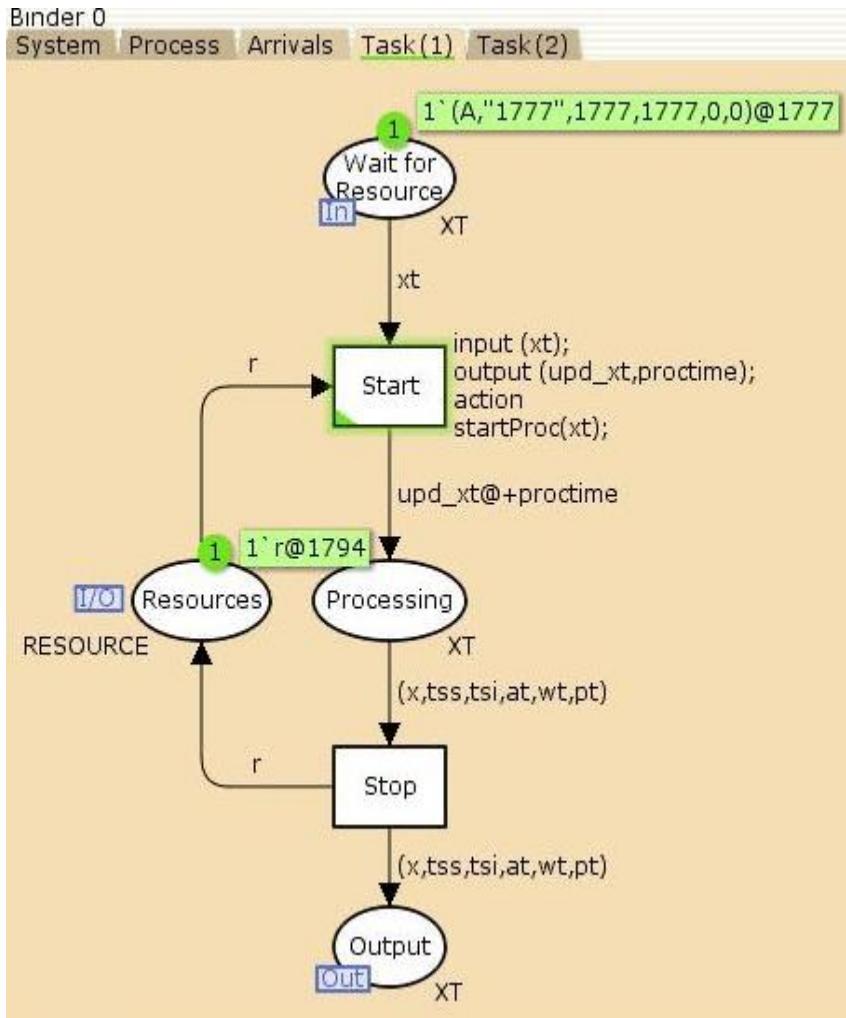
Data processing with time stamps

- A token representing an object together with time attributes.
- The token is on place Wait, i.e., the object has been processed in task 1, and it is waiting to be processed in task 2.



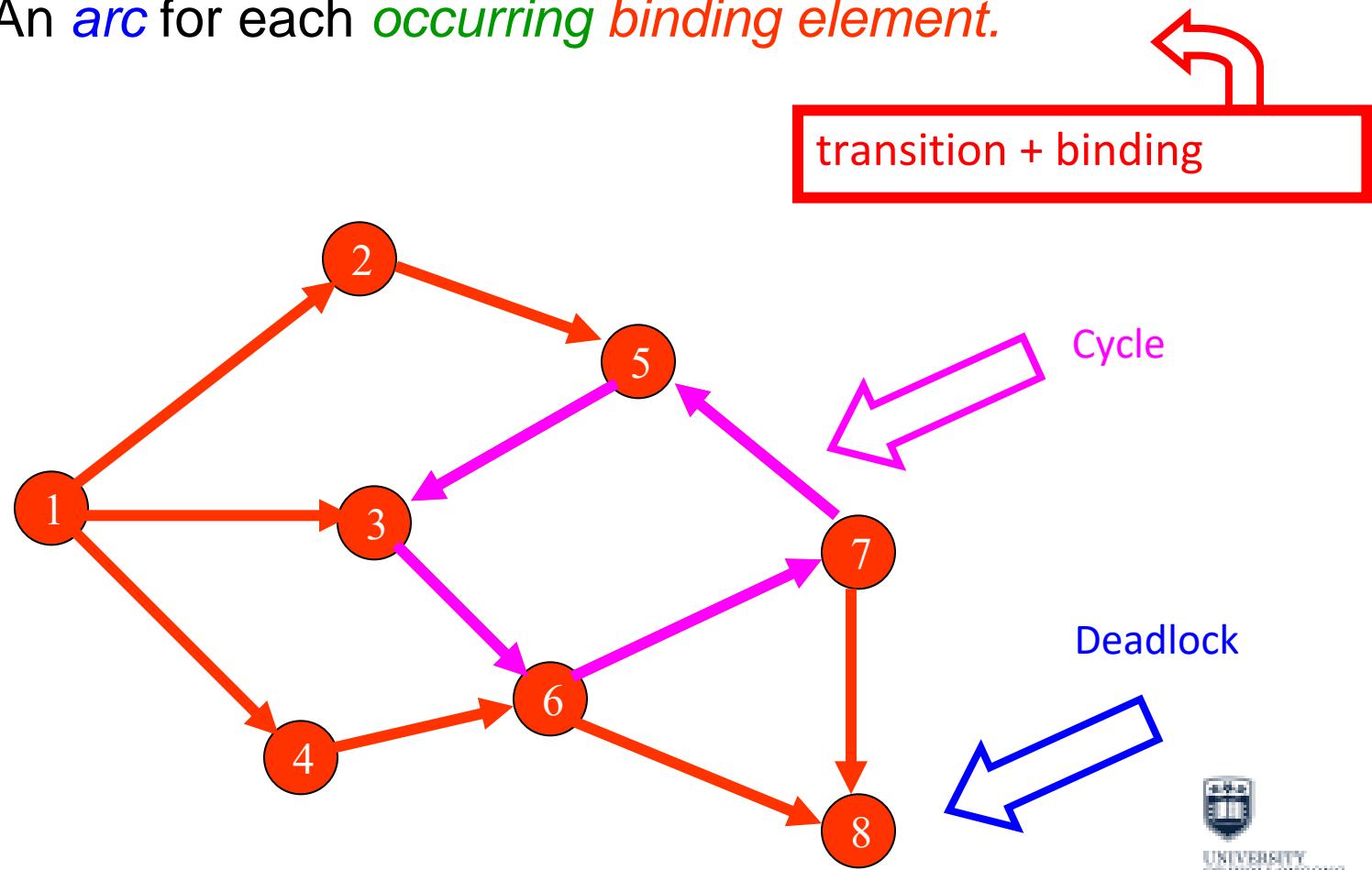
- The object is kind B.
- The time stamp is represented as a string "1165" and an integer 1165 is the token value.
- The object arrived at time 596, has waited 45 units of time before processed by a resource in task 1.
- The processing time for the object in task 1 was 524 units of time.

Data processing with time stamps

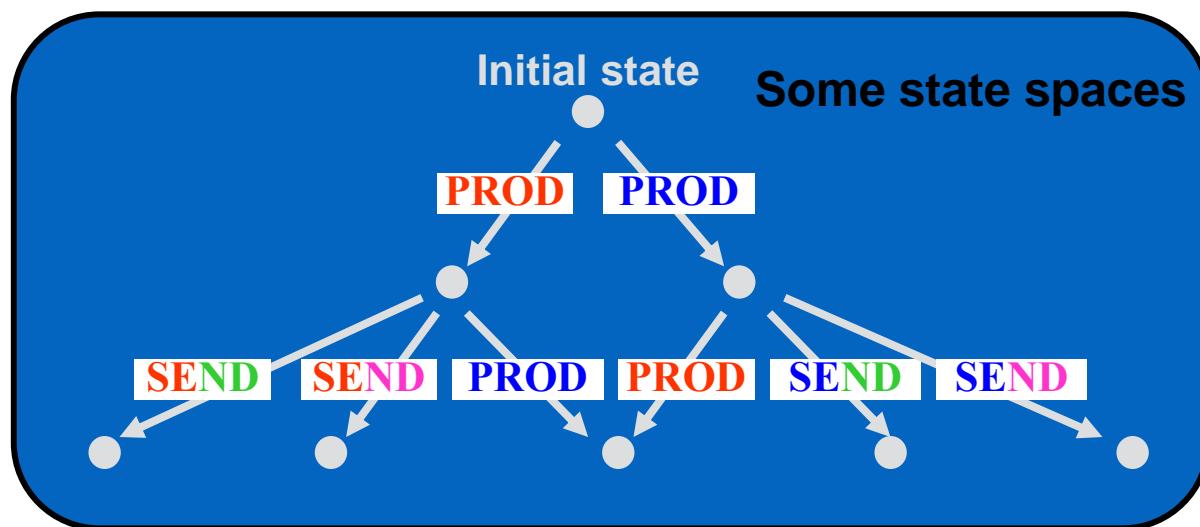
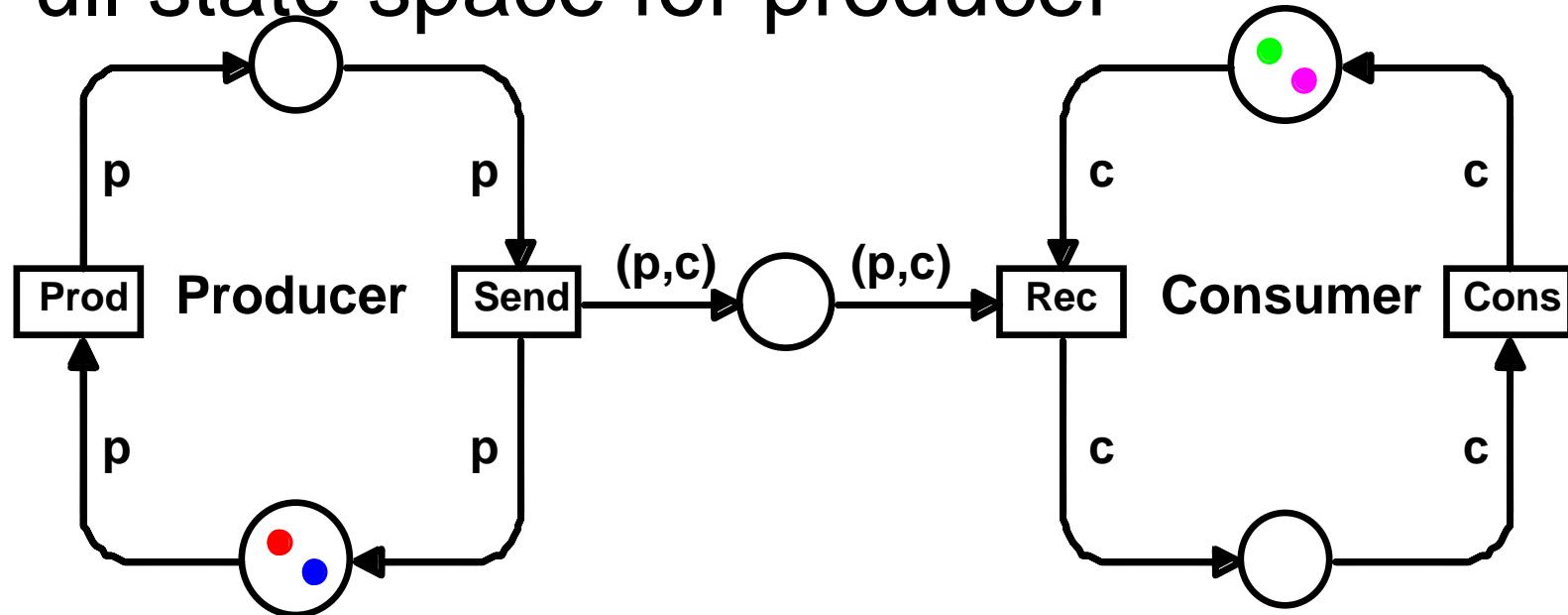


State spaces

- A *state space* is a *directed graph* with:
 - A *node* for each *reachable marking* (i.e., state).
 - An *arc* for each *occurring binding element*.



Full state space for producer



CPN Tool State-Space Analysis



UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Conclusion

- One of the *reasons* for the *success* of CP-nets is the fact that we *simultaneously* have worked in *all three areas*.
 - THEORY
 - models
 - basic concepts
 - analysis methods
 - TOOLS
 - editing
 - simulation
 - verification
 - PRACTICAL USE
 - specification
 - validation
 - verification
 - implementation