

A Metabolic Approach to Protocol Resilience

Christian Tschudin, University of Basel. *Joint work with*

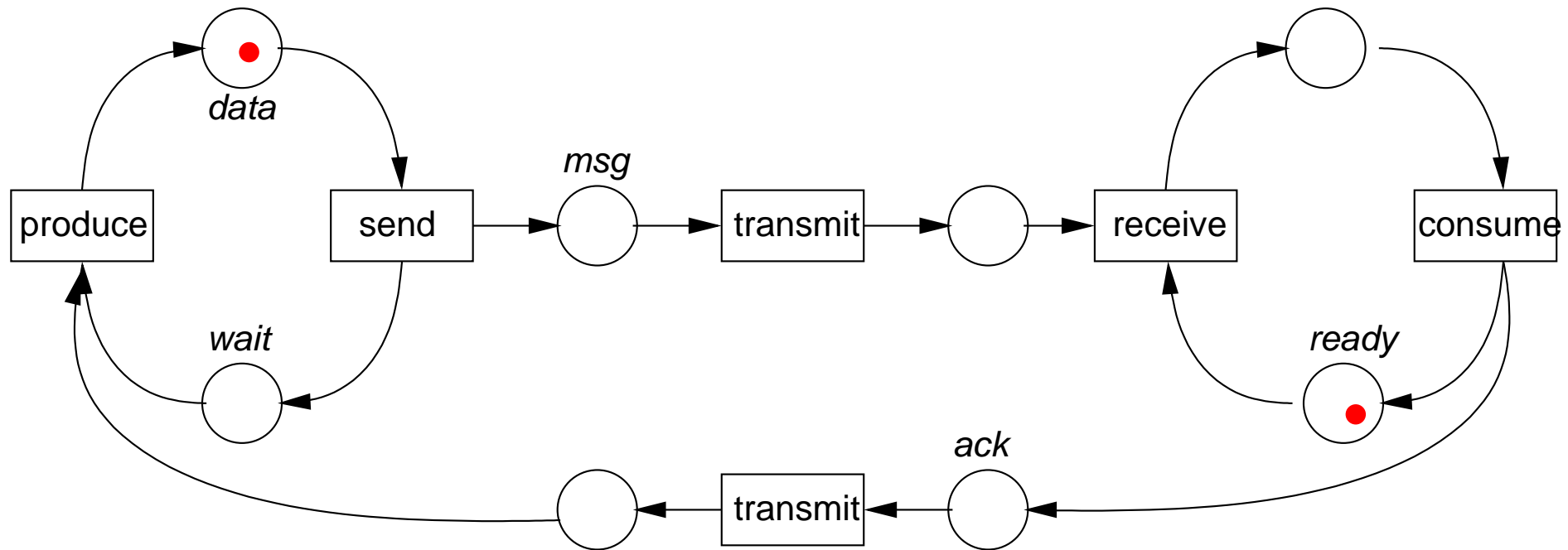
Lidia Yamamoto, Hitachi Europe, Sophia Antipolis

Theme: Self-Healing Protocol Implementations

- Problem statement: partial code deletion
- Examples of instruction-level resilience
- Related work
- Significance, Outlook

WAC2004, Berlin, Oct 18-19, 2004

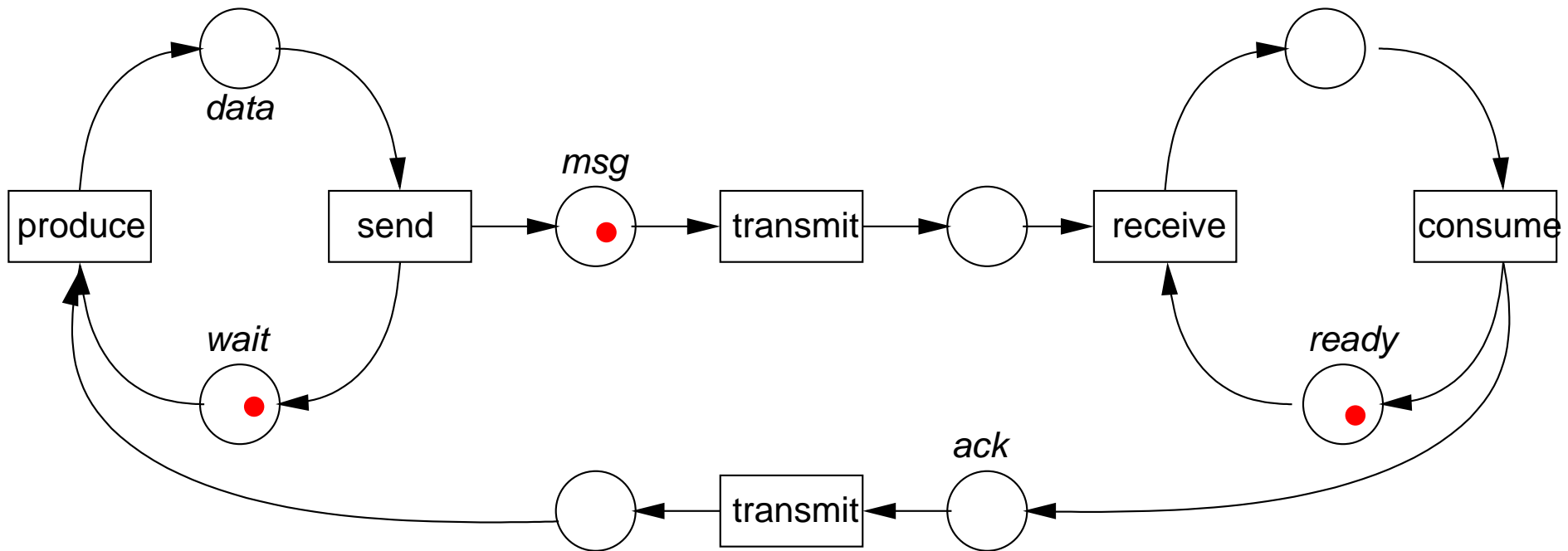
Introduction: Producer/Consumer, shown as a Petri-Net



boxes = transitions = actions = code

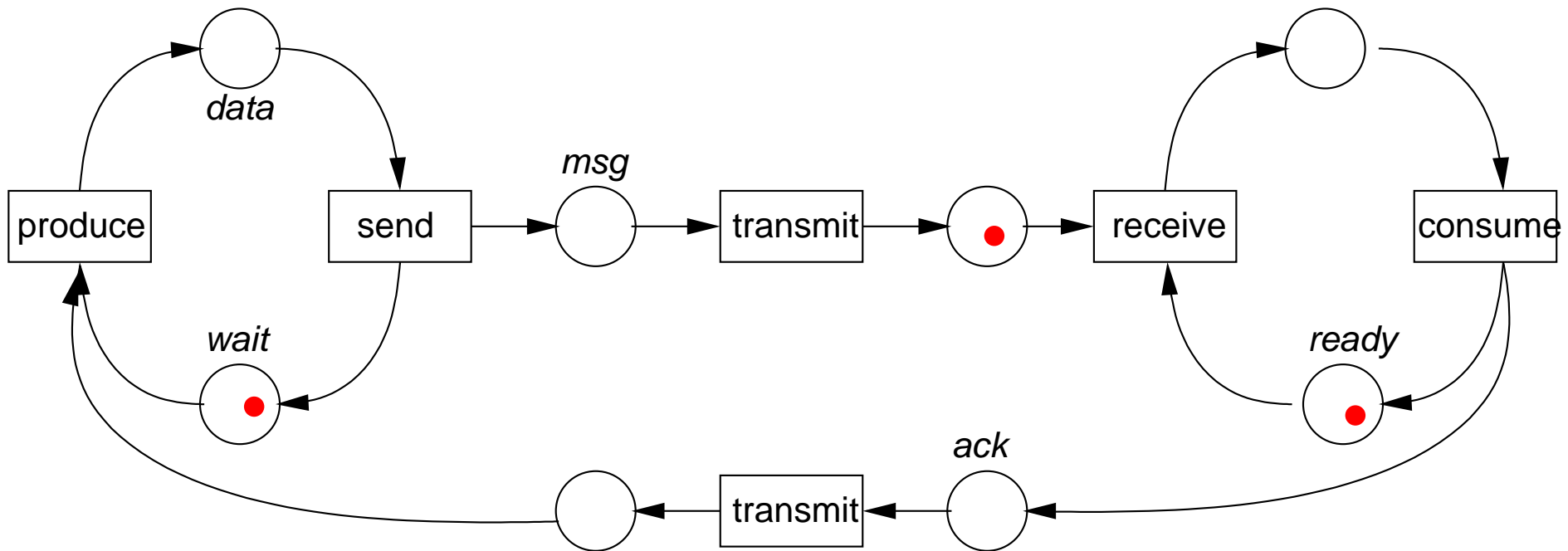
circles = places = state

Producer/Consumer, cont 1



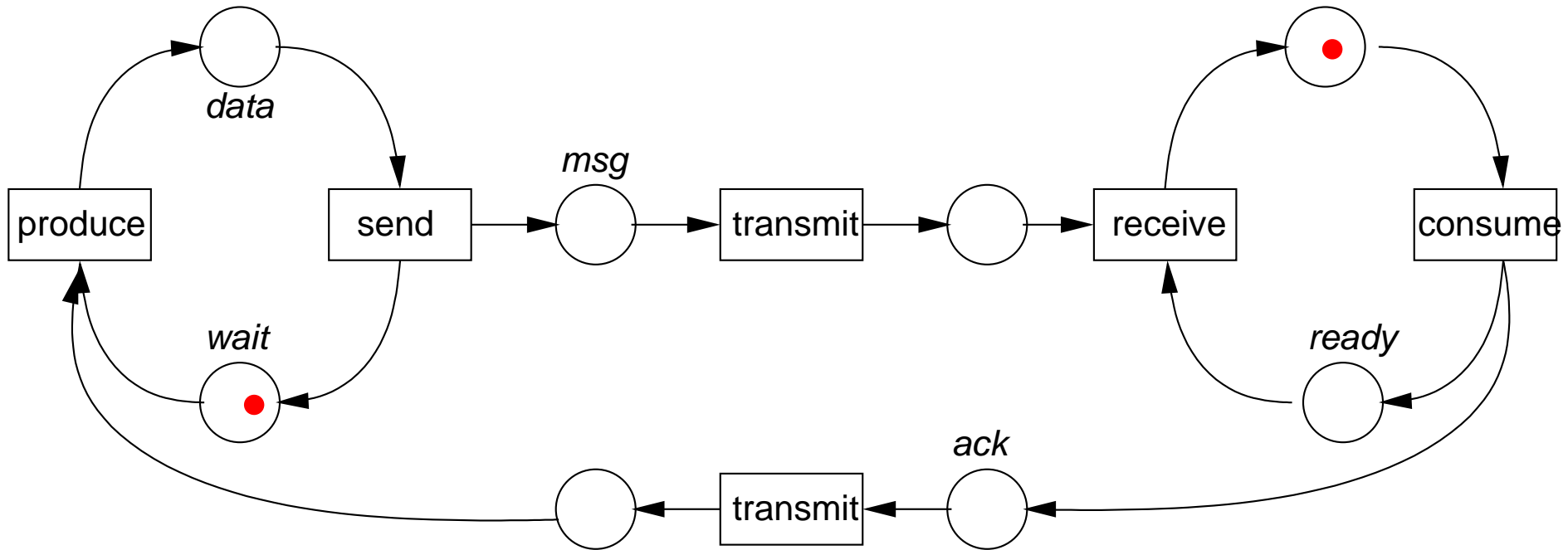
The producer goes into wait state after sending

Producer/Consumer, cont 2



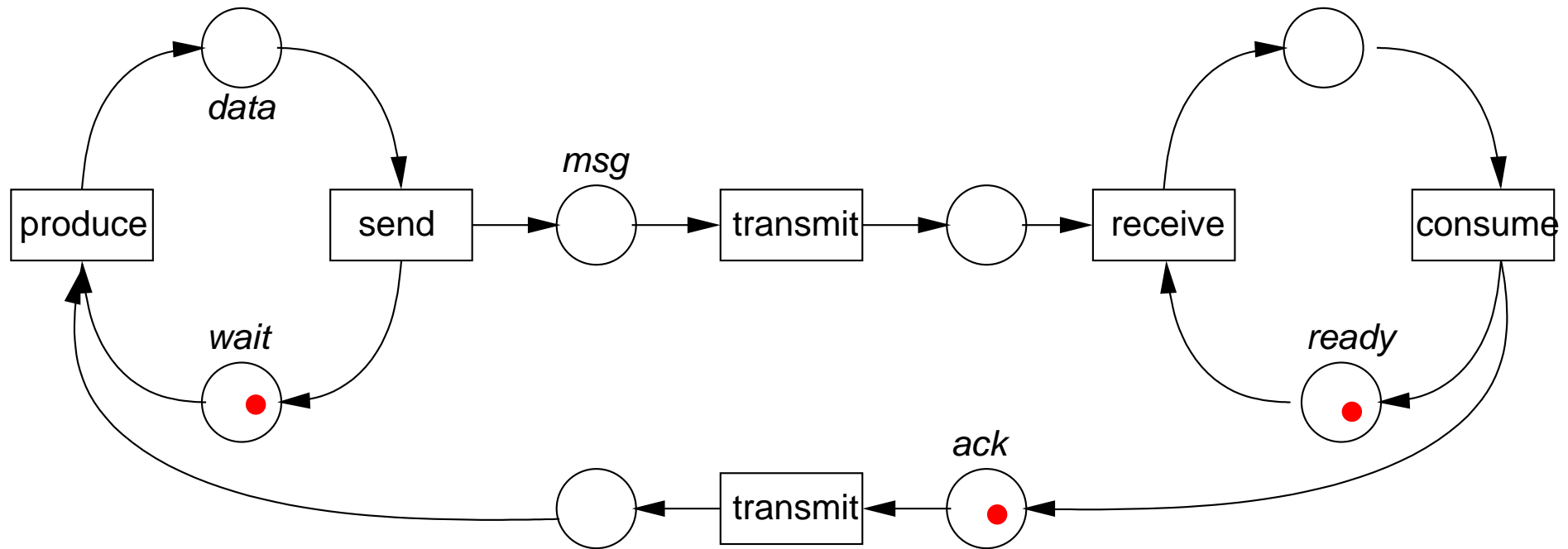
The channel transmits the message

Producer/Consumer, cont 3



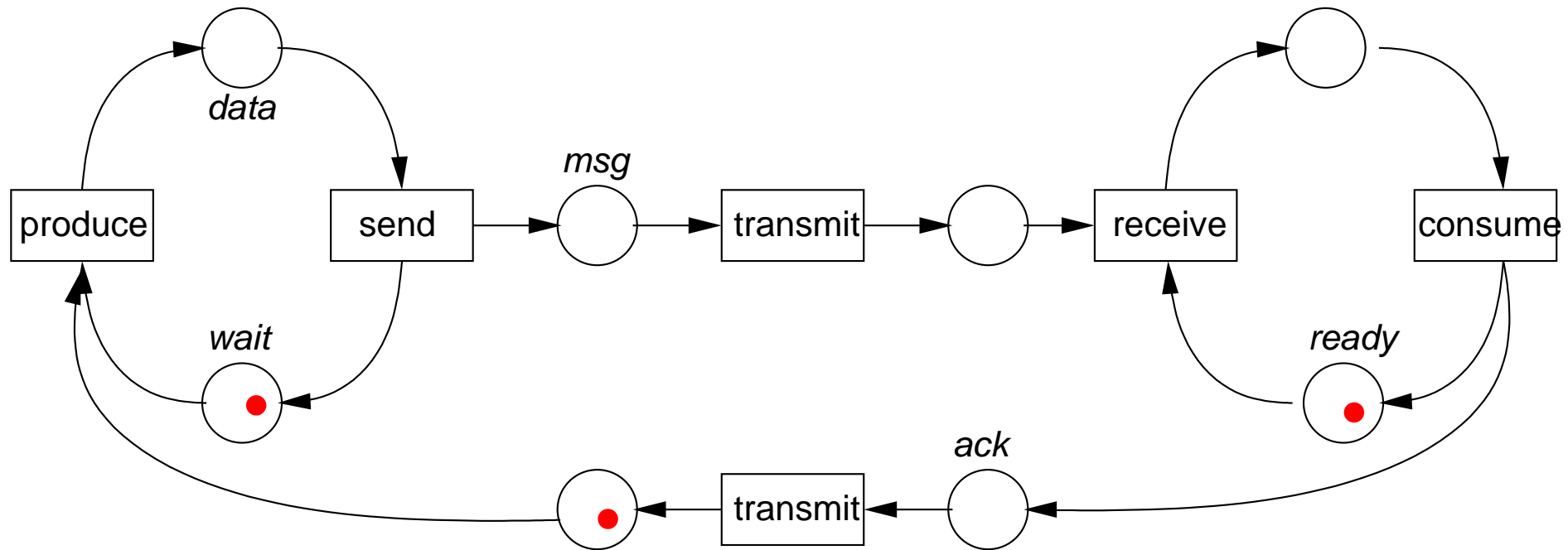
The consumer is ready and receives the message

Producer/Consumer, cont 4



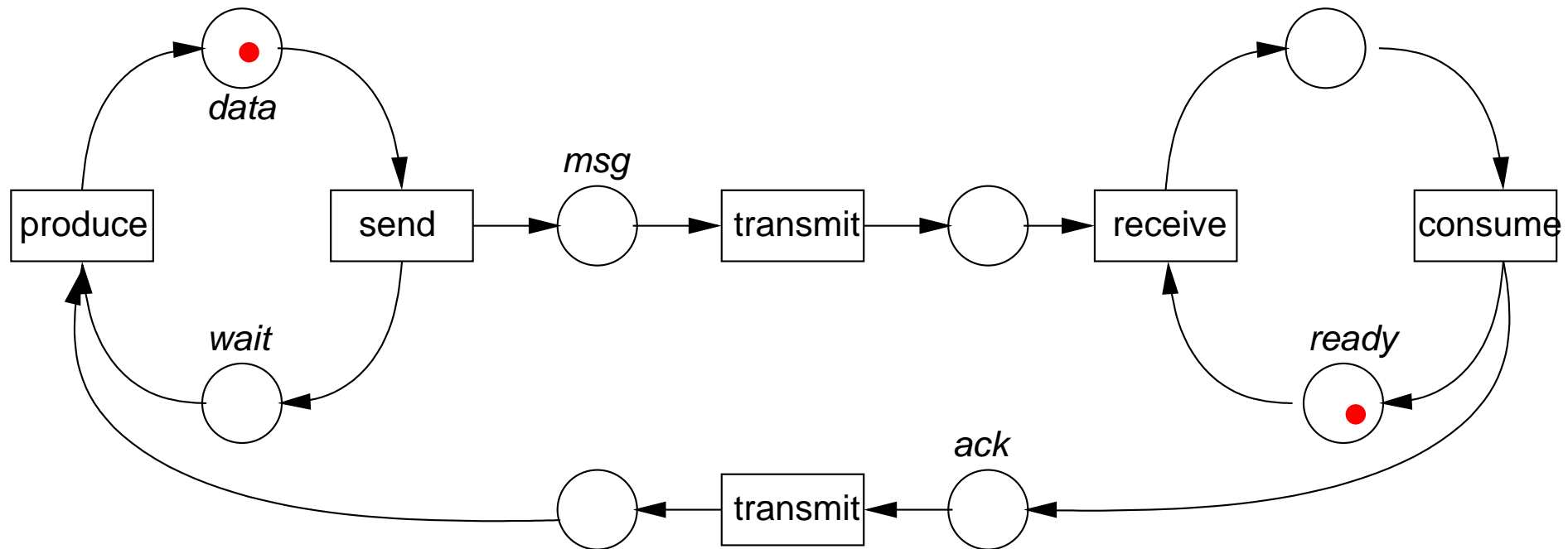
The message is consumed:
an ack message is generated, the consumer is ready again

Producer/Consumer, cont 5



With the ack transmitted, the producer will be able to generate new data, the cycle restarts

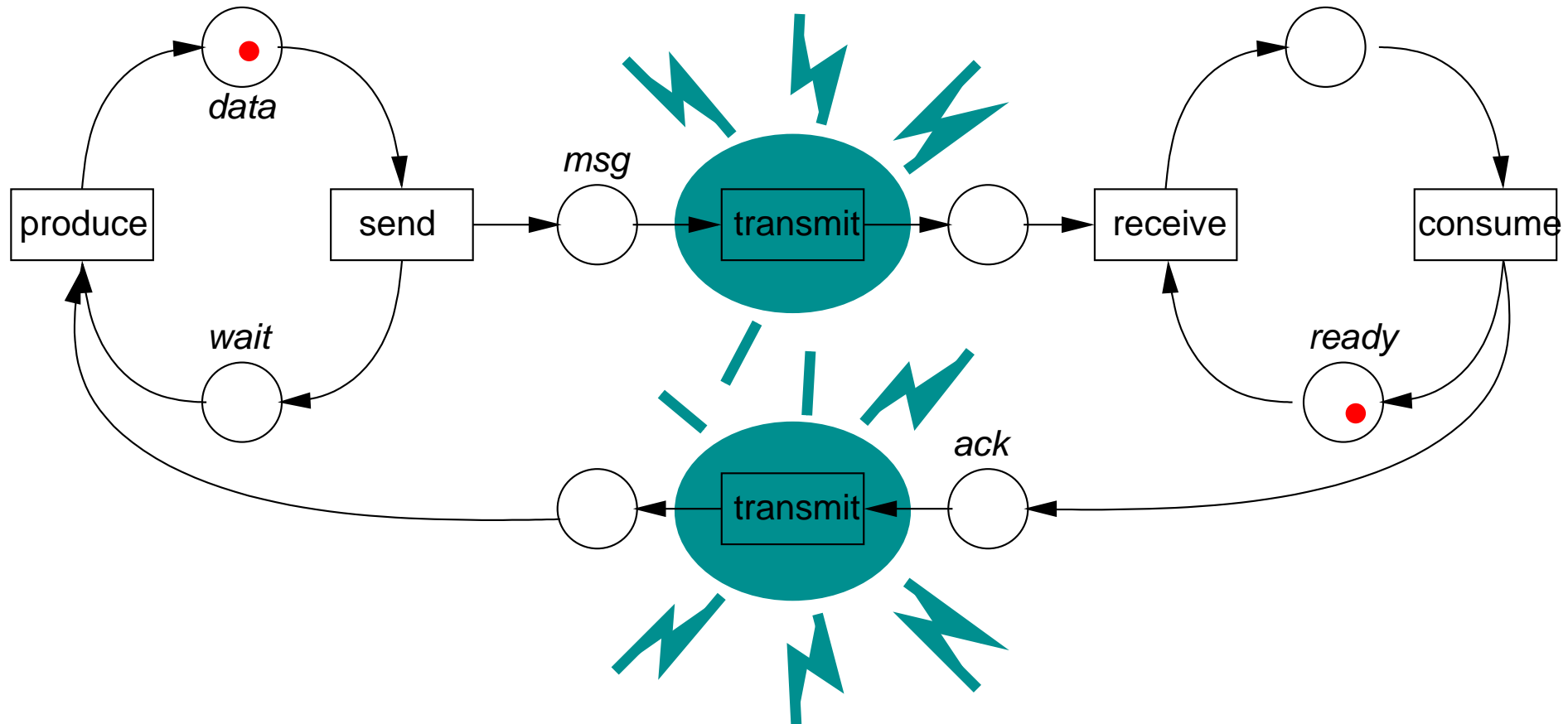
Producer/Consumer, cont 6



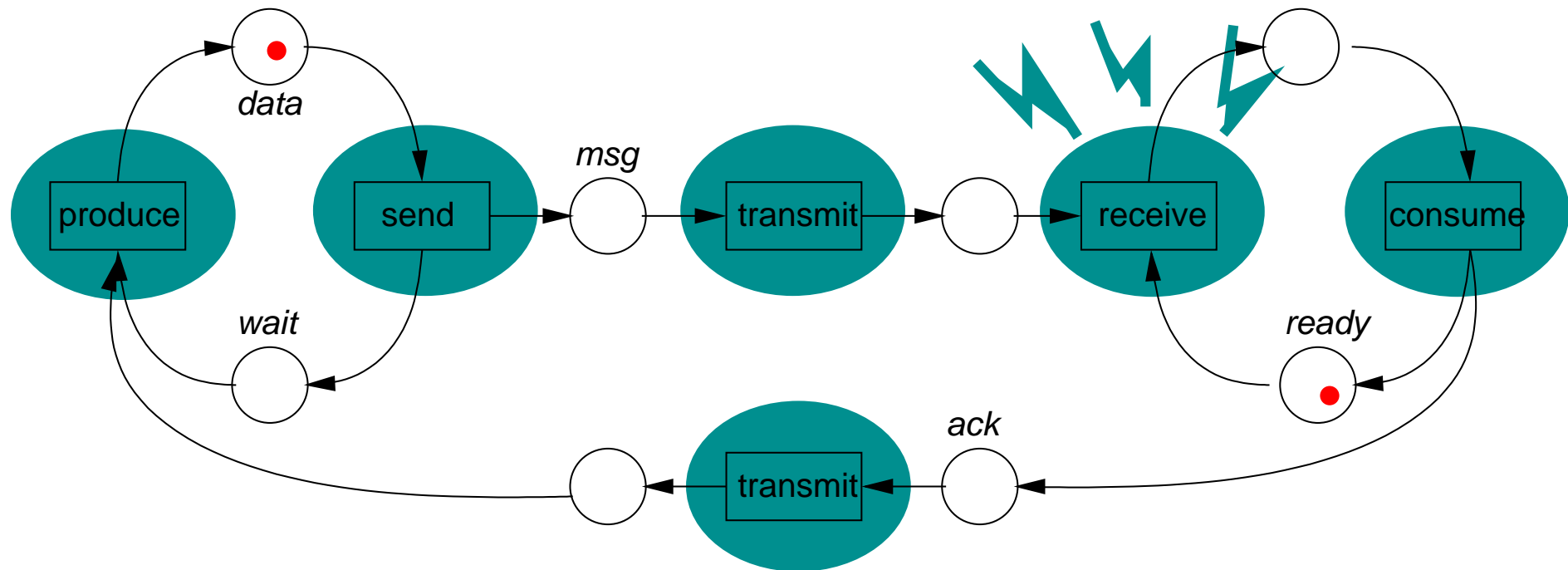
We are back to the initial config, the cycle restarts.

“transmit” failures: problem statement for *reliable transfer protocols*

Why Should Failable “transmit” Actions be Special?



Be Prepared for ANY Action to Fail !



New problem statement: “code–failure–resilient protocols”

Possible failure reasons: missing/lost code, unfaithful execution...

Self-Healing Protocols – a Wish List

Protocol implementations should be able ...

1. to detect their improper execution (diagnosis)
2. to continue proper operation despite (a limited amount of) execution errors (resilience)
3. to regain full operation and resilience (healing)

Two Examples for Resilience and Diagnosis

Approach taken for this research:

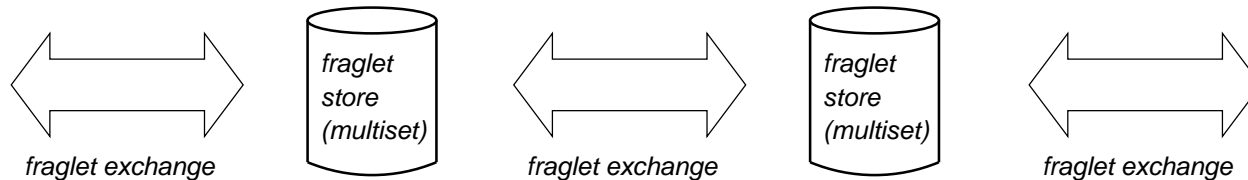
Resilience against the removal of any instruction of a protocol's implementation!

Overview of what will follow:

- Special execution environment:
 - set of condition-action rules (Fraglets)
- Case 1: Message Doubler
- Case 2: Producer/Consumer

Fraglets

Fraglet = computation fragment = “packet” = “rule” = “molecule”



- Fraglet = string W of symbols:
 $W = [s_0 : s_1 : \dots : s_n]$ (sequence of pkt headers)
- Head symbol selects fraglet processing \rightarrow fraglet rewriting
- Two types of fraglet processing (shall be $O(1)$):
 - single fraglet transformation
 - chemical reaction (involving two fraglets)

A Fraglet Language

- Unary operation – special symbols *pop*, *send* etc:
(*V* and *W* are words)

$$[\textit{discard} : V] \rightarrow \emptyset$$

$$[\textit{pop} : t : s_i : V] \rightarrow [t : V]$$

$$[\textit{split} : V : * : W] \rightarrow [V] \text{ and } [W] \text{ (} V \text{ has no } *)$$

$$[\textit{send} : n : V] \rightarrow \text{send } [V] \text{ to node } n$$

- Binary operation – two fraglets “react”:

$$\left. \begin{array}{l} [\textit{match} : s_0 : V] \\ [s_0 : W] \end{array} \right\} \rightarrow [V : W]$$

Sample “Fraglet Program”: Change a Fraglet’s Tag

In: [i : W]
Out: [o : W]

Program: [match : i : o]

Execution Trace:

$$\left. \begin{array}{l} [\text{match} : i : o] \\ [i : W] \end{array} \right\} \Rightarrow [o : W]$$

Sample “Fraglet Program”: Change Fraglet Tag (contd)

Persistent version of “match”: matchp

In: [i : W]

Out: [o : W]

Program: [matchp : i : o]

Execution Trace:

$$\left. \begin{array}{l} [\text{matchp} : i : o] \\ [i : W] \end{array} \right\} \Rightarrow [\text{matchp} : i : o], [o : W]$$

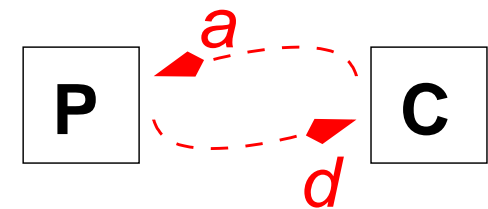
Sample “Fraglet Process” : Producer/Consumer

a.k.a “confirmed delivery protocol”

Prod: [matchp : ack : data : Payload]

Cons: [matchp : data : split : ack : * : discard]

Ack: [ack]



Execution Trace:

[a]

[matchp : a : d : Payload], [a] ⇒ [d : Payload]

[matchp : d : split : a : * : discard], [d : Payload] ⇒ [split : a : * : discard : Payload]

[split : a : * : discard : Payload] ⇒ [a], [discard : Payload]

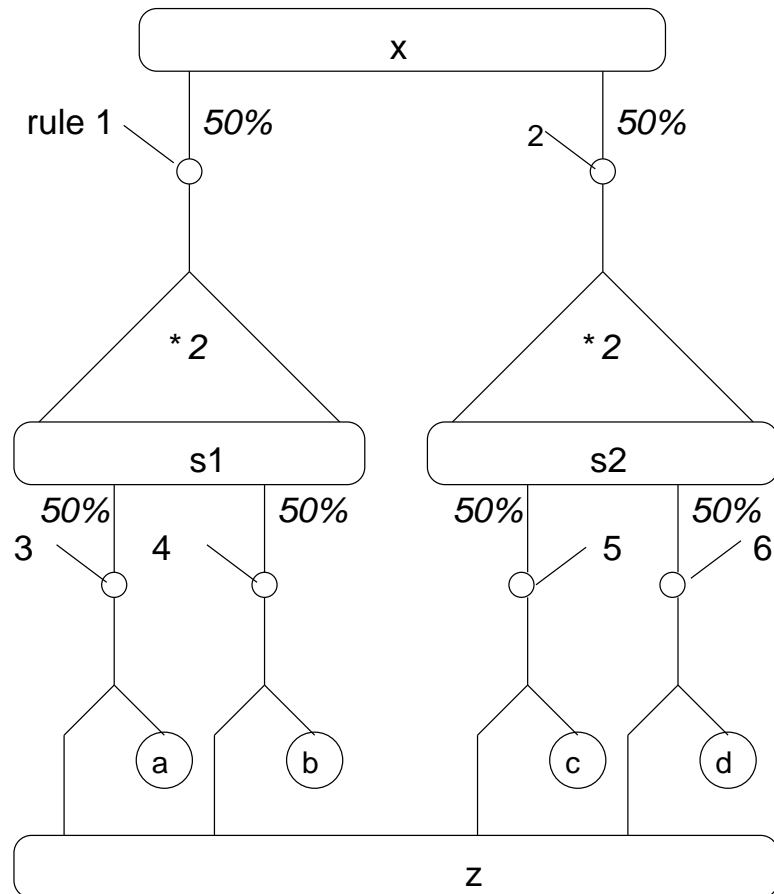
⇒ [a] ...

Case 1: Robust Packet Doubler



- Rate-encoded signal (# of pkt/s e.g., Geiger counter)
Doubler's task: to double the packet frequency
- A simple program (not robust):
[matchp : x : split : z : * : z]
- A version that works despite any single rule being deleted !
[matchp : x : split : s1 : * : s1] [matchp : x : split : s2 : * : s2]
[matchp : s1 : split : z : * : a] [matchp : s1 : split : z : * : b]
[matchp : s2 : split : z : * : c] [matchp : s2 : split : z : * : d]

Case 1: Robust Packet Doubler (contd)



- “Competing” rules: each has 50% load (normally)
- “Competing” rules: serve as backup (and take 100%)
⇒ const 'z' rate despite deletion
- Additional processing stage: “health signals” a,b,c,d
- If a+b is missing: rule1 knockout, only 'a' missing: rule3 knockout etc

Case 2: Robust Producer/Consumer

- Producer/Consumer as previously introduced (not robust):

Prod: [matchp : ack : data : Payload]

Cons: [matchp : data : split : ack : * : discard]

- Enhanced with sending between nodes A and B (not robust):

Prod: _A[matchp : ack : send : B : data : Payload]

Cons: _B[matchp : data : split : send : A : ack : * : discard]

- Robust version:

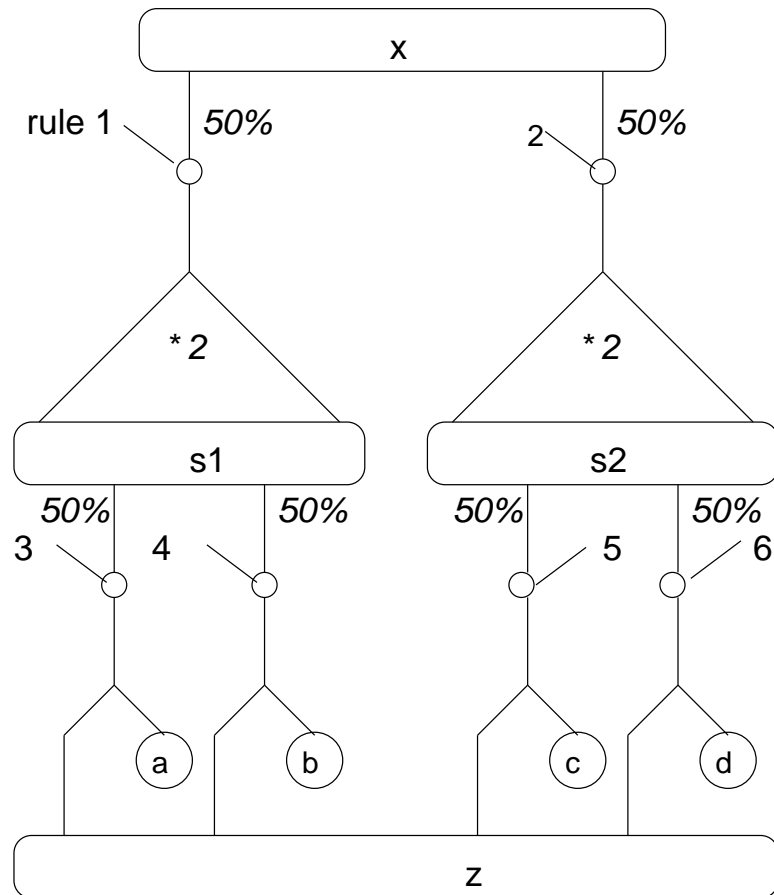
Prod: _A[matchp : ack : split : a : * : send : B : data : Payload]

_A[matchp : ack : split : b : * : send : B : data : Payload]

Cons: _B[matchp : data : split : c : * : split : send : A : ack : * : discard]

_B[matchp : data : split : d : * : split : send : A : ack : * : discard]

Self-Diagnosis (for the Robust Packet Doubler)



- “Health signals” (a,b,c,d):
should annihilate each other
if everything is OK
- Two additional rules:
[matchp : a : match : c]
[matchp : b : match : d]
- Not complete, though
 - needs to be made robust
 - also needs self-diagnosis
 - and self-healing . . .

Self-Healing

How to react on the “health signals”?

- Equilibrium as a health state:
(Differential) diagnosis, as shown on previous slide.
- Next step (work in progress):
To couple health signals with the (re-)creation of rules

One path: continuous rewrite (as it goes) of the protocol software

Alternate path: independent (self-healing) self-healer application

Related Work

Seek inspiration from biological systems at the metabolic level.

However, also look at:

- Error correcting quantum computing
robustness “in the clear” → “encoded protocols”
Limits of ECQC: fixed (hardware) circuits, primitive gates only
- Self-testing and -correcting functions:
However, this is an extrinsic approach, and by itself not robust
- Fault tolerance in distributed systems:
Again an extrinsic approach, not robust by itself.

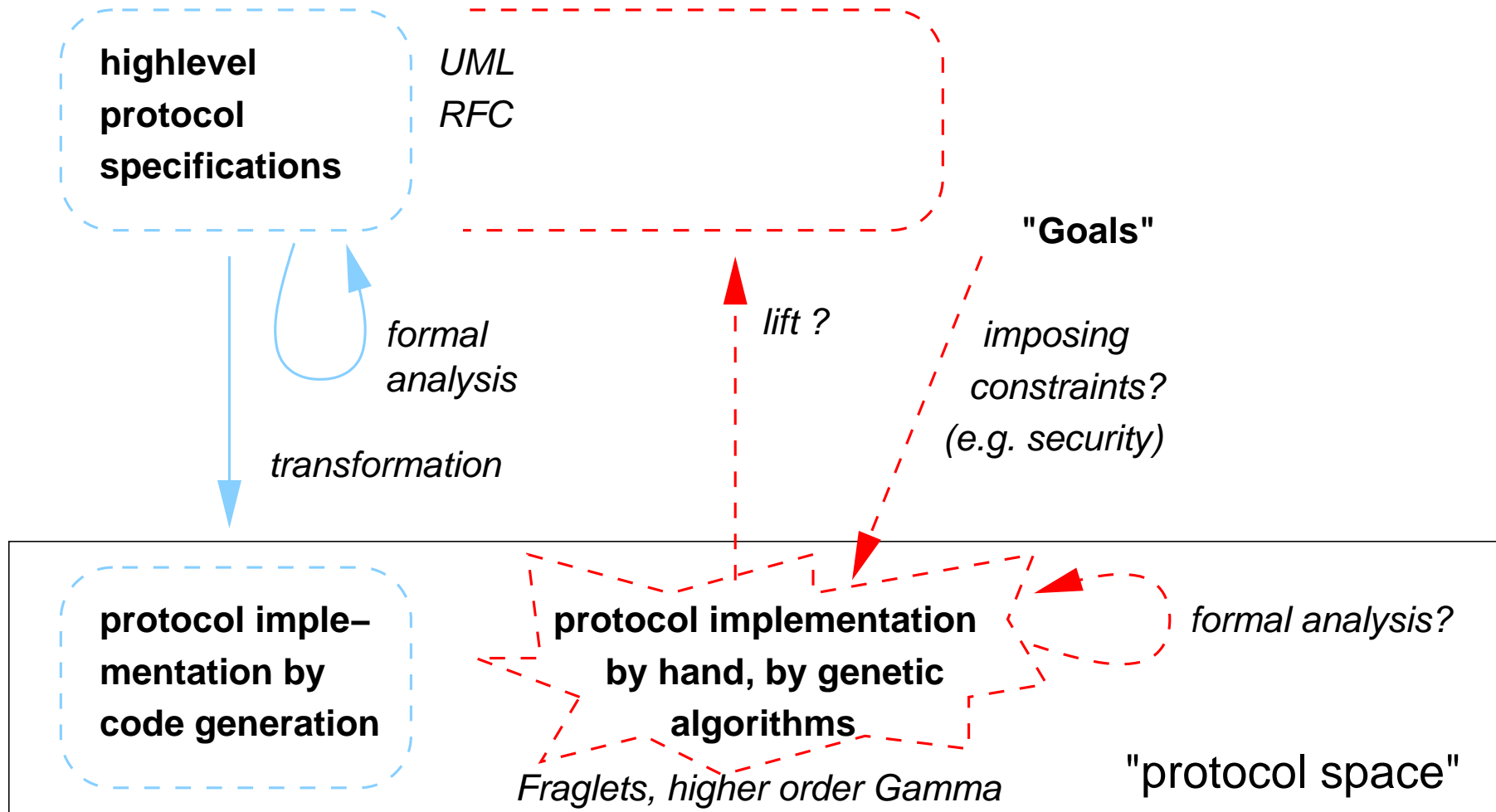
Progressing beyond Instruction Level Resilience

Need to raise the level of granularity, automate the process.

- **Protocol engineering techniques:**
 - handling transmission errors **and** code deletion (spray comp)
 - distributed protocol implementations (proxies for sensors)
- Grouping of instructions: **component level resilience**
 - automatic placements, backup, healing of “distr. servers”
- **Automatic protocol synthesis**
 - searching “program space” (protocols, not just impl.!)

Mobile code technology is key in Autonomic Comm.

Autonomic Communications in Protocol Space



Summary and Outlook

It's about mobile code!

- Study resilience through instruction knockout (bottom up)
- There **exist** protocol implementations which are resilient to partial code deletion! Negative results next?
- Raising the abstraction level:
resilient composition of workflows among components
(component = node, or module, or function etc)
- Self-healing: either continuous code rewrite,
or a standalone self-healer. → code dynamics+control