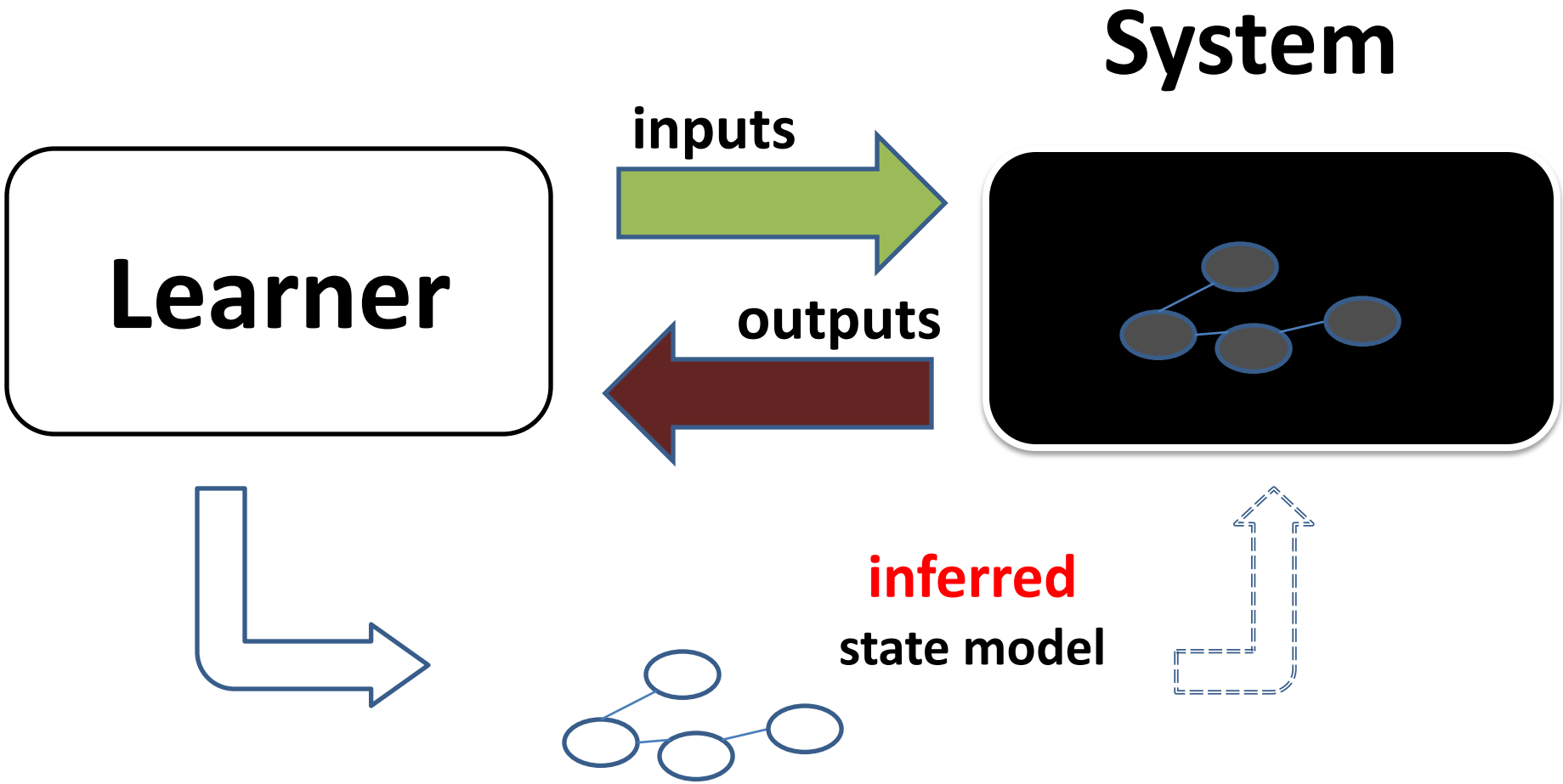
Learning fragments of the TCP communication protocol

using abstraction

Paul Fiterau,
Frits Vaandrager, Ramon Janssen

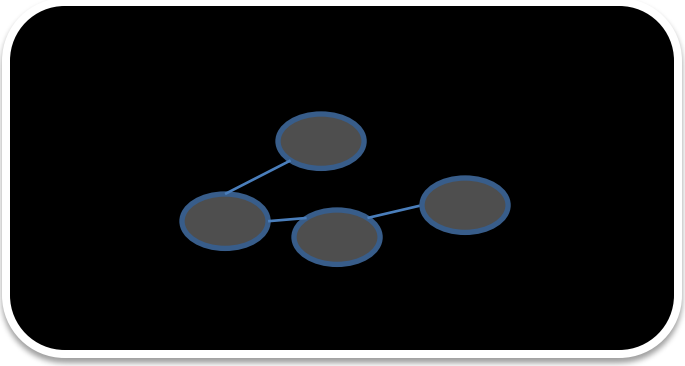
Radboud University

Introduction



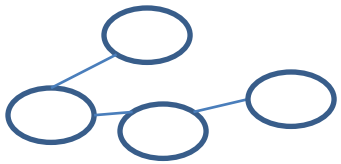
Introduction

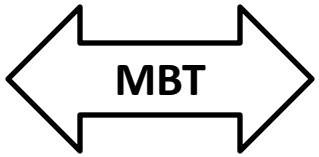
System



- automatic specification (systems underspecified)
- basis for model based testing + model checking

NEW System Version

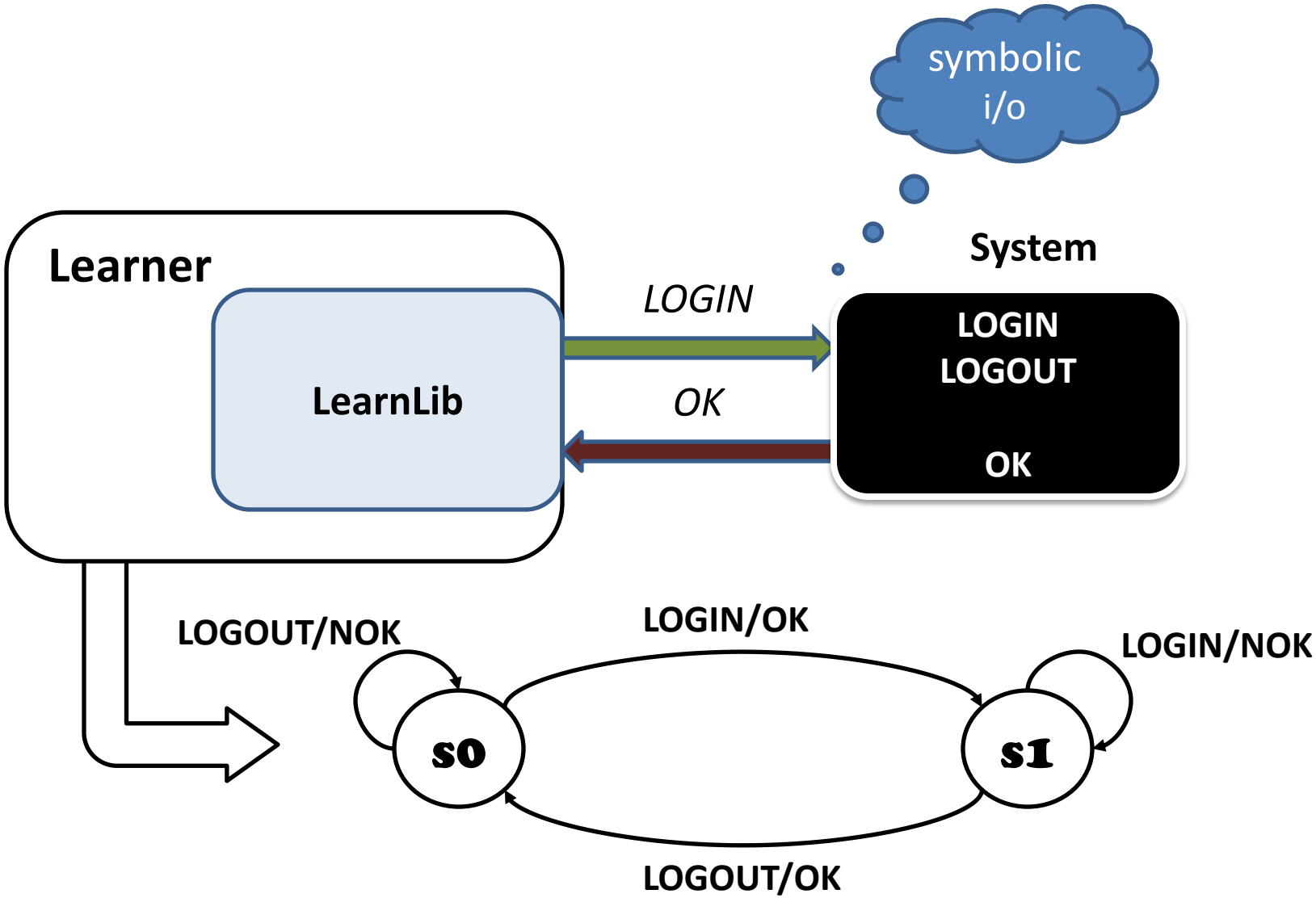




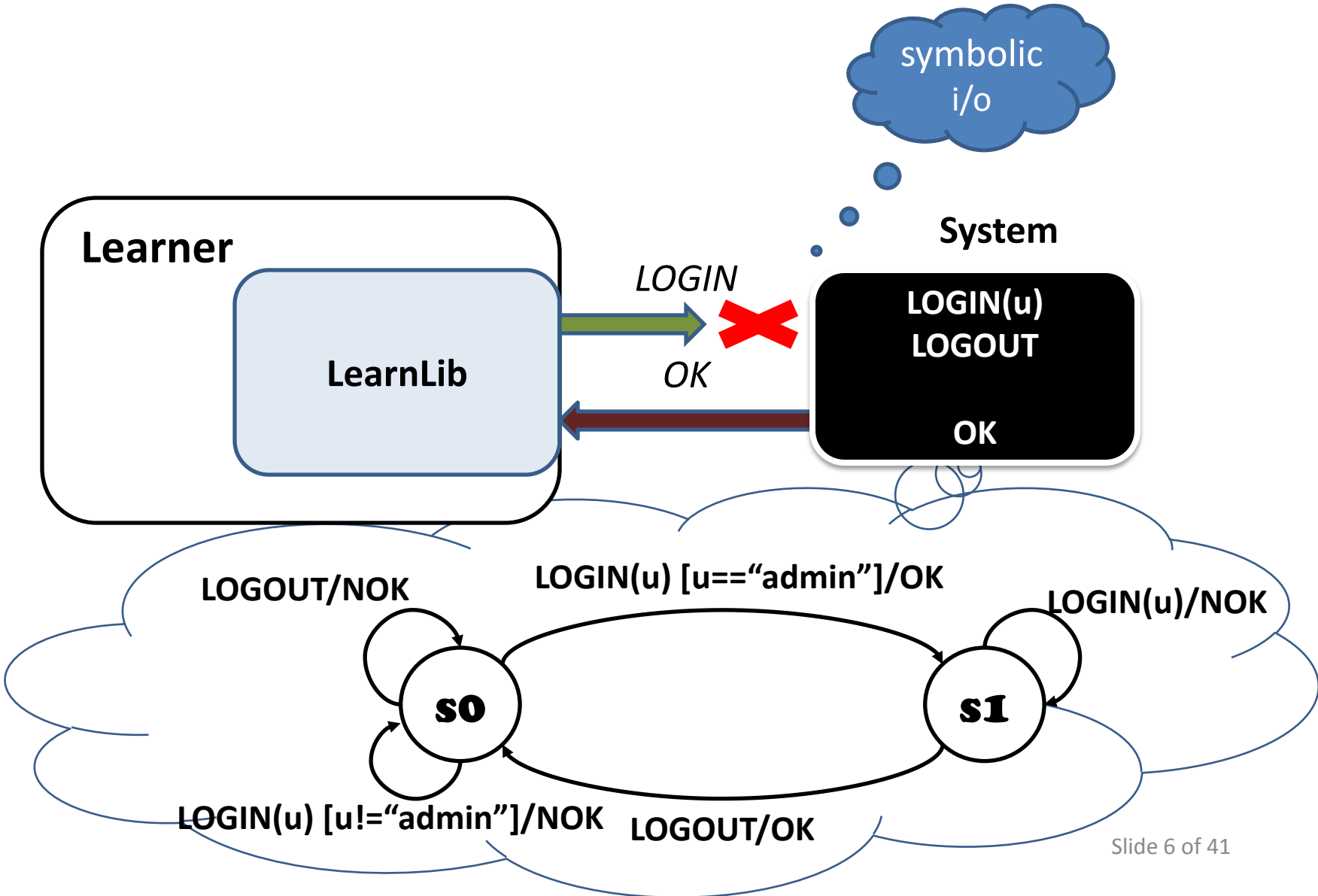
Goals

- apply learning techniques to a realistic system
  - TCP protocol implementations
- show why learning techniques are useful
  - analyze the models obtained against the specification
- establish what we would need to automate
  - milestones for mechanizing the learning process

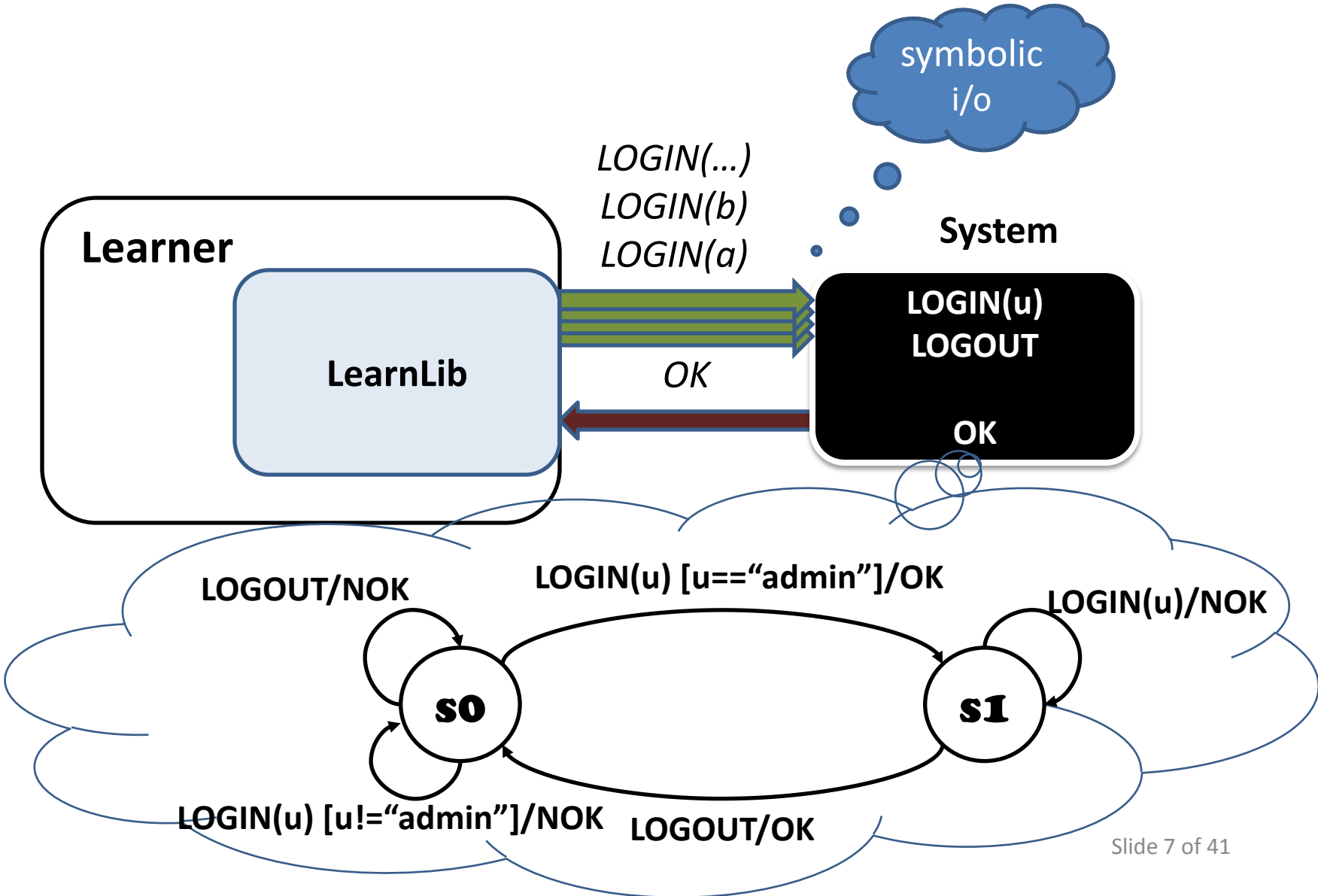
Classical learning



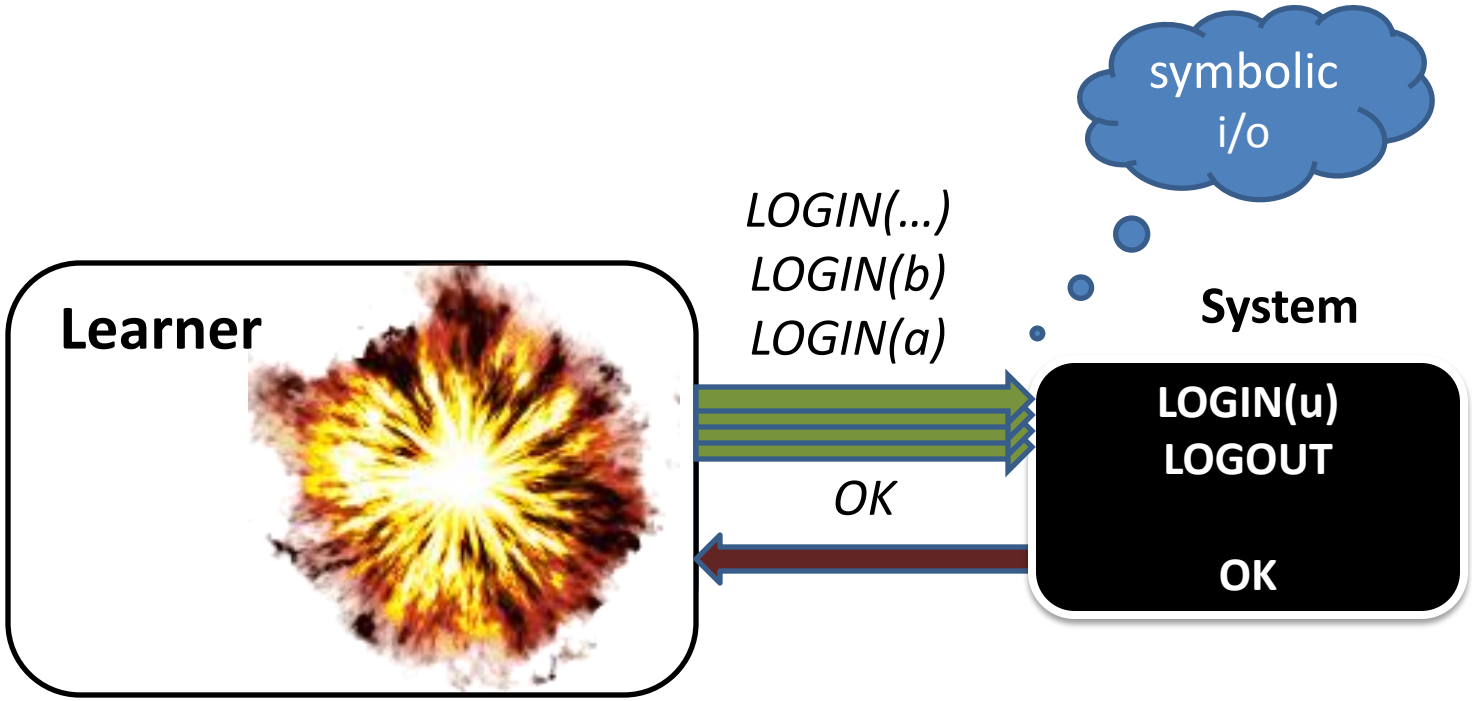
Classical learning



Classical learning

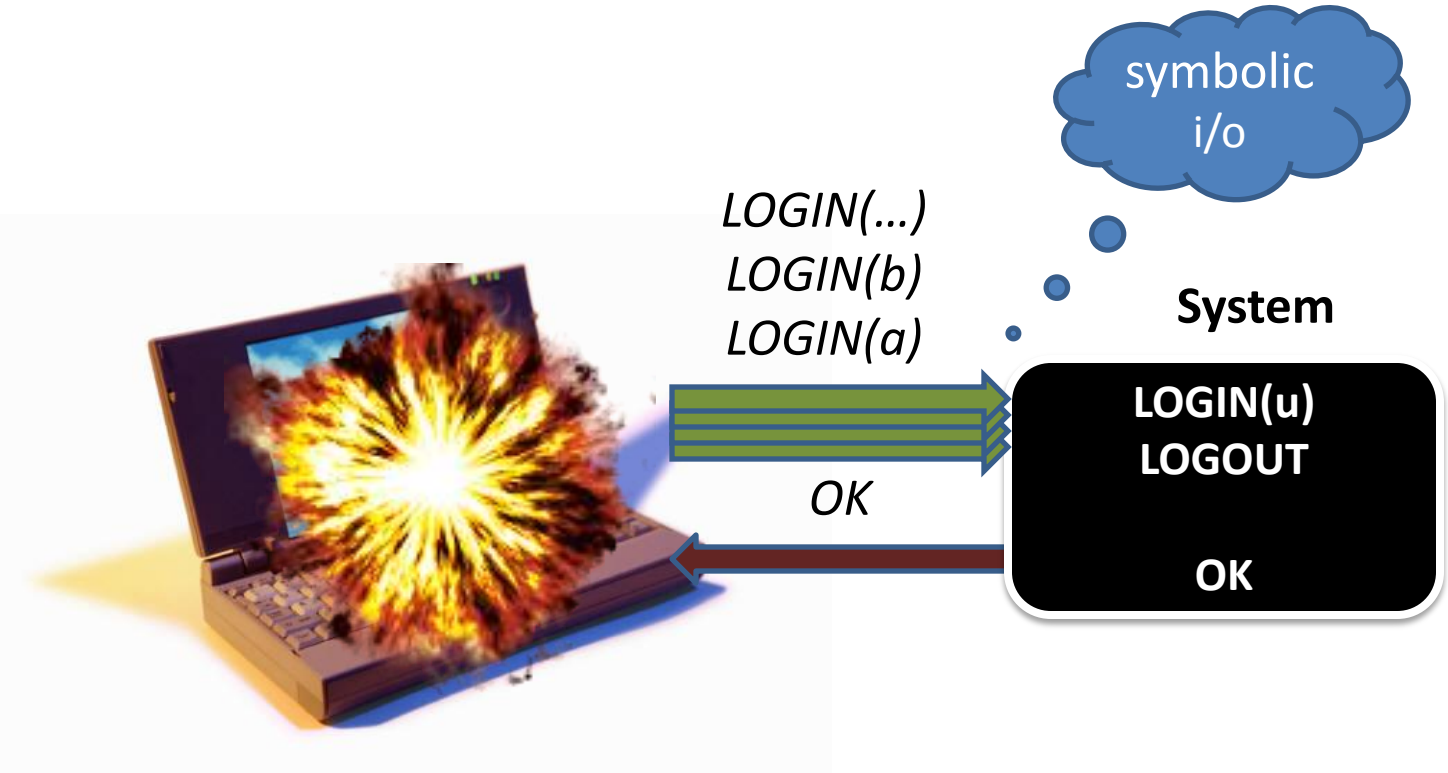


Classical learning



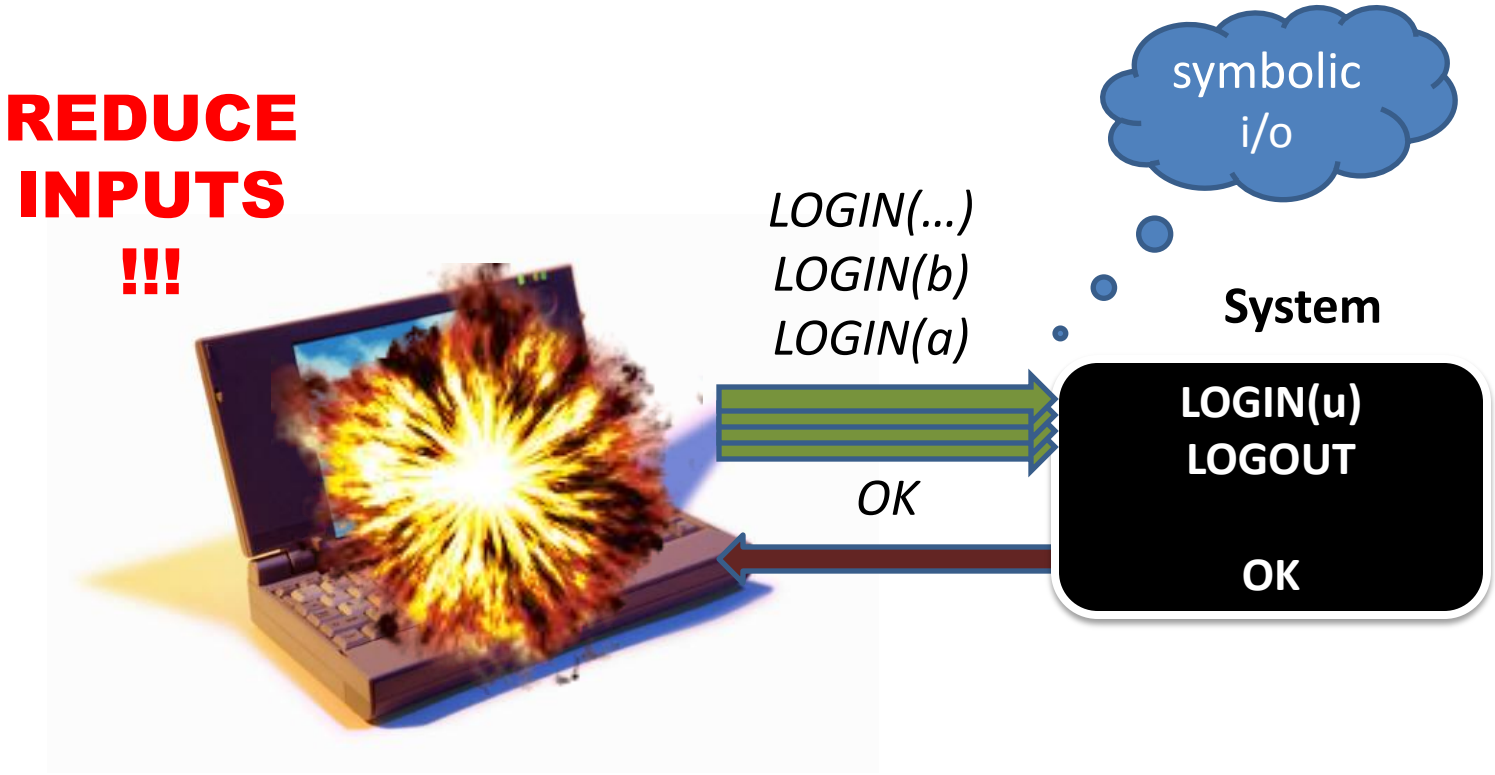
- input symbols for each possible parameter value
- too many for the Learner to handle
  - causes the Learner to explode

Classical learning

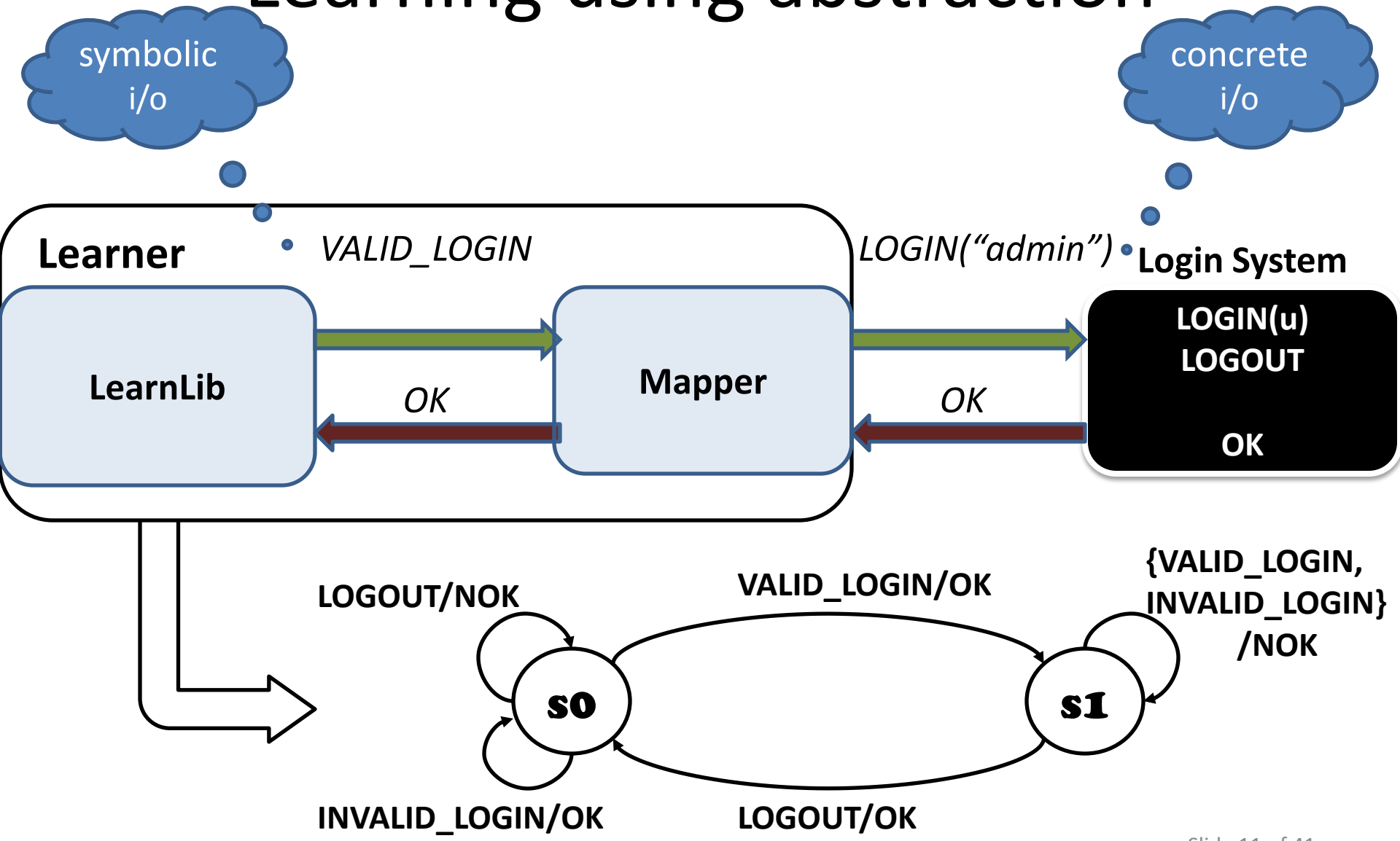


- input symbols for each possible parameter value
- too many for the Learner to handle
  - causes the Learner to explode
  - and the computer

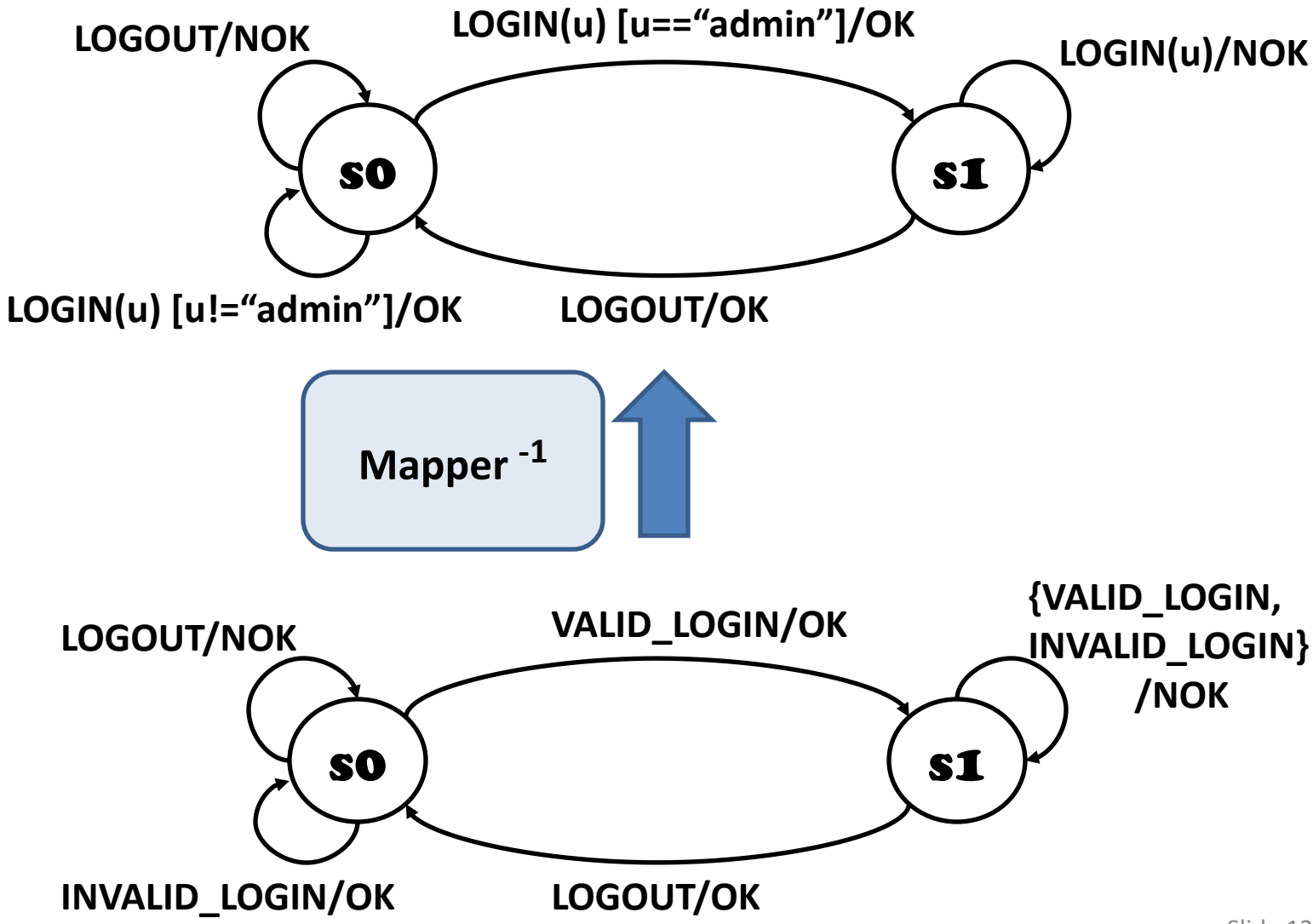
Classical learning



Learning using abstraction



Learning using abstraction



Learning using abstraction

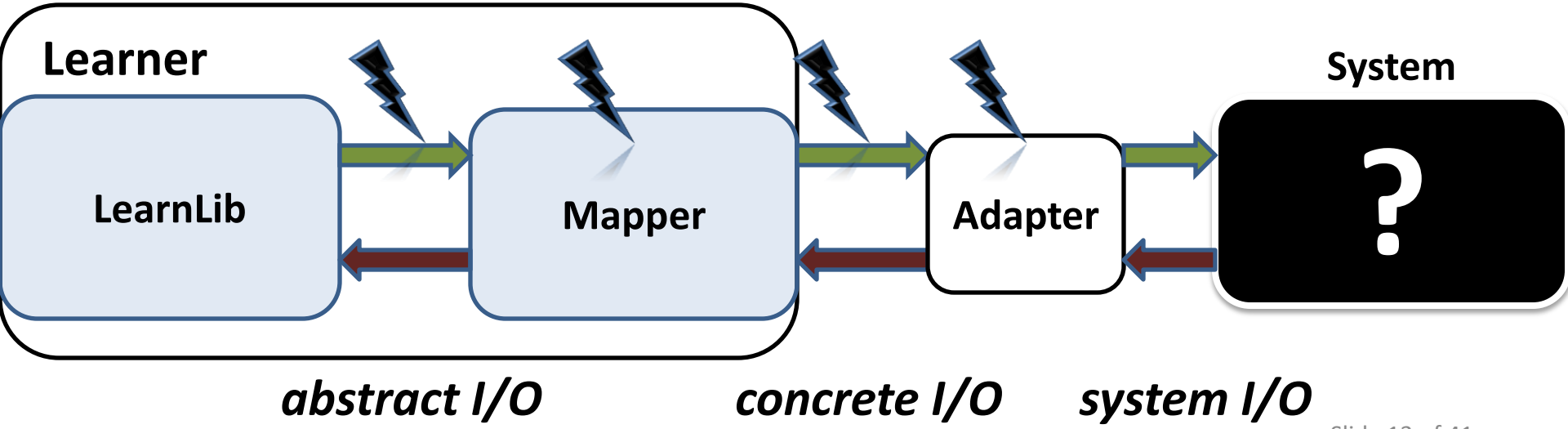
- build **abstract** and **concrete alphabets** (abstract/concrete io)

  {VALID_LOGIN, INVALID_LOGIN…} and {LOGIN(string),LOGOUT…}

- build **Mapper** component

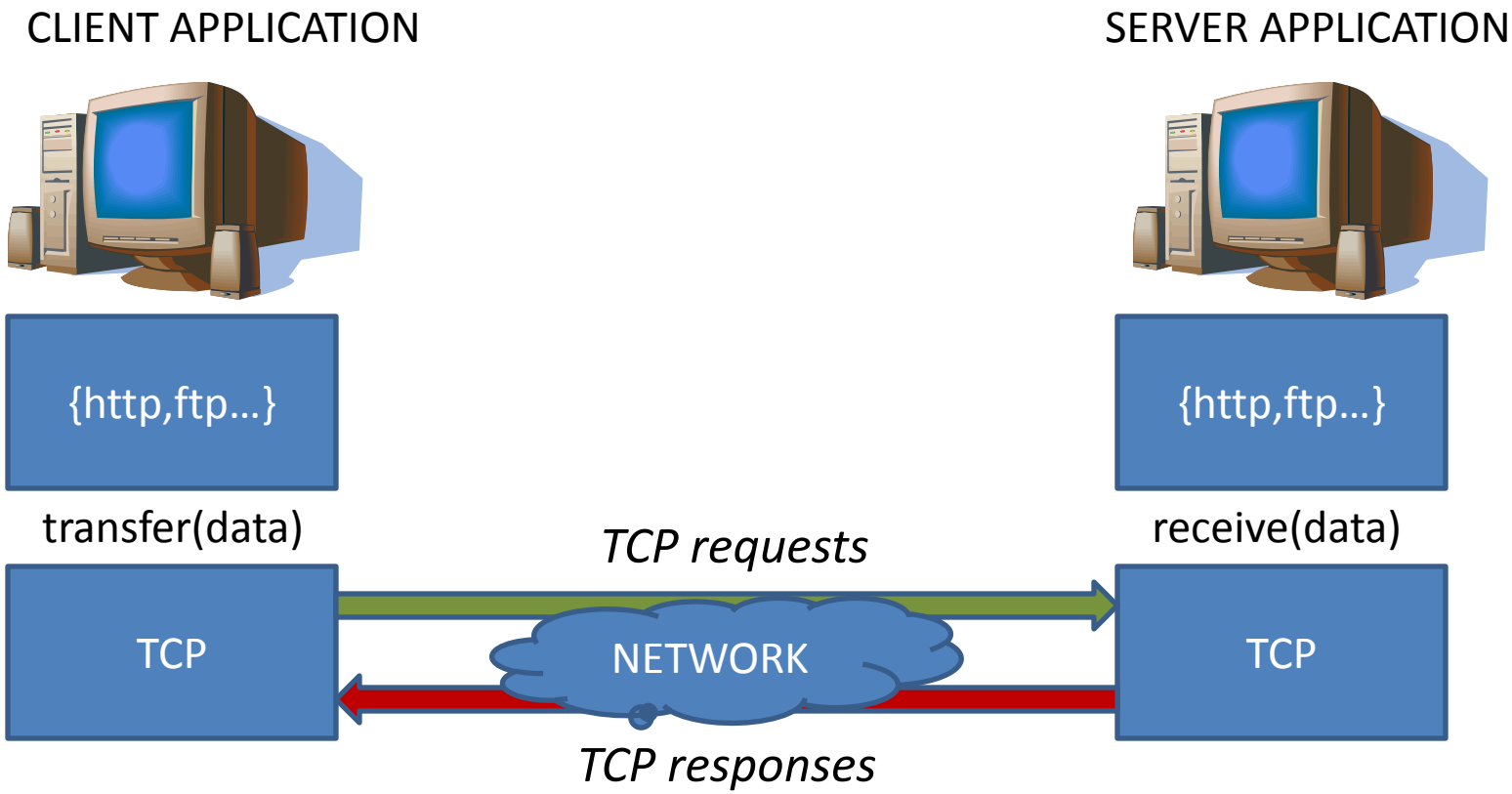
  $$\text{if input} = \text{VALID\_LOGIN then return LOGIN(“admin”)}$$

- build **Adapter** between Learner and System



The TCP Protocol

- connection based protocol for reliable data transfer
- connection = (clientIP, clientPort, serverIP, serverPort)

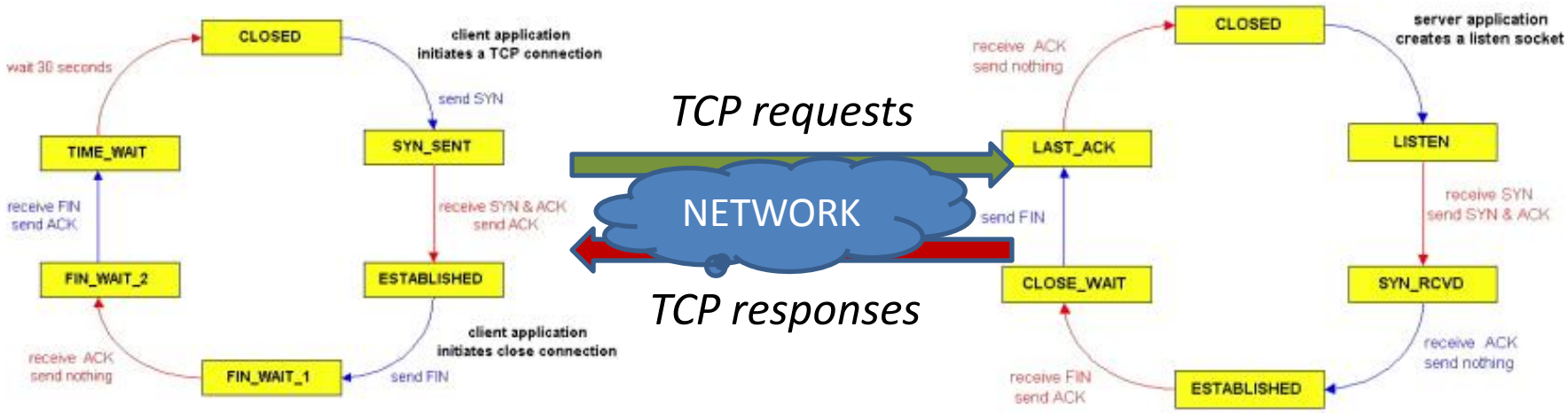


Learning the TCP Protocol

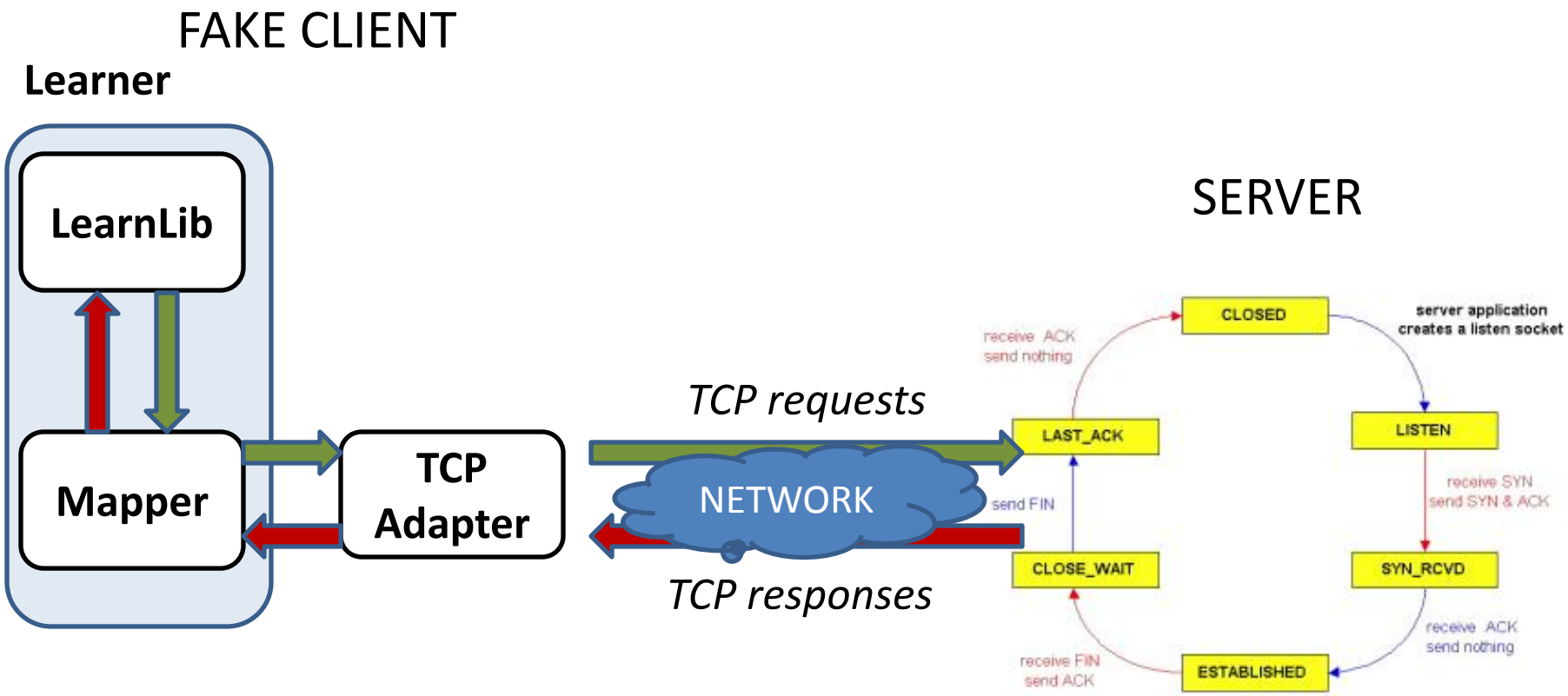
- both the client and the server are state machines

CLIENT

SERVER



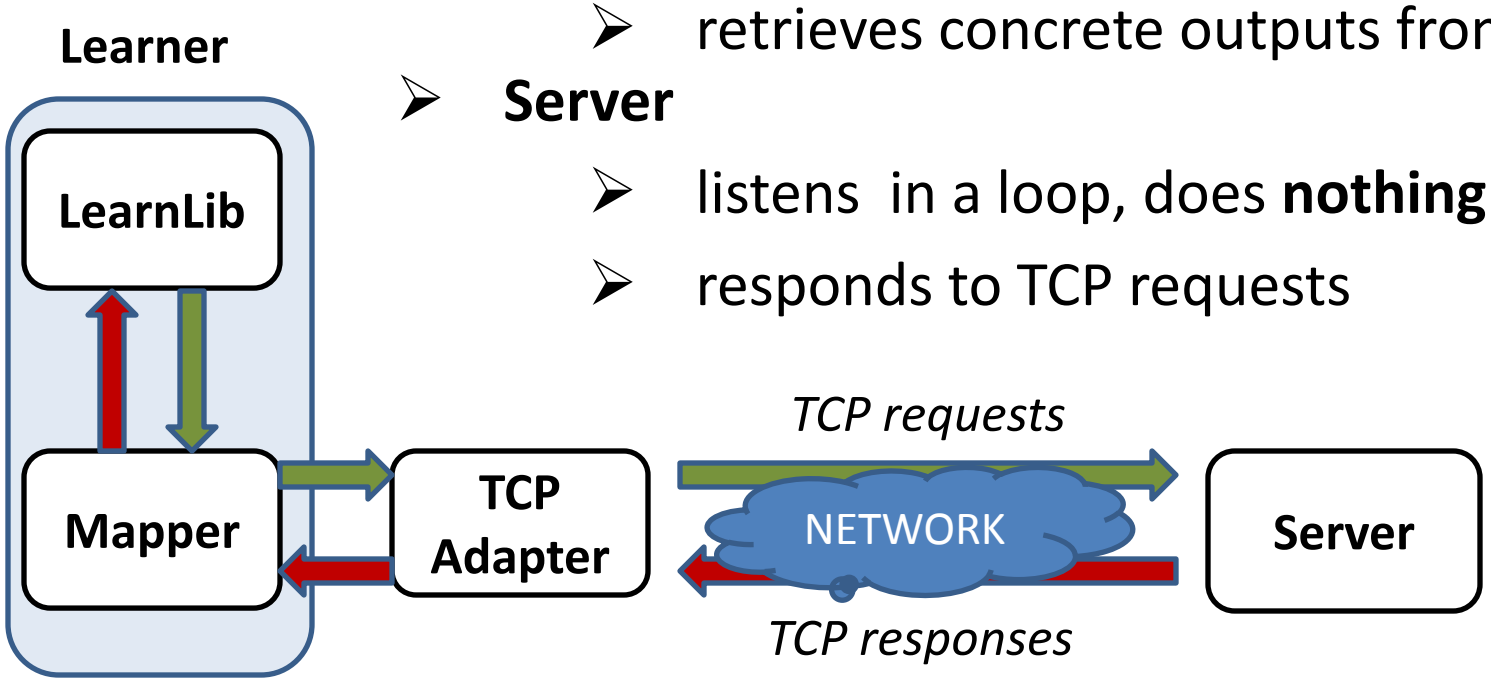
The learning setup



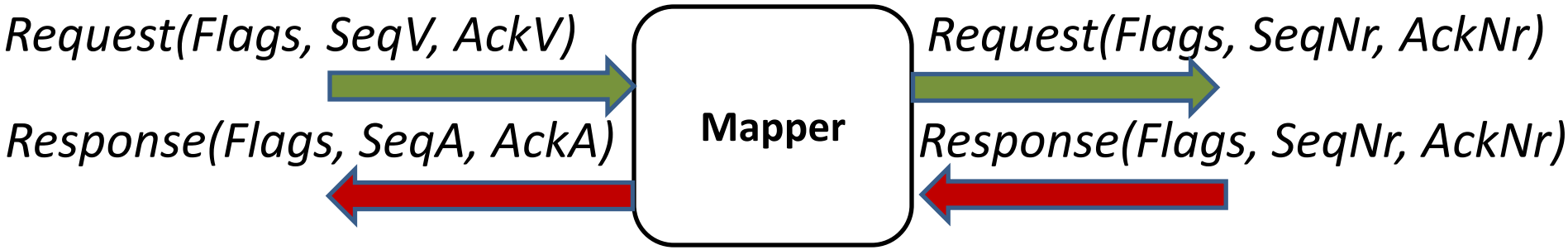
The learning setup

- **Learner** = **LearnLib** + **Java Mapper**
- **TCP Adapter** = **Python** tool built on **Scapy, Pcapy, Impacket**

  - **Learner**
    - sends concrete inputs to **Adapter**
    - retrieves concrete outputs from **Adapter**
  - **Server**
    - listens in a loop, does **nothing** else
    - responds to TCP requests



The alphabet



Abstract Parameters	Values
SeqV, AckV	*{V,INV}*
SeqA, AckA	*{CLSN, SVSN,…}*
Flags	*{SYN, SYN+ACK, ACK, FIN, FIN+ACK, RST, RST+ACK}*

Concrete Parameters	Values
SeqNr	$[0 , 2^{32}-1]$
AckNr	$[0 , 2^{32}-1]$

The mapper

```
Request(Flags, SeqV, AckV):
    if SeqV = V:
        SeqNr := clientSeqNr
    else:
        SeqNr := rand other than clientSeqNr
    if AckV = V:
        AckNr := serverSeqNr +1
    else:
        AckNr := rand other than serverSeqNr

    update clientSeqNr

    return Request(Flags, SeqNr, AckNr)
```

abstract to concrete

update

Request(Flags, SeqV, AckV)

Request(Flags, SeqNr, AckNr):

The mapper

Response(**Flags**, **SeqNr**, **AckNr**):
    SeqA :=getAbstractValue(**SeqNr**)
    AckA :=getAbstractValue(**AckNr**)

    update clientSeqNr, serverSeqNr

    return Response(**Flags**, **SeqA**, **AckA**)

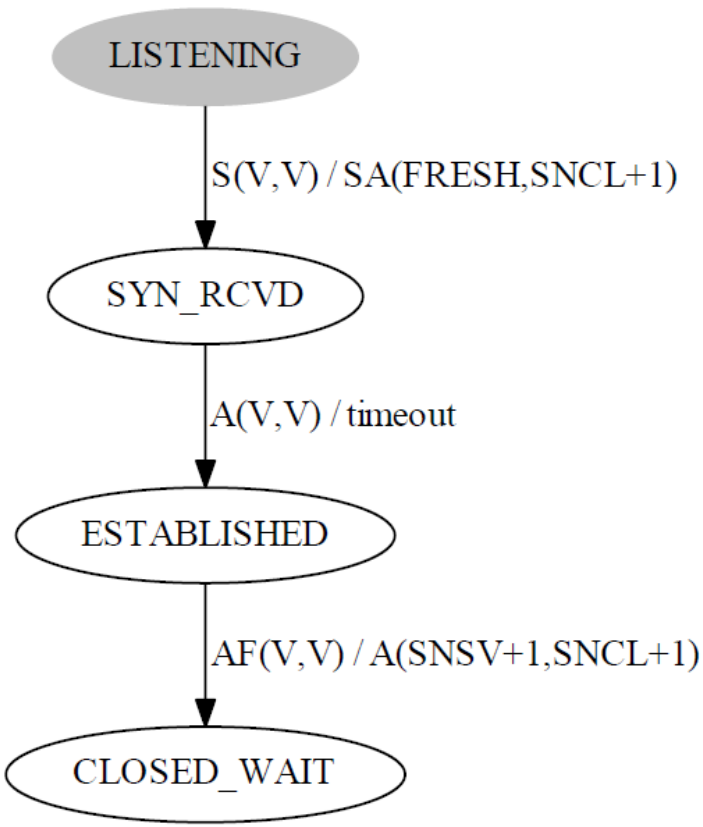
getAbstractValue(**RegNr**):
    matches **RegNr** to clientSeqNr, serverSeqNr,
        last **AckNr./SeqNr.** sent/received
    results in: CLSN, SVSN, LSS, LAS, ZERO…

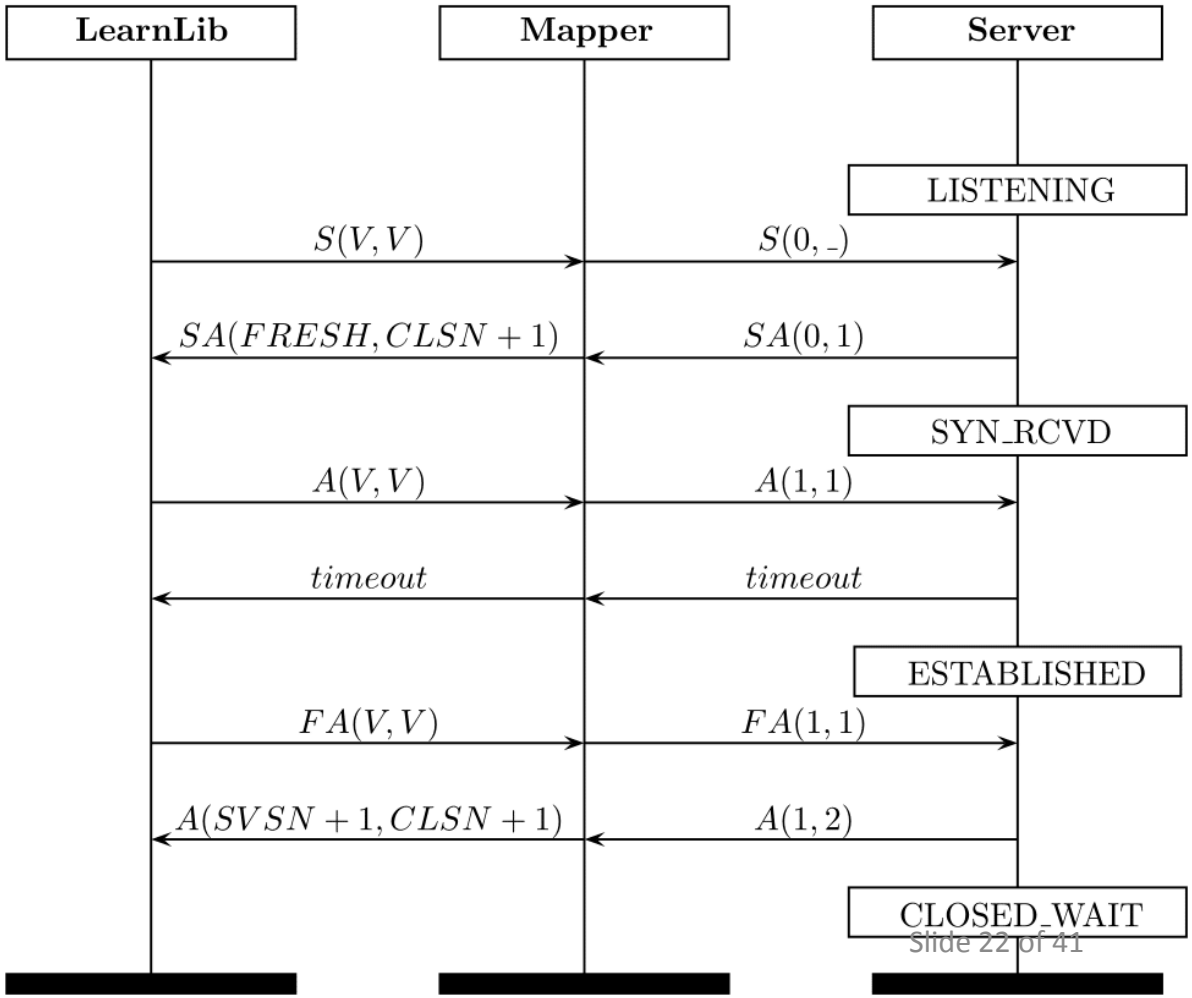
concrete to abstract

update

Example trace

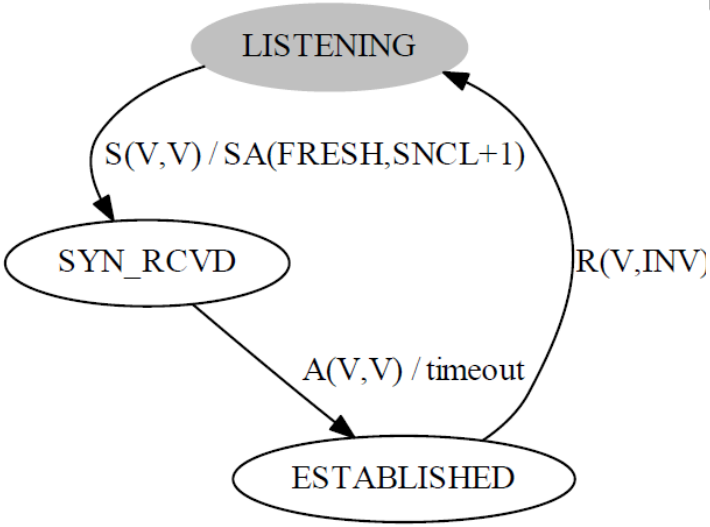
TCP Handshake plus closing:

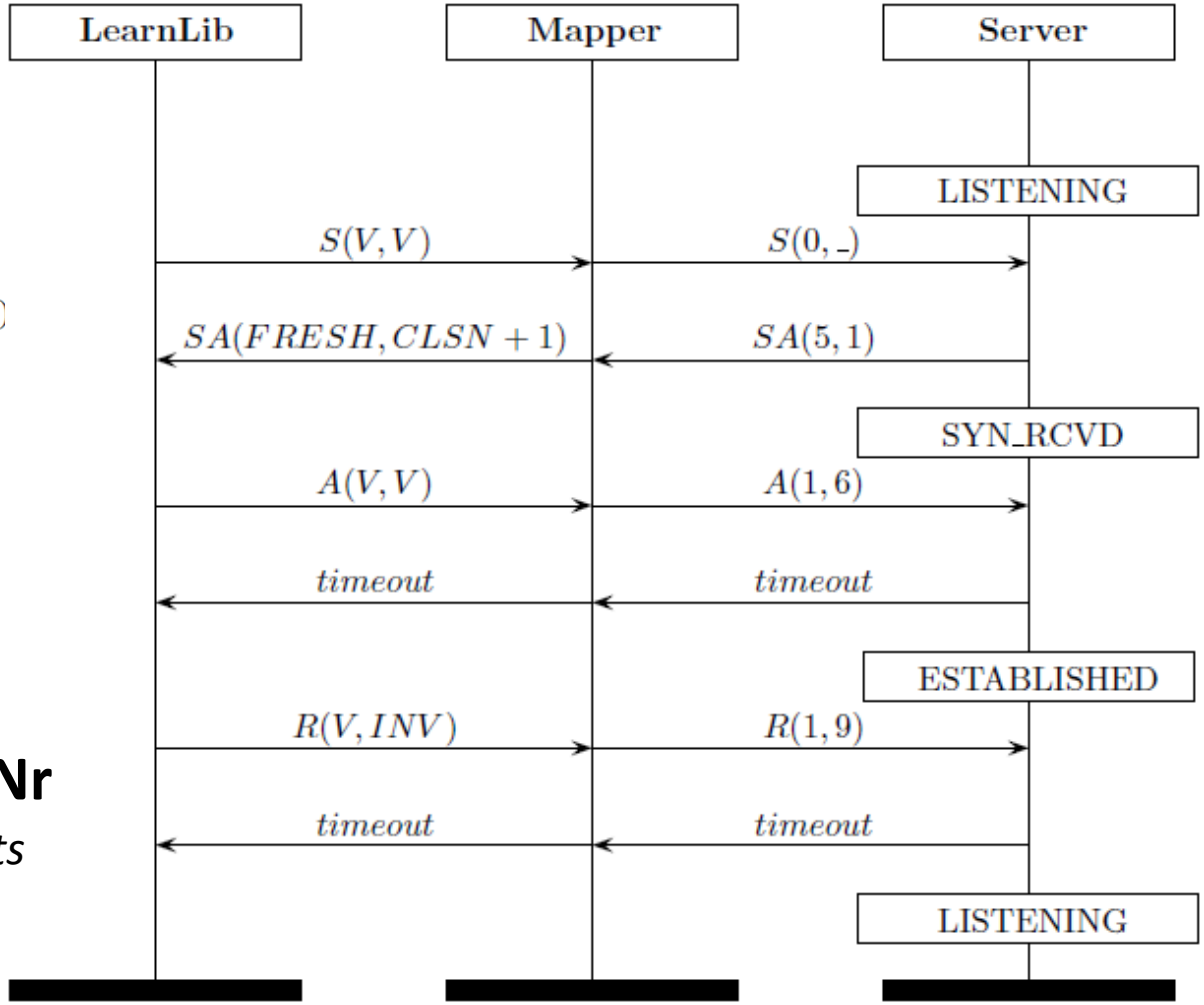




Example trace(2)

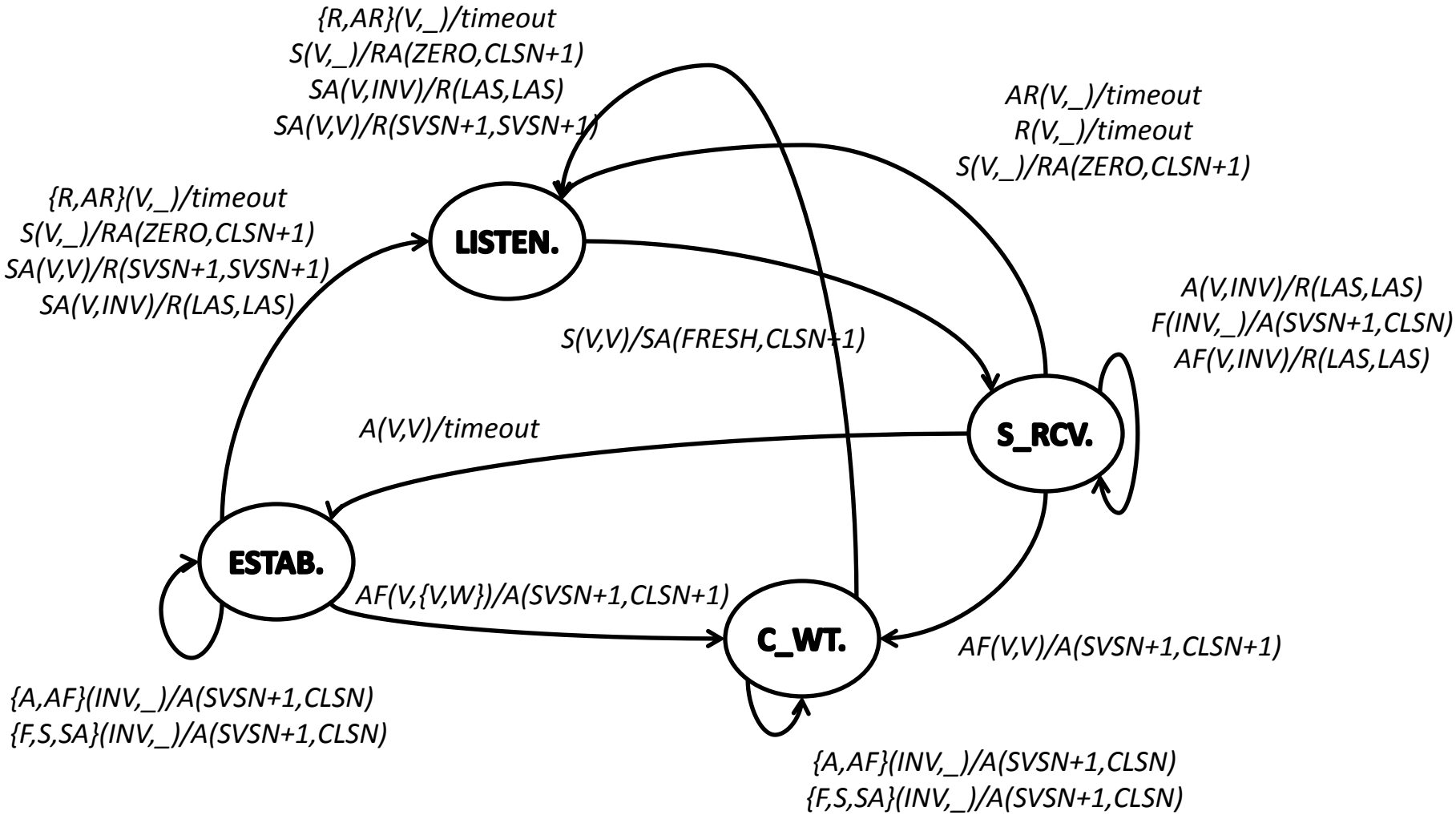
Resetting a connection:





- send *RST* with **valid SeqNr**
  - *alternative: change ports*

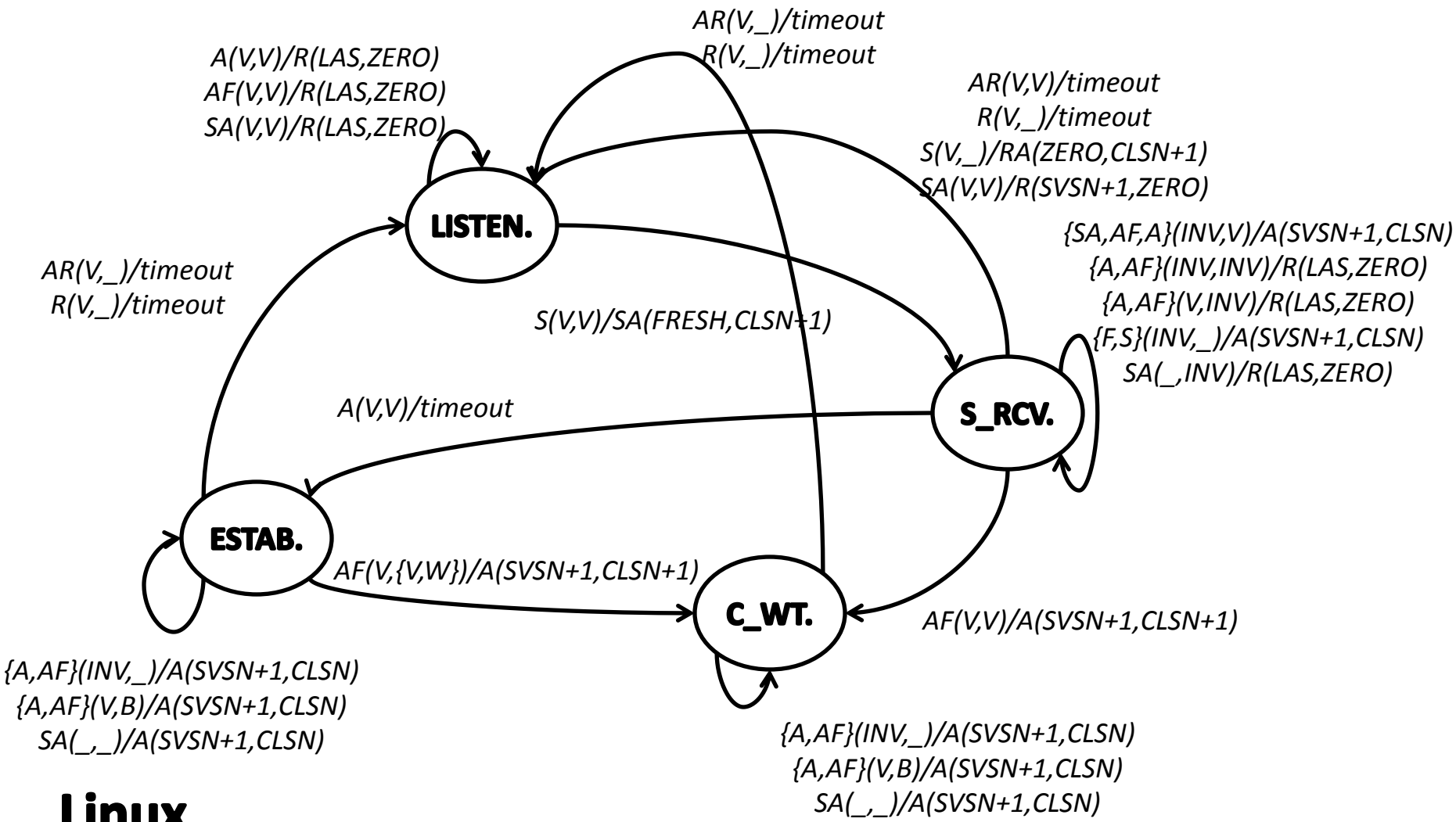
Learning results



Windows 8

Learning results

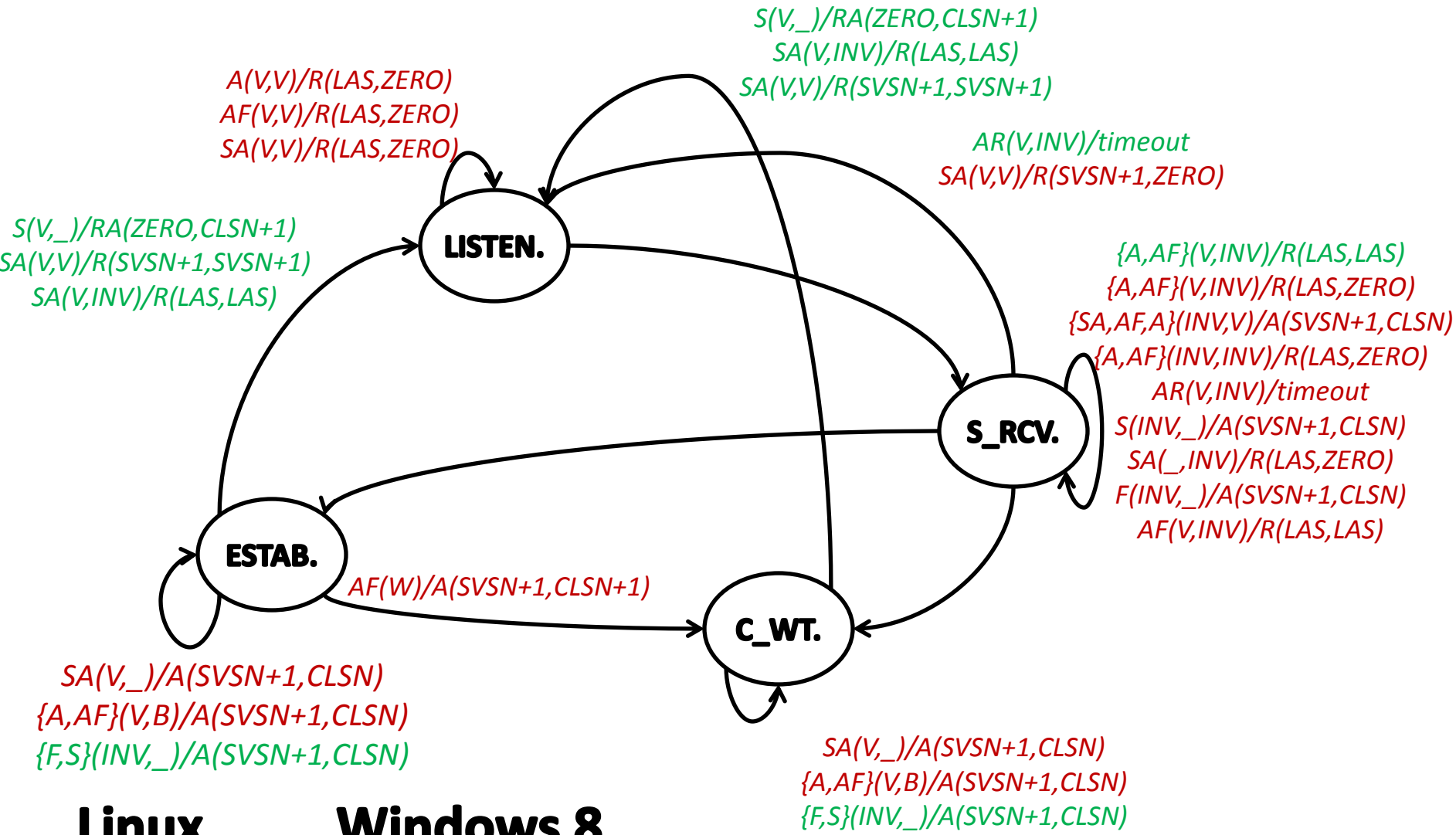
[INV]=[B:W:A]



Linux

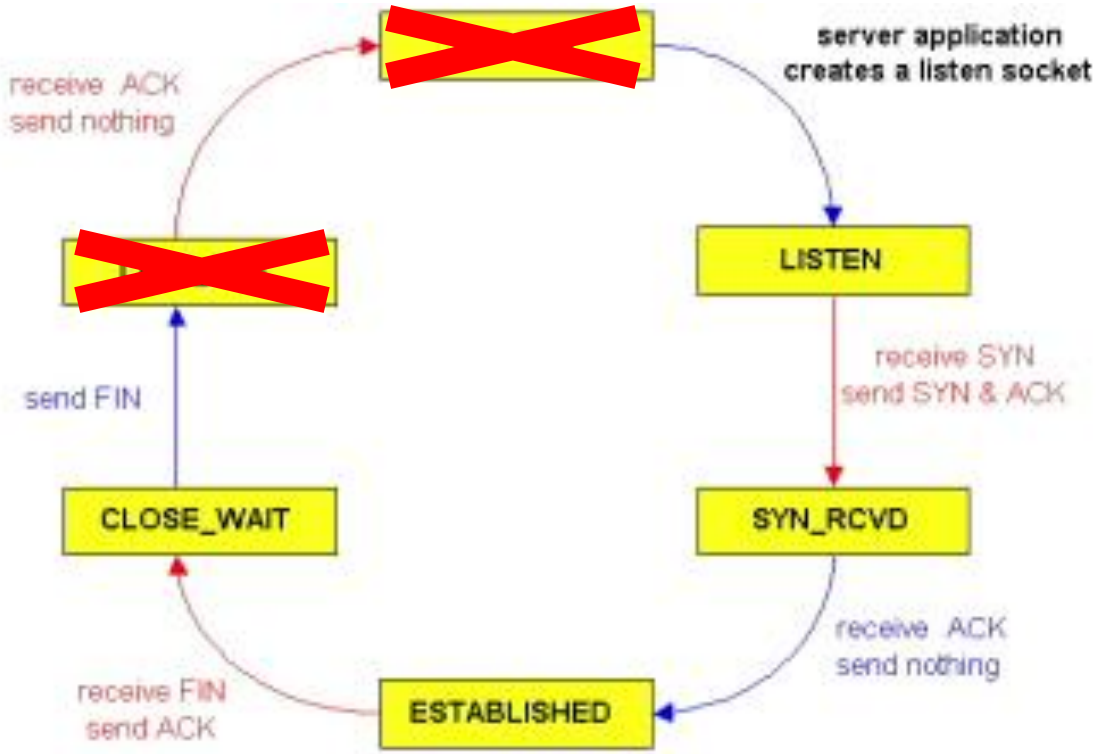
Results comparison

[INV]=[B:W:A]



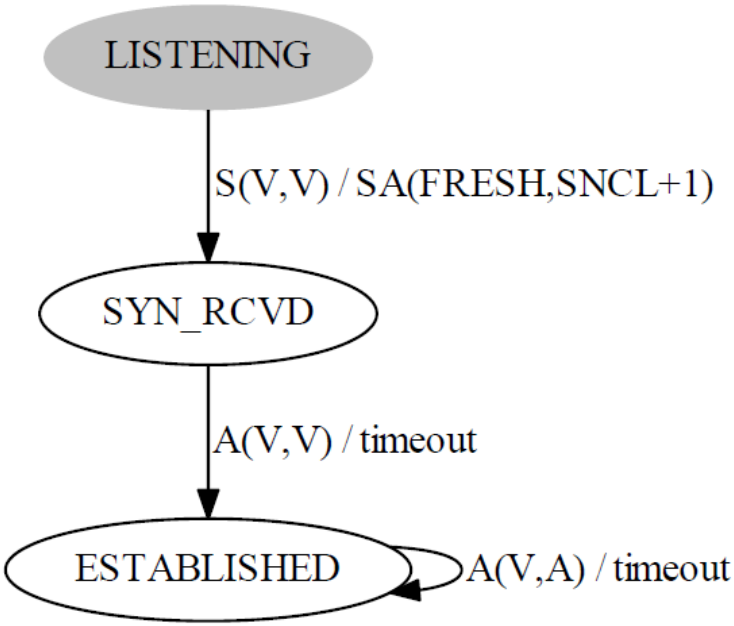
Linux **Windows 8**

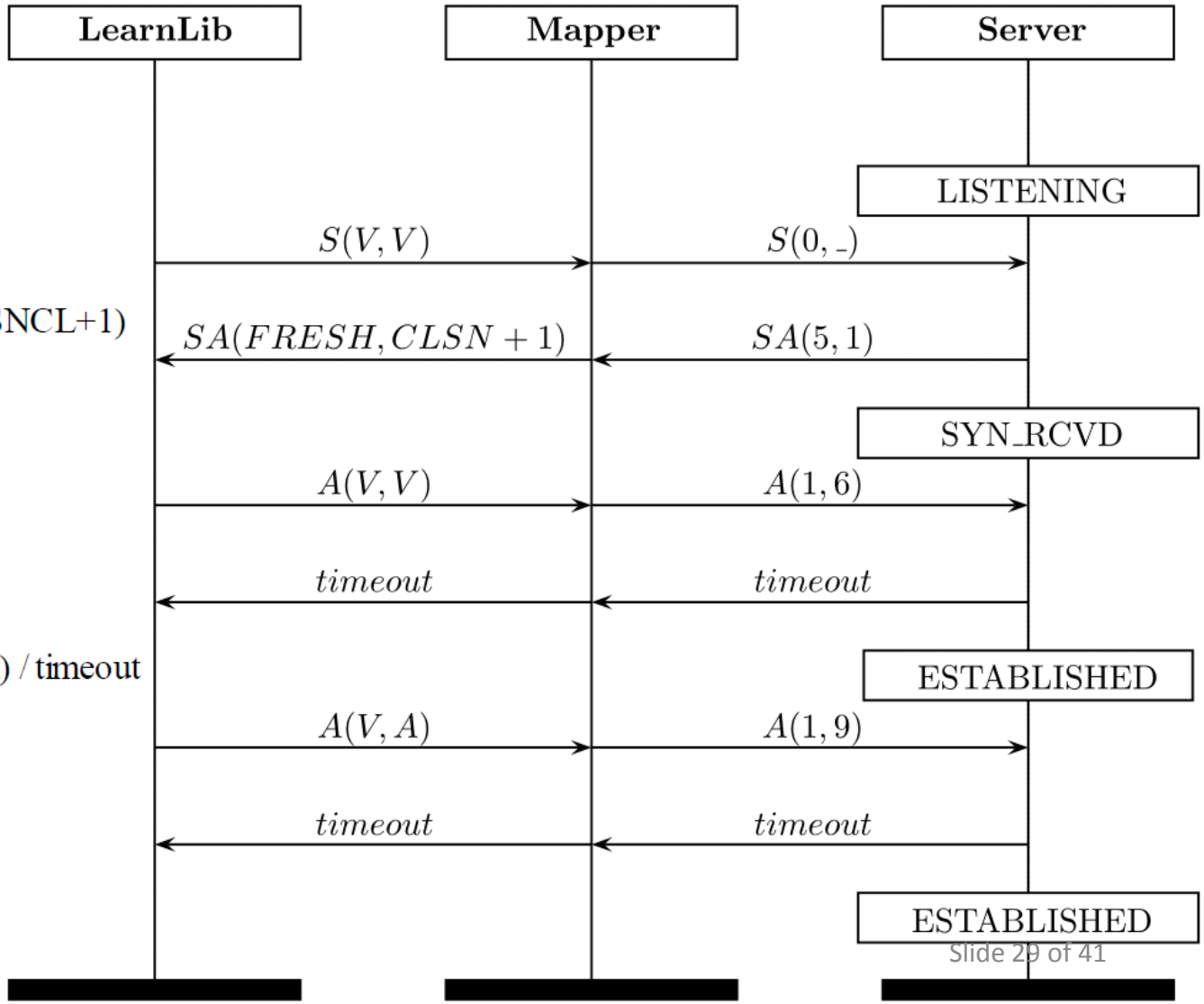
Learning results



Non-compliance

RFC 793 non-compliance





Linux is not compliant

```
/* If the ack includes data we haven't sent yet, discard
 * this segment (RFC793 Section 3.9).
 */
if (after(ack, tp->snd_nxt))
   goto invalid_ack;
…
invalid_ack:
         SOCK_DEBUG(…);
         return -1;
```

http://lxr.free-electrons.com/source/net/ipv4/tcp_input.c#L3374

Linux is not compliant

If the ACK acks something not yet sent (…) then **send an ACK,** drop the segment, and return.

(RFC 793, p. 72)

```
/* If the ack includes data we haven't sent yet, discard
 * this segment (RFC793 Section 3.9).
 */

  …
```

http://lxr.free-electrons.com/source/net/ipv4/tcp_input.c#L3374

Implications

- non-compliance detected
  - specification **not met**, suggested change
- easy comparison between implementations
  - easy OS-fingerprinting
- same approach, same framework to learn other protocols
  - FTP, SCTP…

Future work

- **learn** more TCP
  - add more parameters (data, length, window size)
  - complicate server (echo server)
  - test other implementations
- **learn** higher level protocols

Future work

- **learn** more TCP
  - add more parameters (data, length, window size)
  - complicate server (echo server)
  - test other implementations
- **learn** higher level protocols
  - then the world

Unfortunately…

- writing mappers manually **IS** difficult:
  - implies forming abstractions
    - which necessitates reading the specification
  - large number of experiments
    - lengthy (sometimes endless) experiments
  - significant manual labor
    - only rewarding once you learn something

Solution

- Tomte – learning tool
  - can learn a subset of EFSMs **automatically**
    - no mapper required
  - systems learned:
    - login system, SIP protocol, biometric passport
    - more recently: stack and queue systems

Future work

- Tomte – learning tool
  - can learn a **subset** of EFSMs

Tomte	TCP requirements
outputs derived from previous inputs	outputs can be fresh values

Future work

- Tomte – learning tool
  - can learn a **subset** of EFSMs

Tomte	TCP requirements
outputs derived from previous inputs	outputs can be fresh values
outputs derived by assignment	outputs derived by simple arithmetic (assignment, increment…)

Future work

- Tomte – learning tool
  - can learn a **subset** of EFSMs

Tomte	TCP requirements
outputs derived from previous inputs	outputs can be fresh values
outputs derived by assignment	outputs derived by simple arithmetic (assignment, increment…)
time agnostic	time matters

Conclusions

- built a setup for learning a fragment of TCP
  - 4 states, flags, seq and ack numbers, no data
- obtained models which showed functionality
  - and discrepancy Windows/Linux/RFC 793
- set new automation goals
  - so we cut out the middle-man