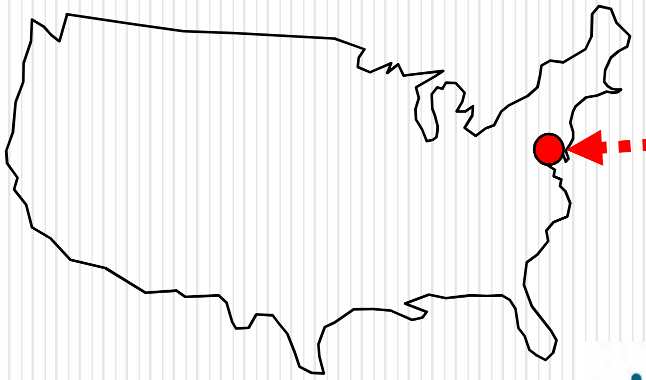
A Model for Detecting Transport Layer Data Reneging

Nasif Ekiz, Paul D. Amer

Protocol Engineering Laboratory

Computer and Information Sciences,

University of Delaware



supported by CISCO

PFLDNeT 2010

OUTLINE

1. What is data reneging?
2. Why study reneging?
3. A model to detect reneging
4. Model verification
5. Work in progress

OUTLINE

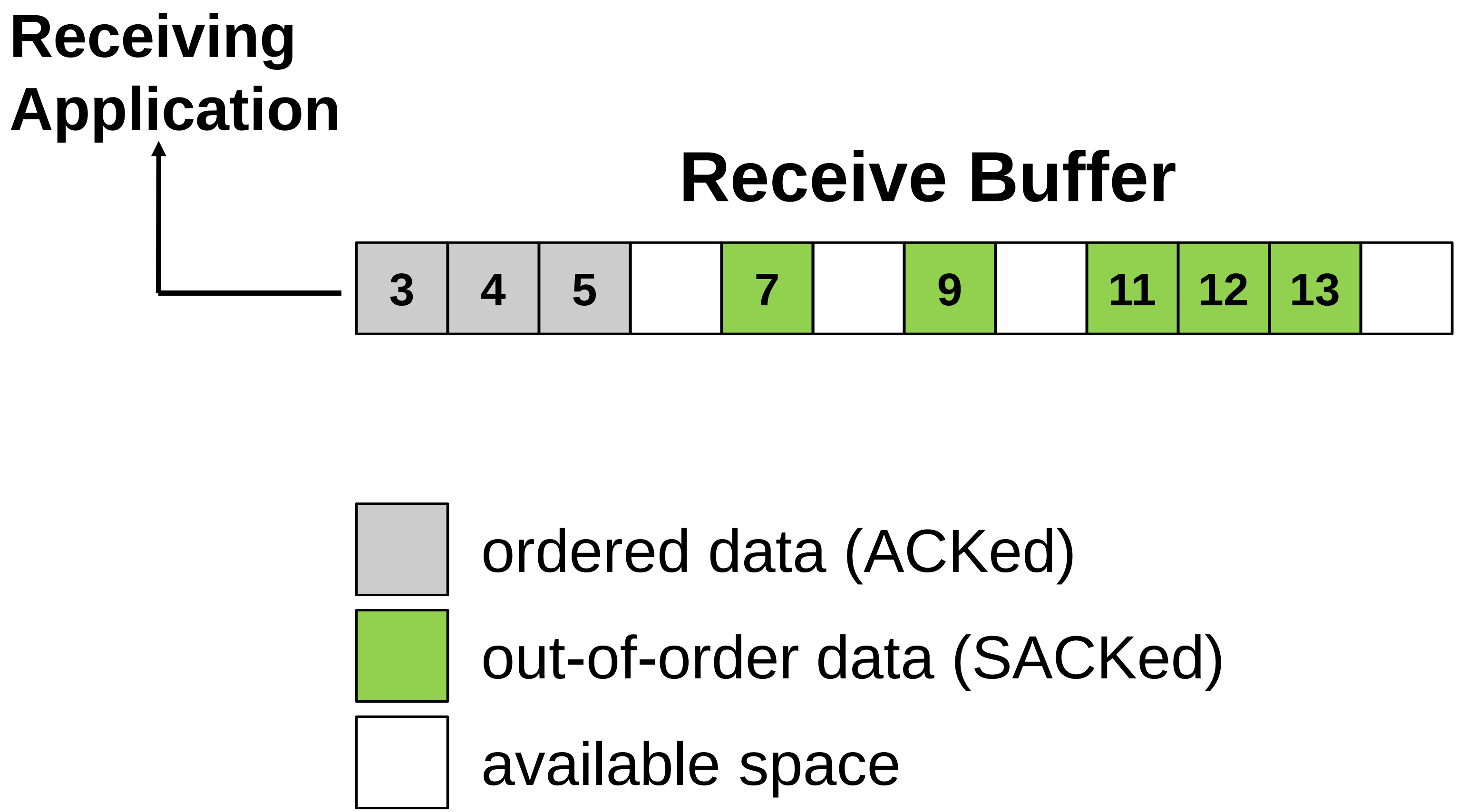
1. What is data reneging?
2. Why study reneging?
3. A model to detect reneging
4. Model verification
5. Work in progress

Types of acknowledgements

- For ordered data - cumulative ACK *n*
  - bytes [*… to n-1*] (TCP) [RFC 793]
  - segments [*… to n*] (SCTP) [RFC 2960]

- For out-of-order data - selective ACK (SACK) *m-n*
  - bytes [*m* to *n-1*] (TCP) [RFC 2018]
  - segments [*m* to *n*] (SCTP) [RFC 2960]
  - ➢ Prevents unnecessary retransmissions during loss recovery
  - ➢ Improves throughput when multiple losses in same window

Receive buffer



Data reneging

- TCP is designed to tolerate reneging
  - [RFC 2018]: “The SACK option is *advisory*, in that, while it notifies the data sender that the data receiver has received the indicated segments, the data receiver is permitted to later *discard* data which have been reported in a SACK option.”
  - ➢ discarding SACKed data is “reneging”
  - ➢ TCP data sender retains copies of all SACKed data until ACKed

TCP and SCTP tolerate reneging

- We argue that tolerating reneging is wrong

1. Hypothesis: “data reneging rarely if ever occurs in practice”
2. Research demonstrates improved performance if SACKed data were not renegable
   - better utilization of send buffer

•Natarajan, Ekiz, Yilmaz, Amer, Iyengar, Stewart, “**Non-renegable selective acks (NR-SACKs) for SCTP**” Int'l Conf on Network Protocols (ICNP), Orlando, 10/08

   - improved throughput (SCTP only)

•Yilmaz, Ekiz, Natarajan, Amer, Leighton, Baker, Stewart, "**Throughput analysis of Non-Renegable Selective Acknowledgments (NR-SACKs) for SCTP**", Computer Communications. 2010

OUTLINE

1. What is data reneging?
2. Why study reneging?
3. A model to detect reneging
4. Model verification
5. Work in progress

Why study reneging?

- Let’s assume transport protocols are designed to NOT tolerate data reneging
  - optimal send buffer utilization
  - improved throughput (SCTP only)

- Changing current TCP and SCTP into non-reneging protocols is easy:
  - SACK semantics changed from advisory to permanent
  - If data receiver needs to renege, data receiver must first RESET the connection

Why study reneging?

- Suppose reneging occurs 1 in 100,000 TCP (or SCTP) flows
- Case A (current practice): reneging tolerated
  - 99,999 non-reneging connections underutilize send buffer (and for SCTP may achieve lower throughput)
  - 1 reneging connection continues (maybe?)
- Case B (proposed change): reneging not tolerated
  - 99,999 connections have equal or better send buffer utilization (and for SCTP throughput)
  - 1 reneging connection is RESET

Why study reneging?

- Data reneging has never been studied
  - Does data reneging happen or not?
  - If reneging happens, how often?

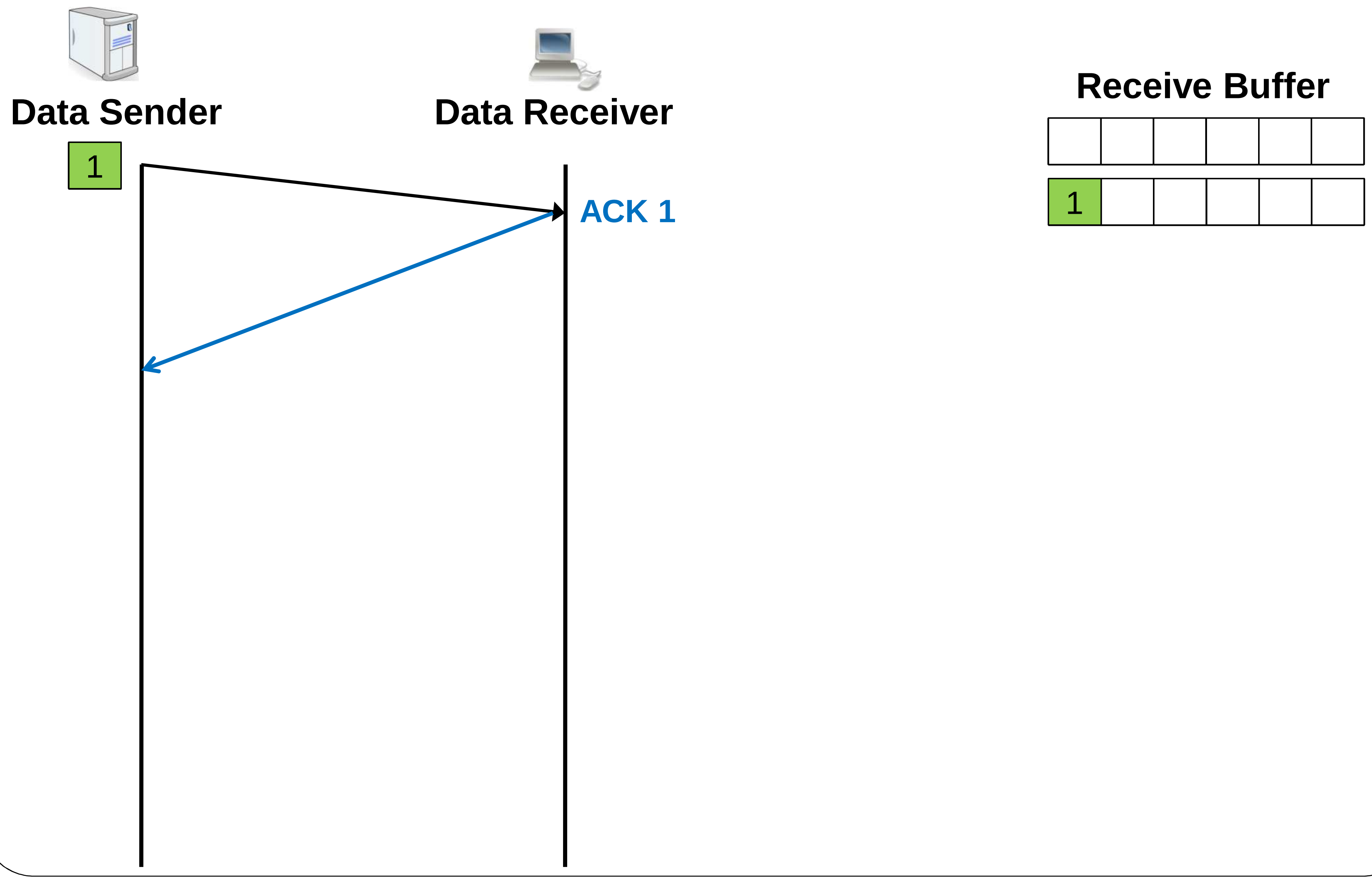
OUTLINE

1. What is data reneging?
2. Why study reneging?
3. A model to detect reneging
4. Model verification
5. Work in progress

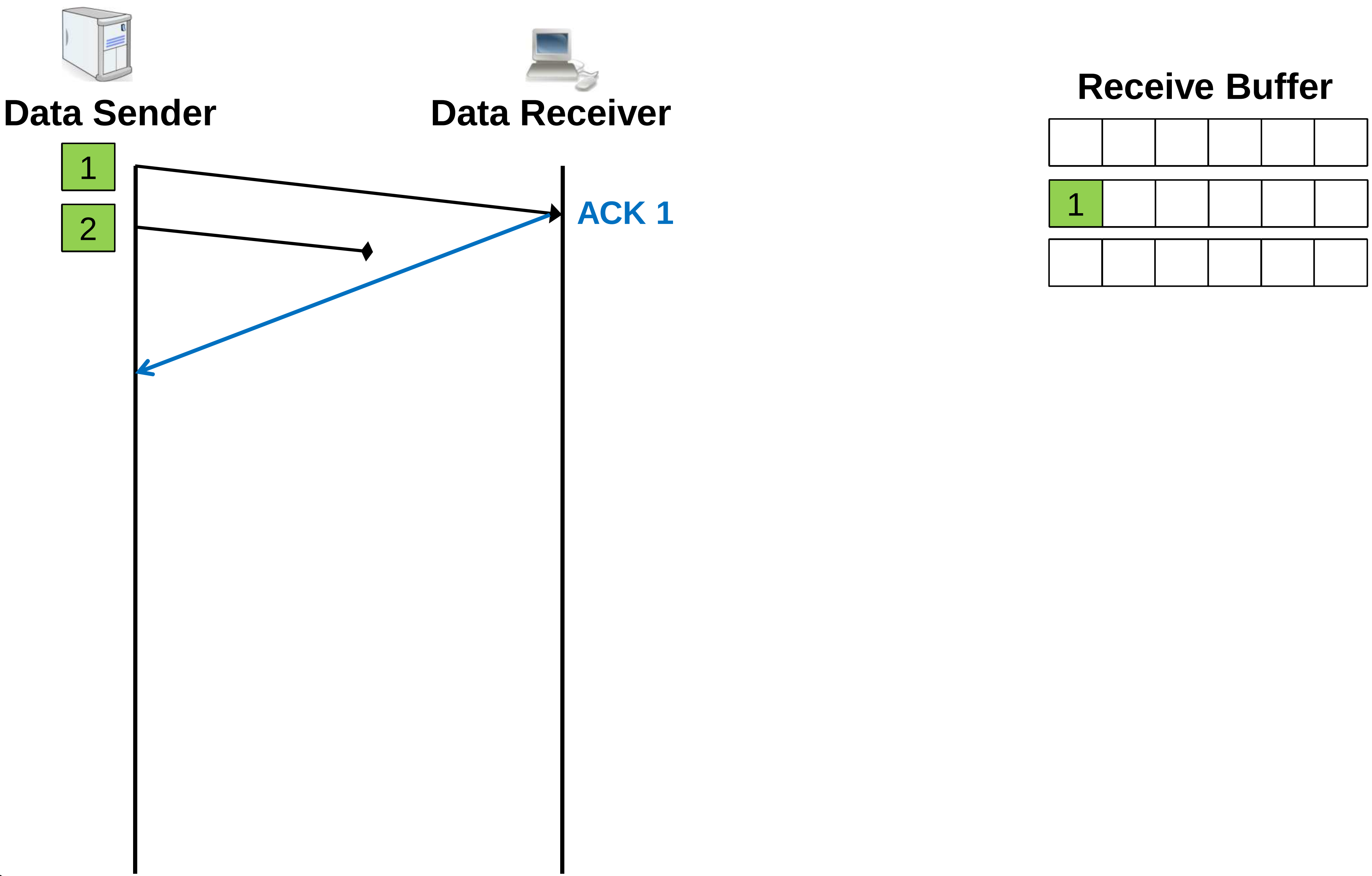
Detecting reneging at TCP data sender

- TCP has no mechanism to detect reneging
- To tolerate reneging, [RFC 2018] suggests the following retransmission policy
  - For each SACKed segment, "SACKed" flag is set
  - "SACKed" segments are not retransmitted until a timeout
  - At timeout, "SACKed" information is cleared

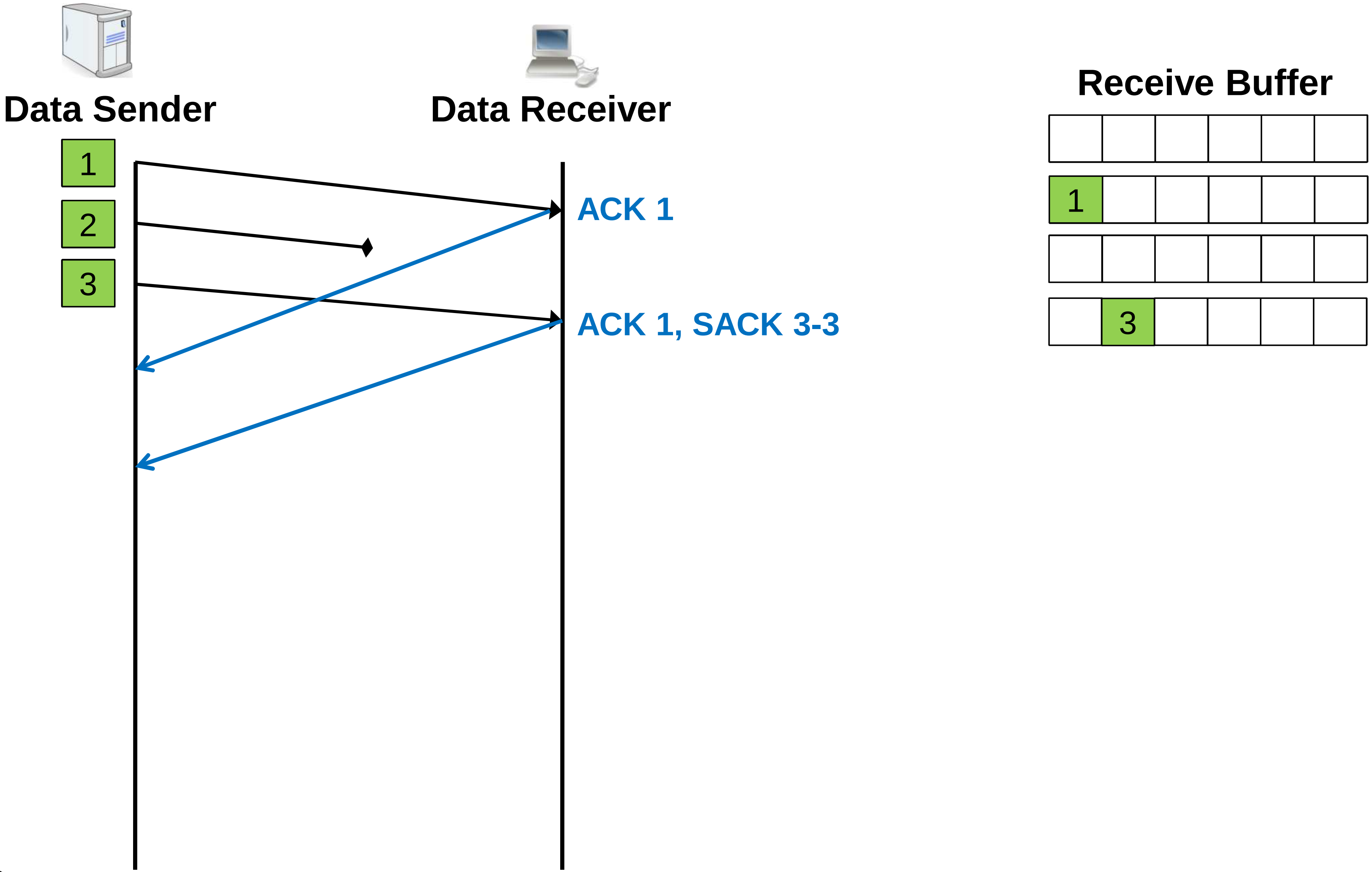
Detecting reneging at SCTP data sender



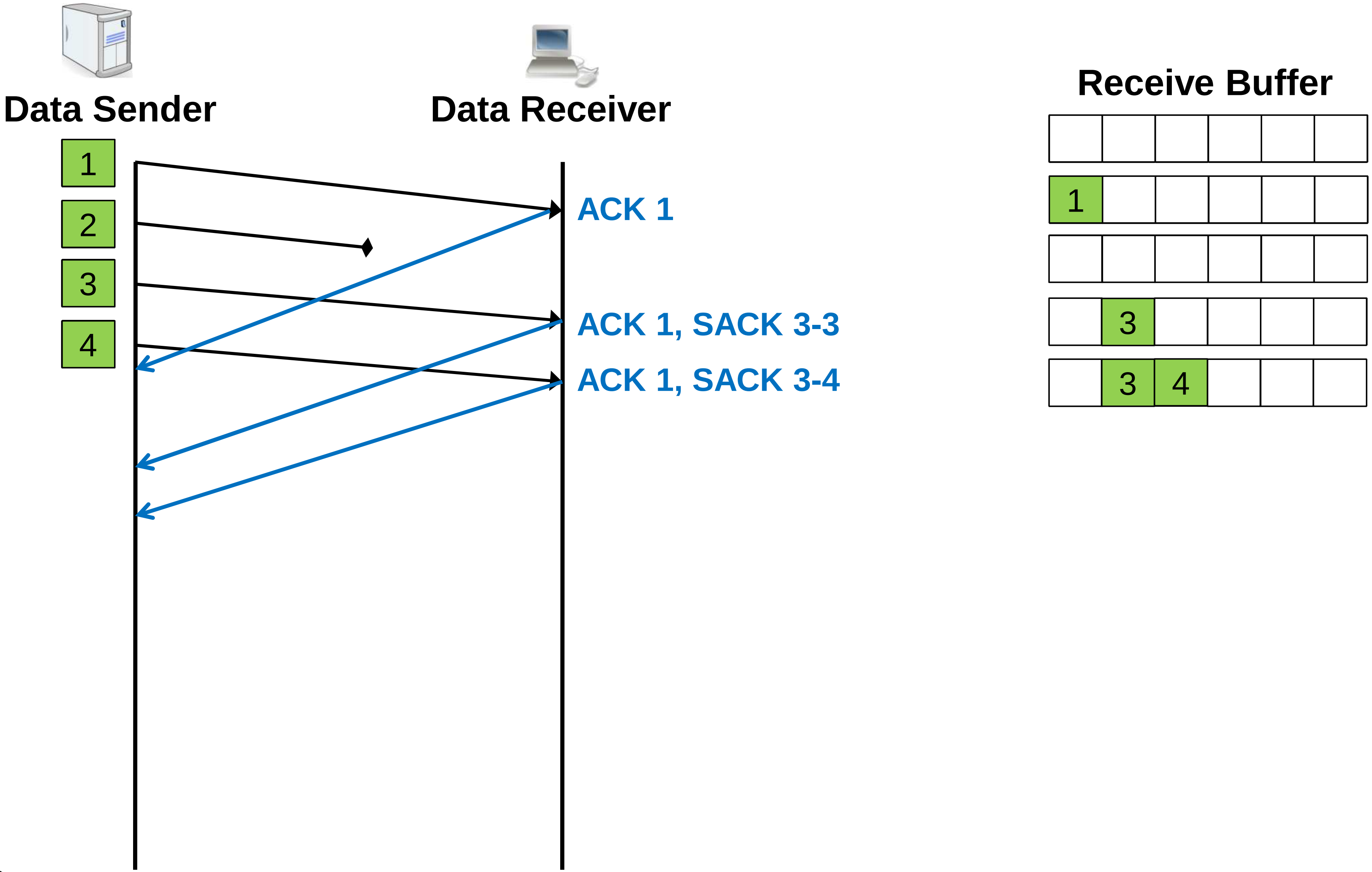
Detecting reneging at SCTP data sender



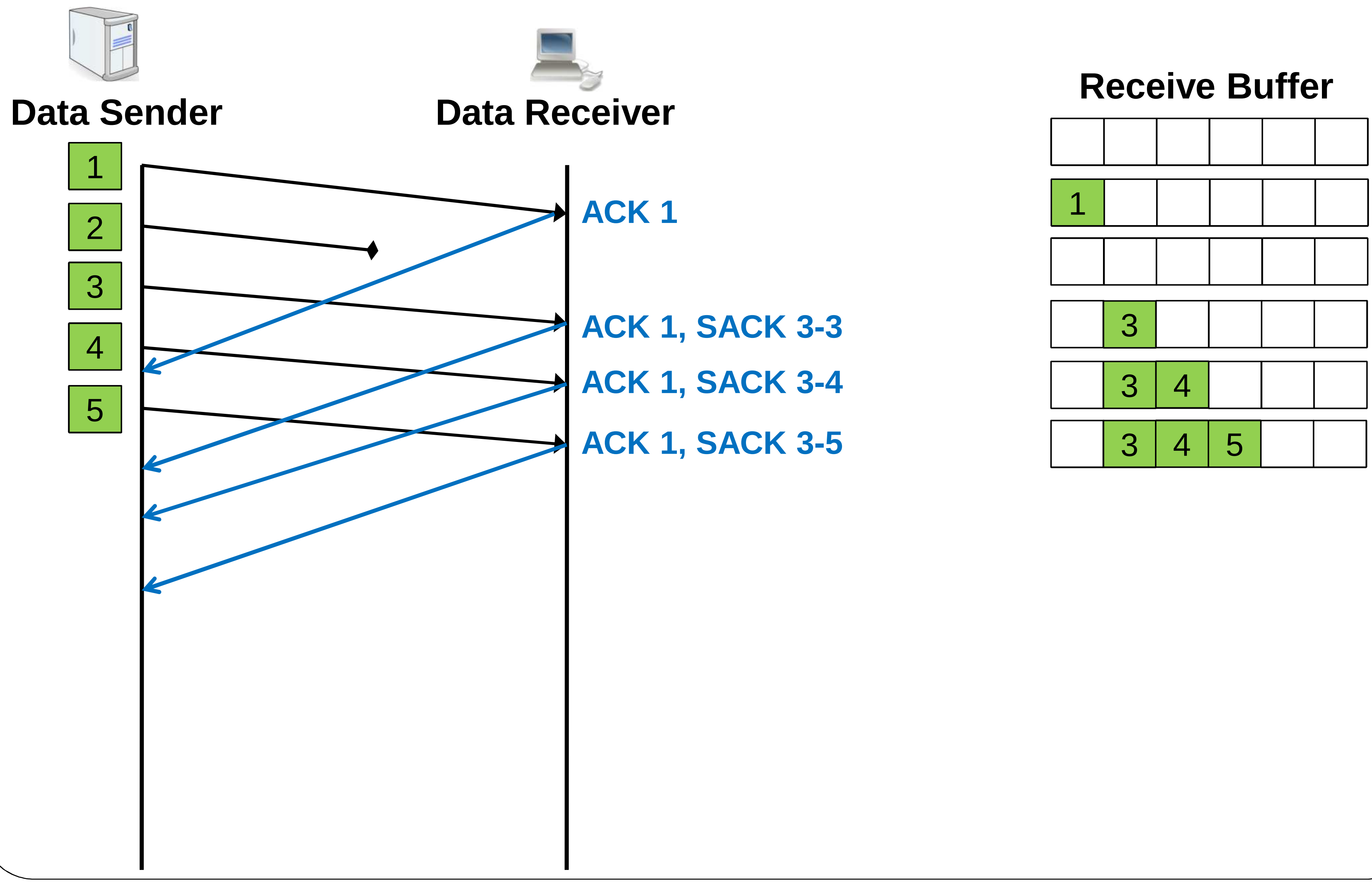
Detecting reneging at SCTP data sender



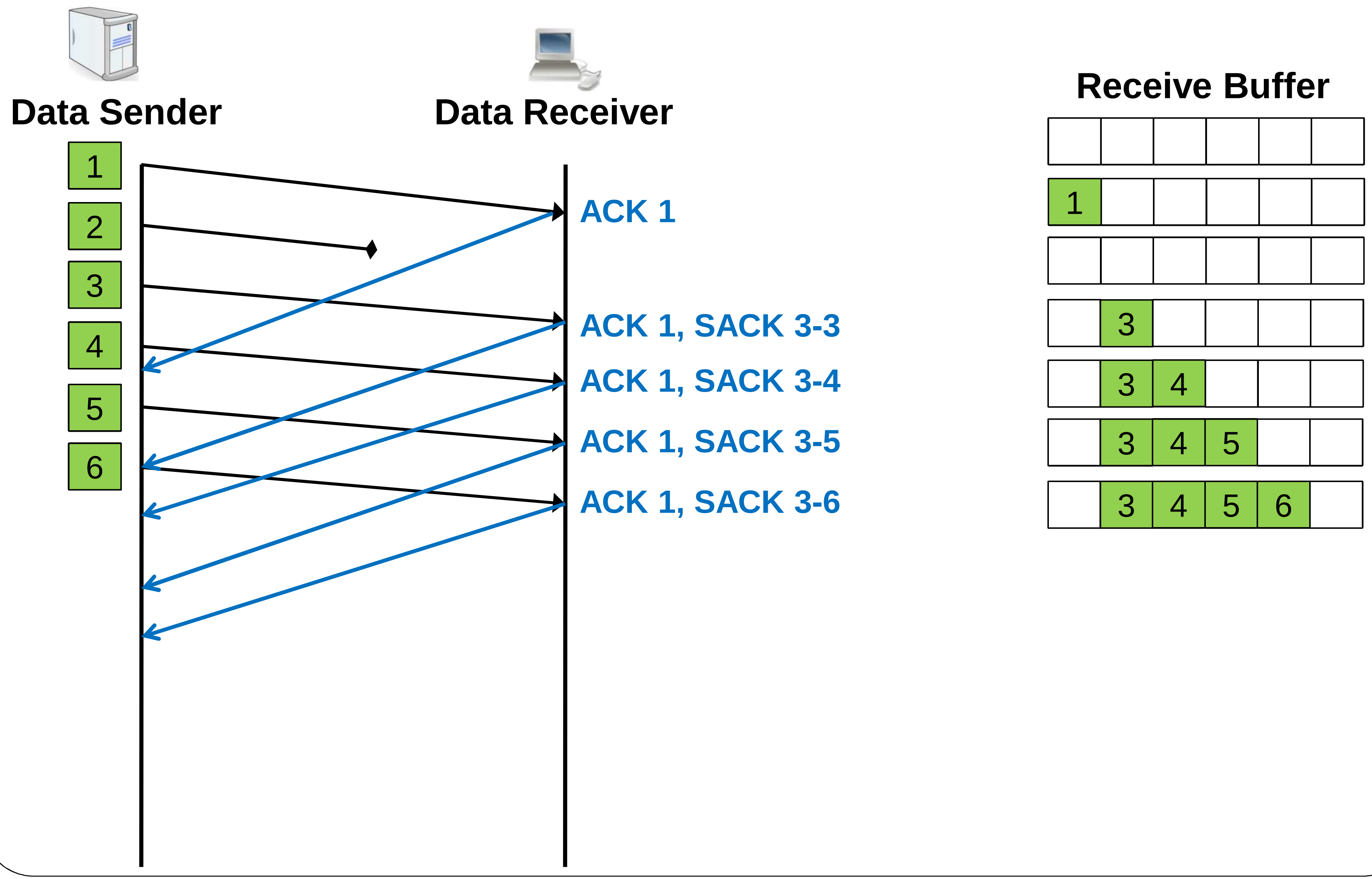
Detecting reneging at SCTP data sender



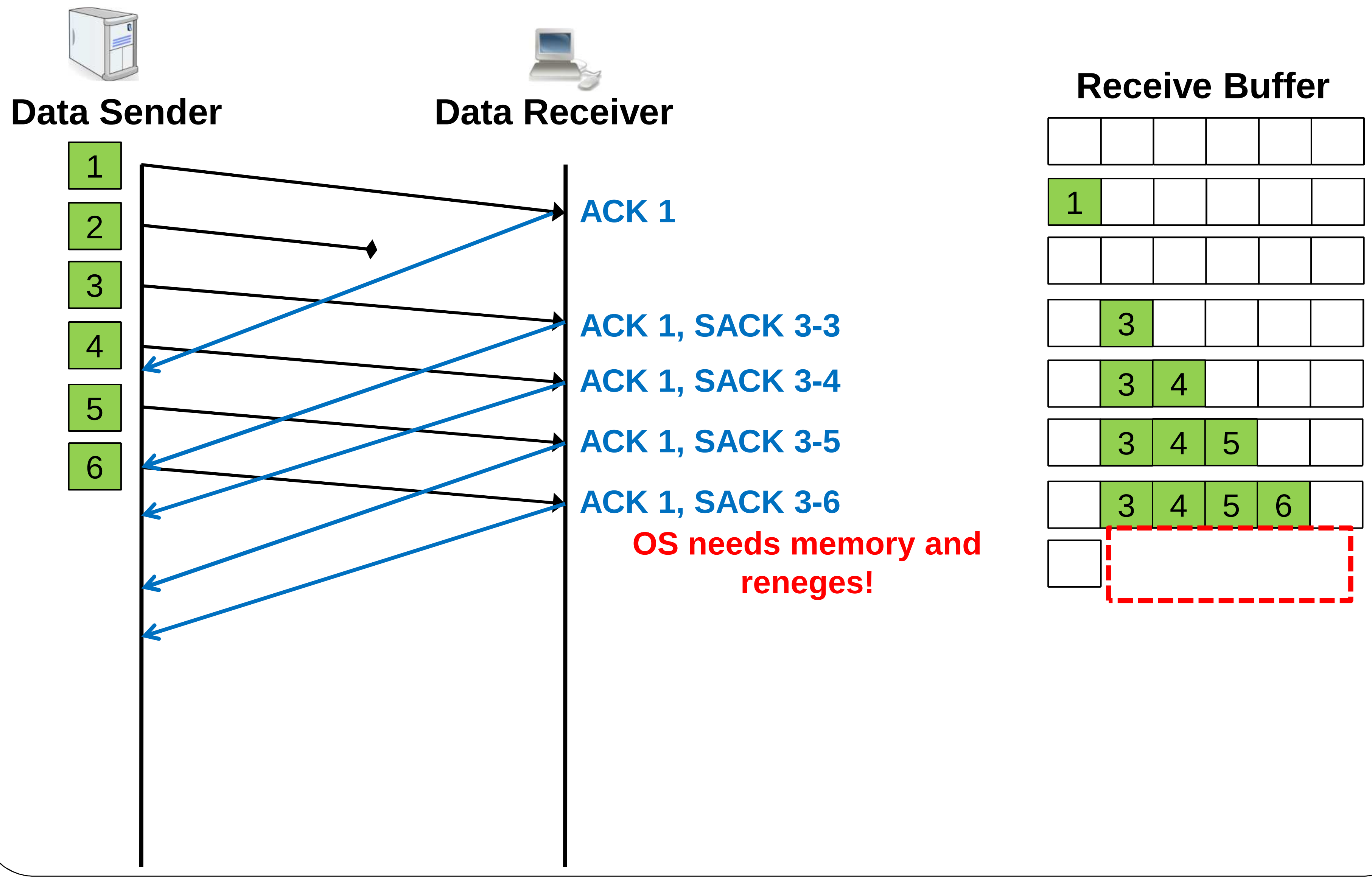
Detecting reneging at SCTP data sender



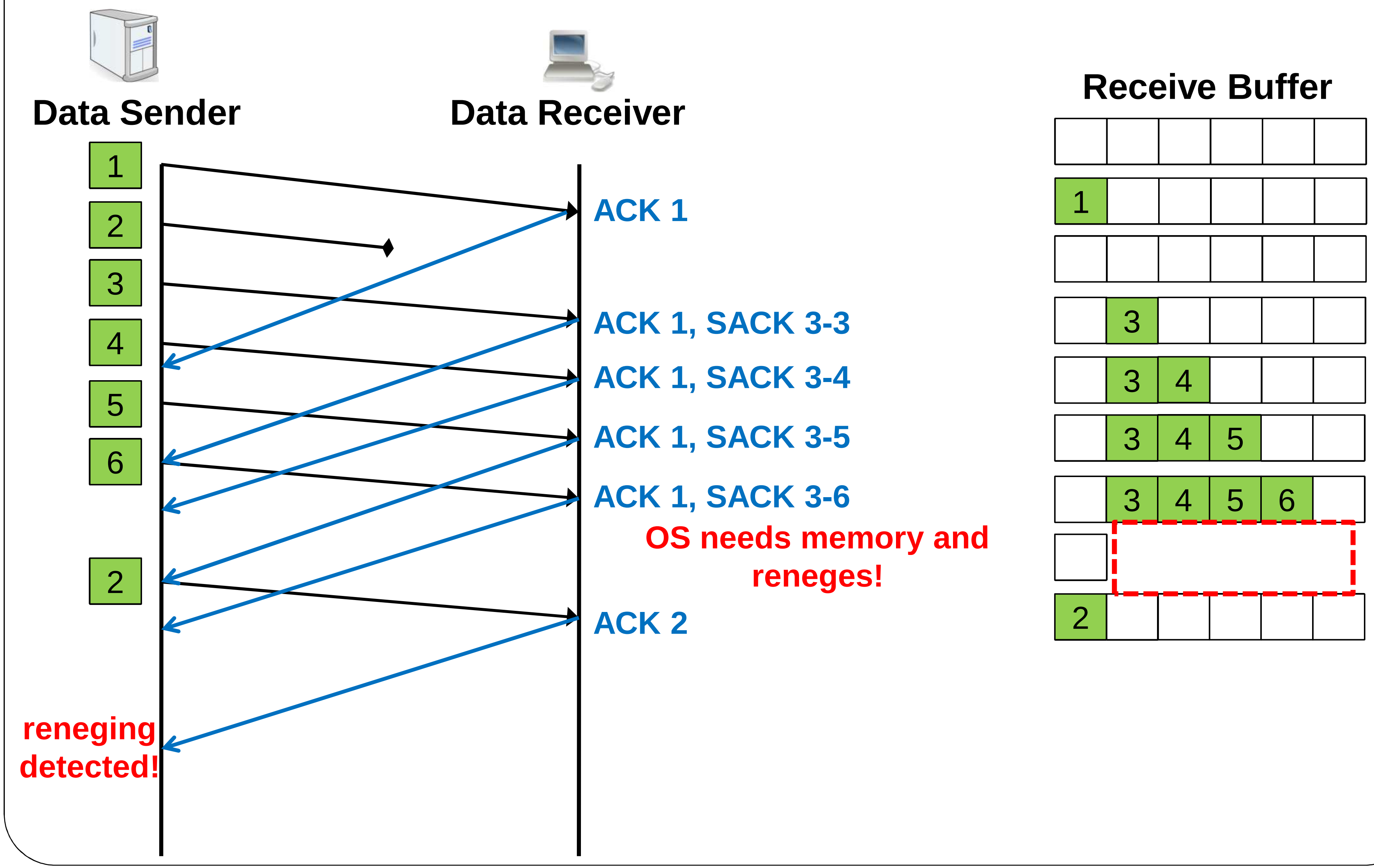
Detecting reneging at SCTP data sender



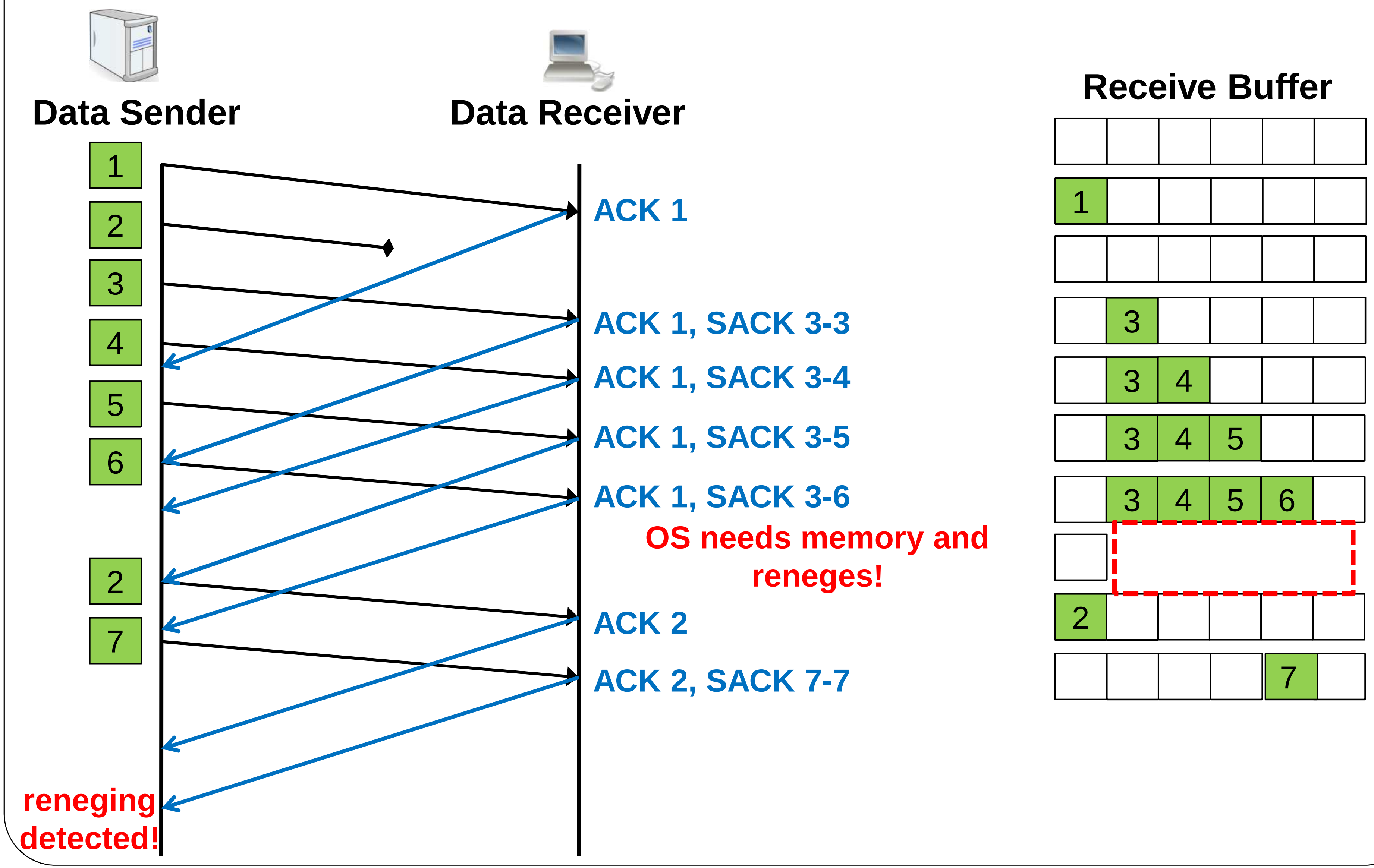
Detecting reneging at SCTP data sender



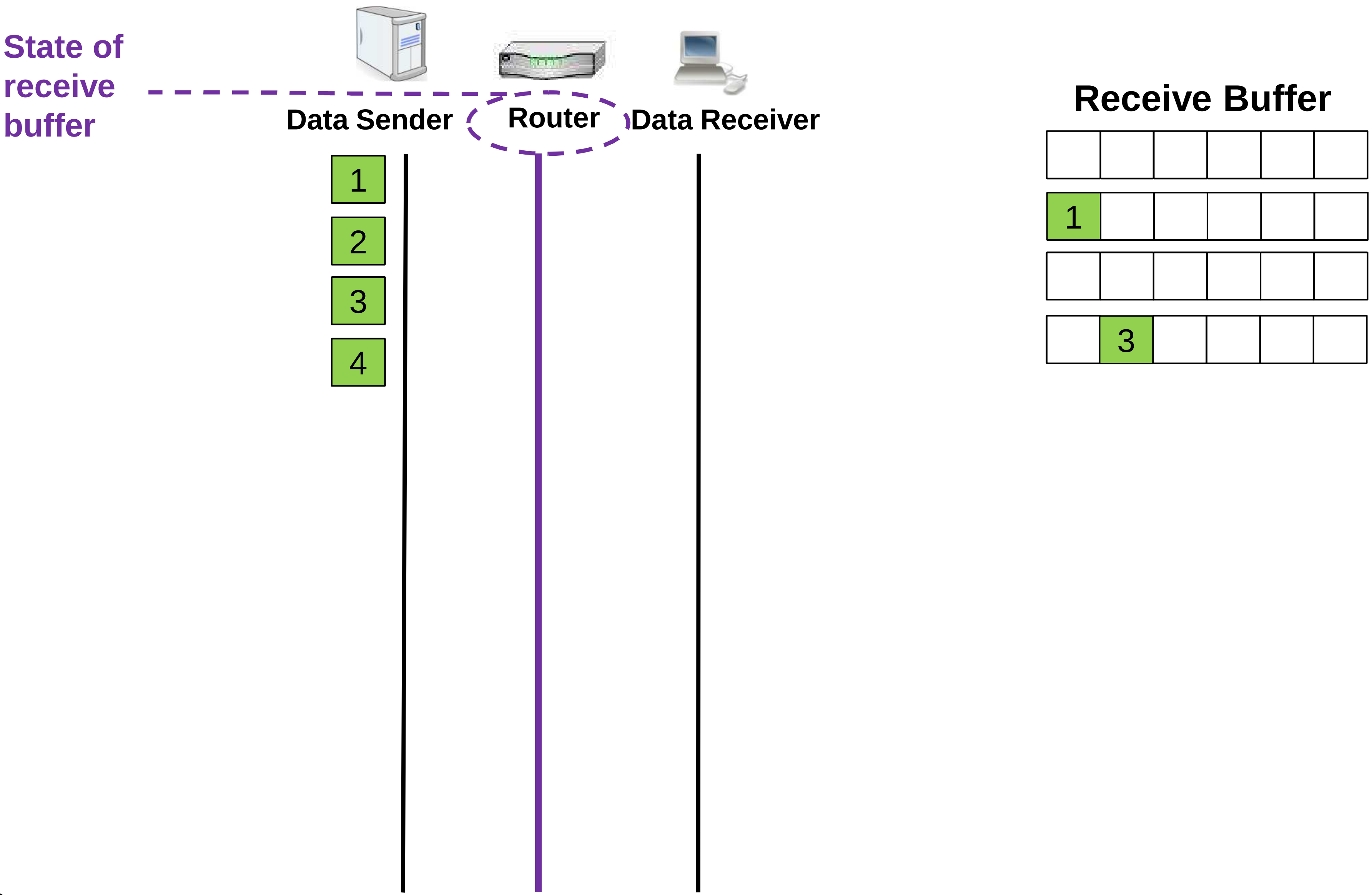
Detecting reneging at SCTP data sender



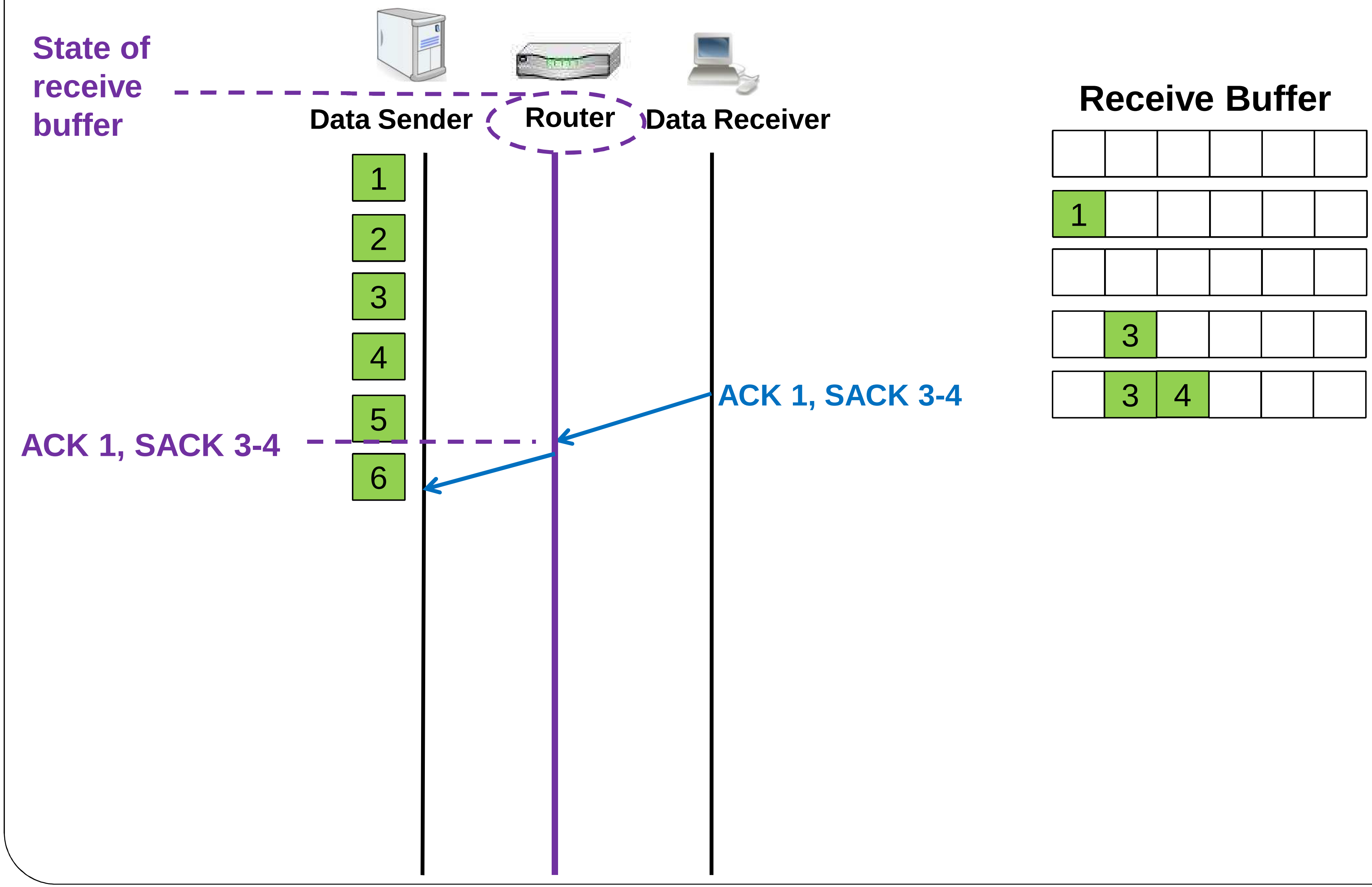
Detecting reneging at SCTP data sender



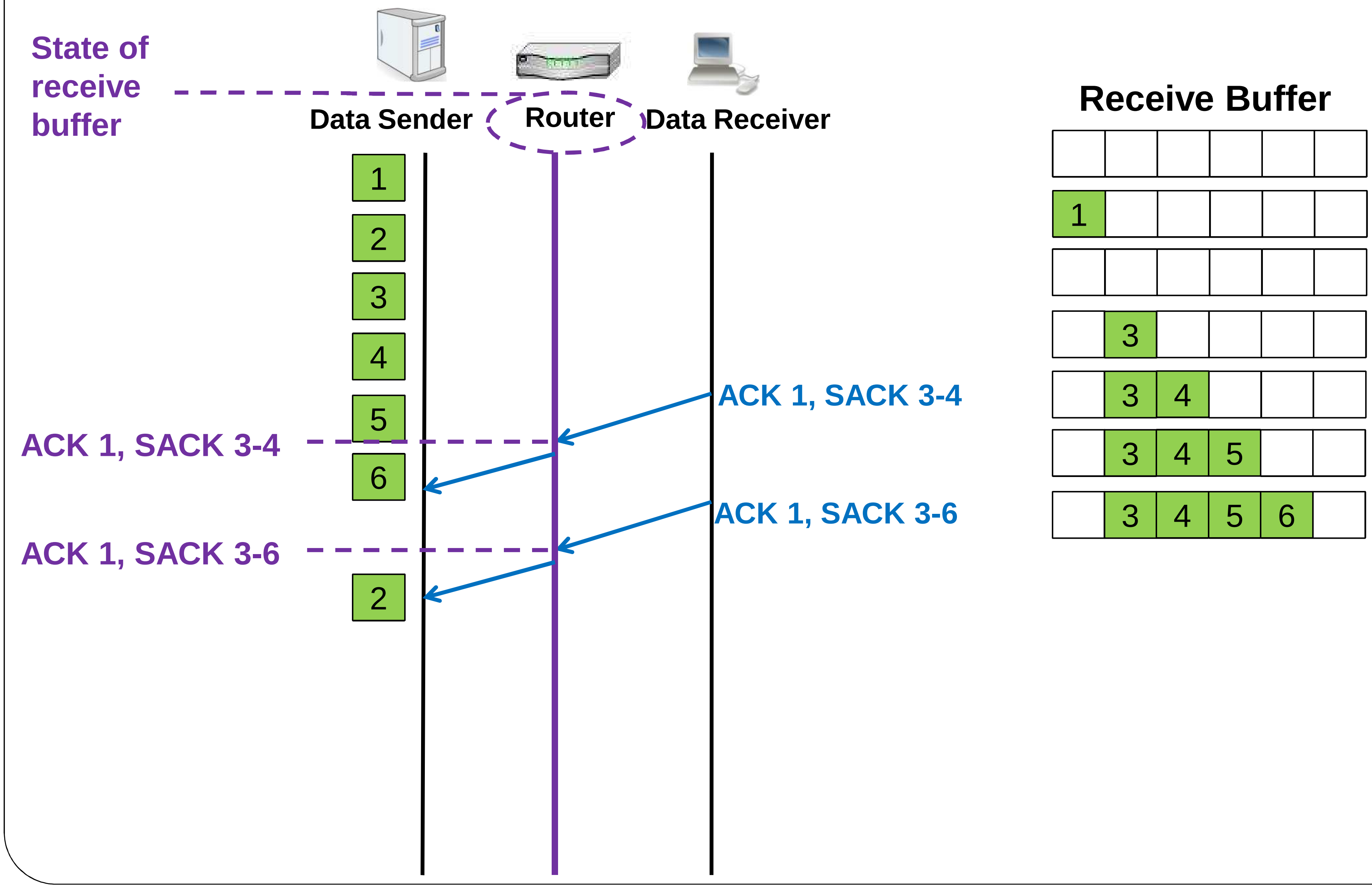
TCP reneging detected at a router



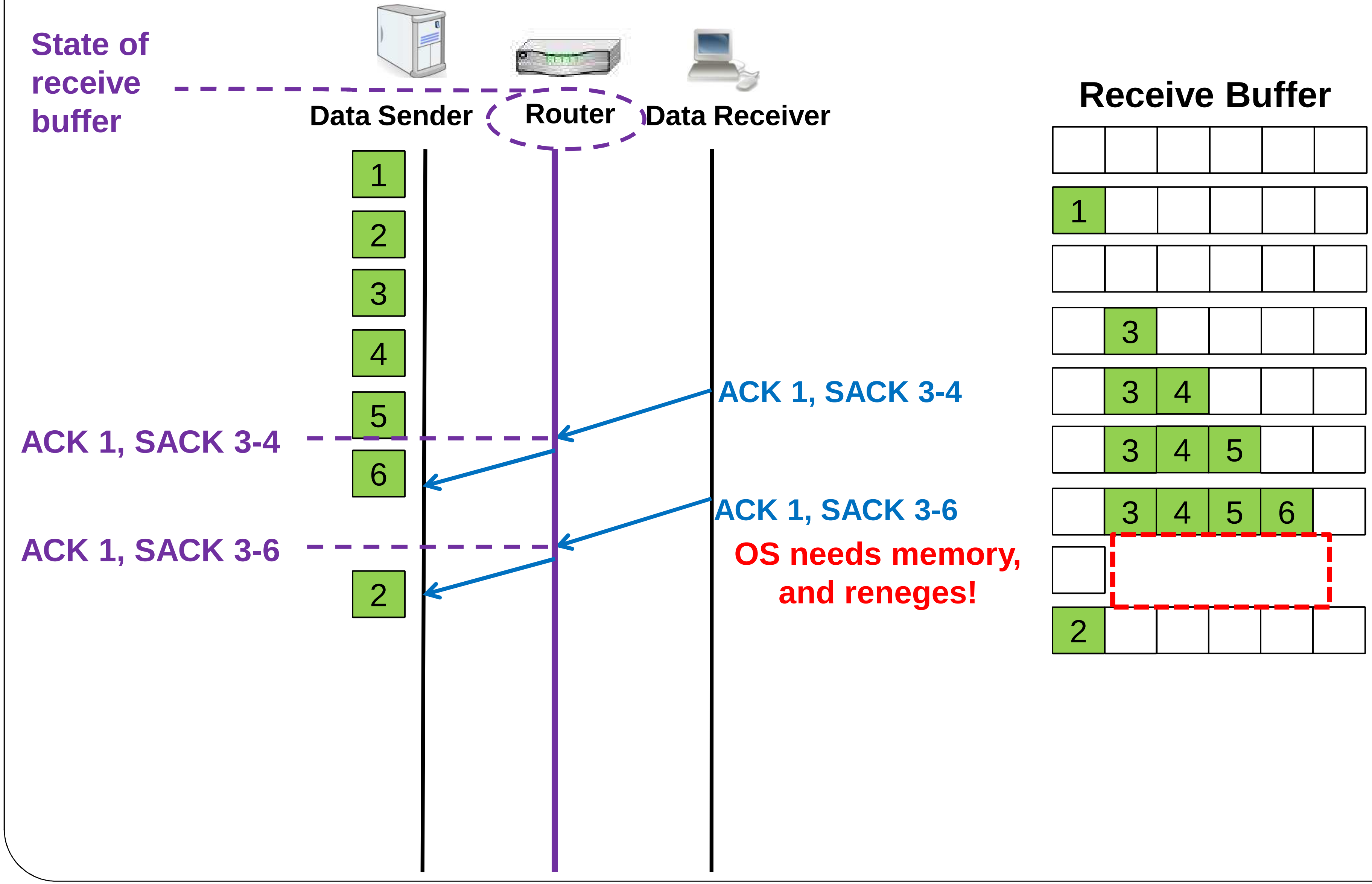
TCP reneging detected at a router



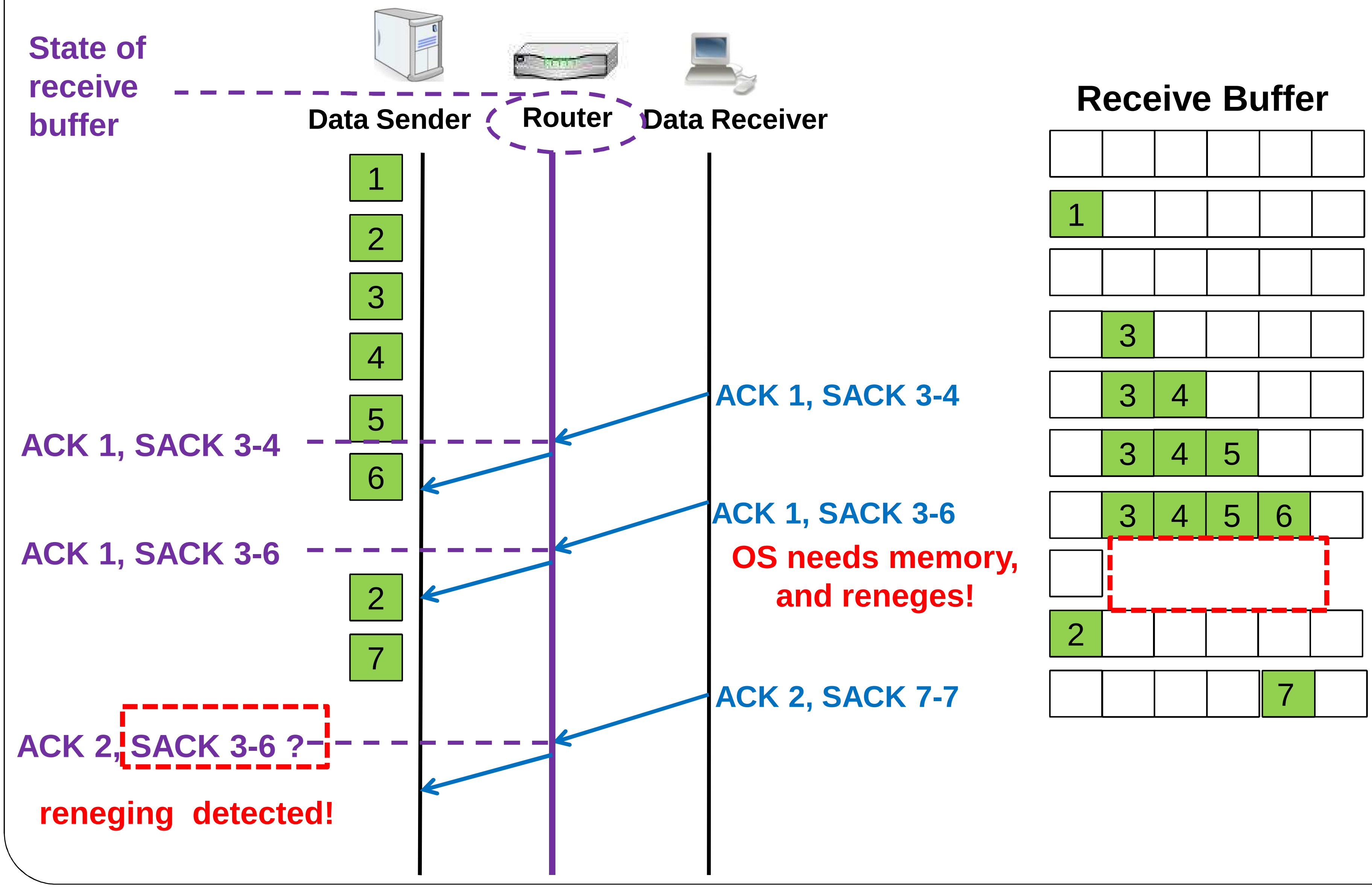
TCP reneging detected at a router



TCP reneging detected at a router



TCP reneging detected at a router



Model to detect reneging

- Current state (C) and new SACK (N) are compared
- 4 possibilities:

	Current	New
N is a superset of C $(N \supseteq C)$	SACK 12-15	SACK 12-17

Model to detect reneging

- Current state (C) and new SACK (N) are compared
- 4 possibilities:

	Current	New
N is a superset of C $(N \supseteq C)$	SACK 12-15	SACK 12-17
N is a proper subset of C $(N \subset C)$	SACK 12-17	SACK 12-13

Model to detect reneging

- Current state (C) and new SACK (N) are compared
- 4 possibilities:

	Current	New
N is a superset of C $(N \supseteq C)$	SACK 12-15	SACK 12-17
N is a proper subset of C $(N \subset C)$	SACK 12-17	SACK 12-13
N does not intersect with C $(N \cap C = \emptyset)$.	SACK 12-17	SACK 22-25

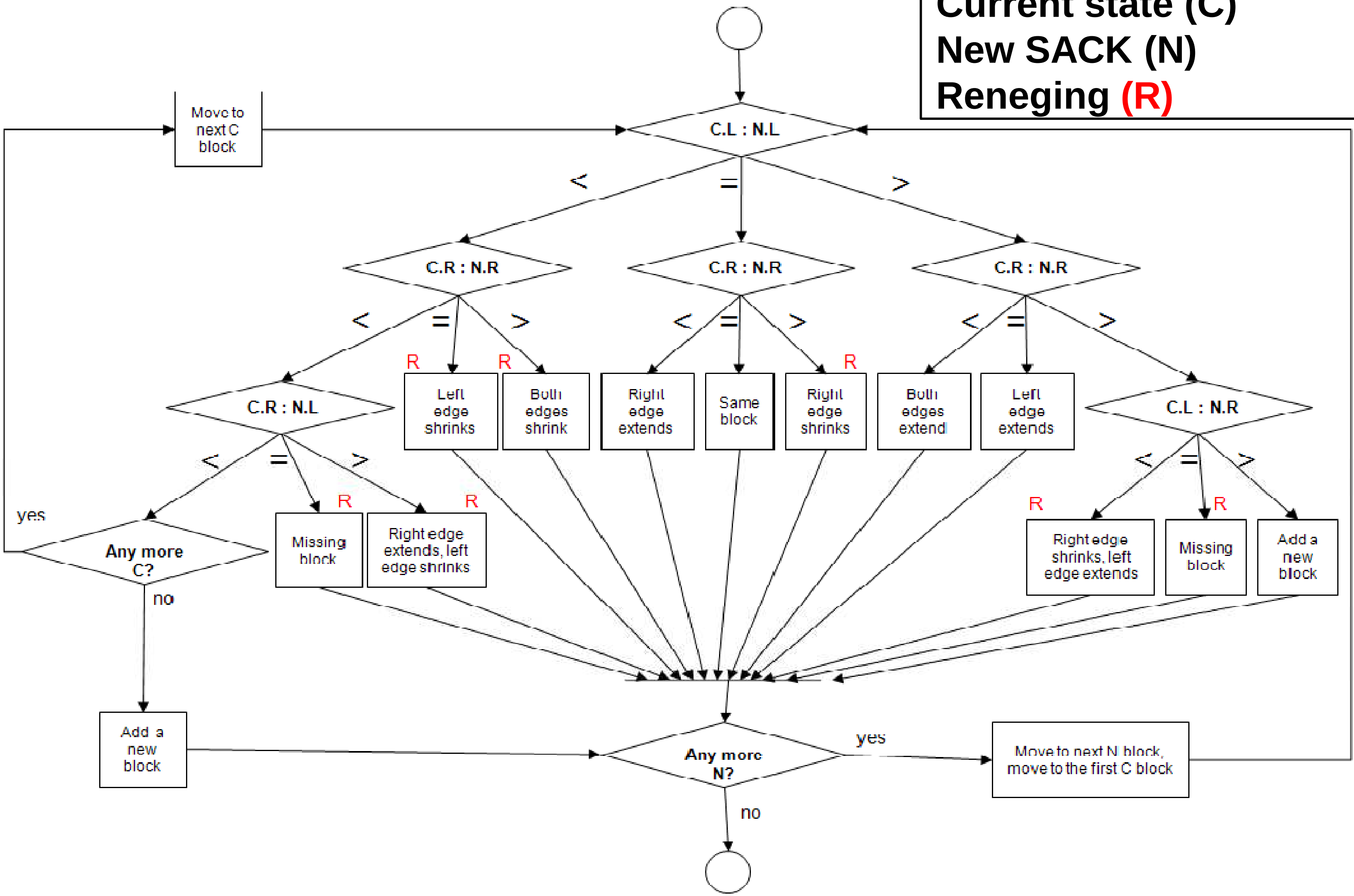
Model to detect reneging

- Current state (C) and new SACK (N) are compared
- 4 possibilities:

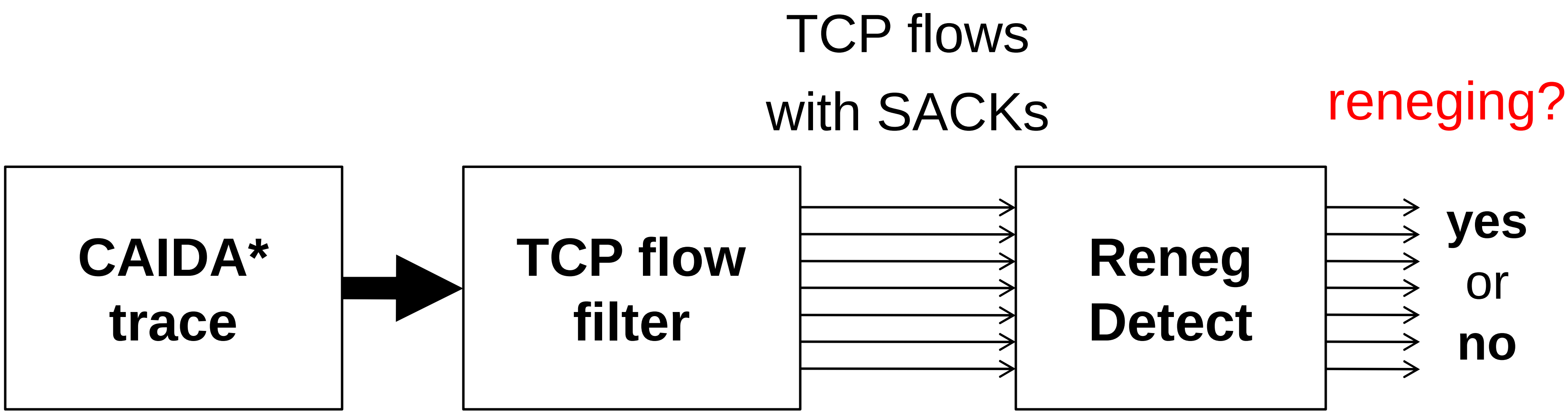
	Current	New
N is a superset of C $(N \supseteq C)$	SACK 12-15	SACK 12-17
N is a proper subset of C $(N \subset C)$	SACK 12-17	SACK 12-13
N does not intersect with C $(N \cap C = \emptyset)$.	SACK 12-17	SACK 22-25
N intersects with C, and N and C each have some data not in C and N, respectively $((N \cap C \neq \emptyset) \wedge !(N \supseteq C) \wedge !(N \supset C))$	SACK 12-17	SACK 15-20

Model to detect reneging

Current state (C)
New SACK (N)
Reneging (R)



Model to detect reneging



- .pcap

- tshark
- editcap
- mergecap

- ~4600 lines of C code
- ACK reordering check

*Cooperative Association for Internet Data Analysis

OUTLINE

1. What is data reneging?
2. Why study reneging?
3. A model to detect reneging
4. Model verification
5. Work in progress

Model verification

- RenegDetect was tested with synthetic TCP flows
  - Created reneging flows with *text2pcap*
  - All reneging flows were identified correctly
- RenegDetect was tested with real TCP flows from CAIDA Internet traces
  - At first, reneging seemed to occur frequently
  - On closer inspection, we found that many SACK implementations are incorrect !

•Ekiz, Rahman, Amer, "**Misbehaviors in SACK generation**" (submitted)

Incorrect SACK implementations

Operating System	Misbehavior						
	A	***B***	***C***	***D***	***E***	***F***	***G***
FreeBSD 5.3, 5.4	**Y**			**Y**			
Linux 2.2.20 (Debian 3)						**Y**	
Linux 2.4.18 (Red Hat 8)						**Y**	
Linux 2.4.22 (Fedora 1)						**Y**	
Linux 2.6.12 (Ubuntu 5.10)						**Y**	
Linux 2.6.15 (Ubuntu 6.06)						**Y**	
Linux 2.6.18 (Debian 4)						**Y**	
OpenBSD 4.2, 4.5, 4.6, 4.7	**Y**			**Y**			
OpenSolaris 2008.05						**Y**	**Y**
OpenSolaris 2009.06						**Y**	**Y**
Solaris 10							**Y**
Windows 2000	**Y**	**Y**	**Y**	**Y**	**Y**		
Windows XP	**Y**	**Y**	**Y**	**Y**	**Y**		
Windows Server 2003	**Y**	**Y**	**Y**	**Y**	**Y**		
Windows Vista				**Y**	**Y**		
Windows Server 2008				**Y**	**Y**		
Windows 7				**Y**	**Y**		

OUTLINE

1. What is data reneging?
2. Why study reneging?
3. A model to detect reneging
4. Model verification
5. Work in progress

Experiment design – how to "prove" reneging does not happen?

- Event A: TCP flow reneges
- Hypothesis:

$$H_0: p(A) \geq 10^{-5}$$

- We want to design an experiment which rejects H_0 with 95% confidence to conclude

$$p(A) < 10^{-5}$$

- Our experiment will observe n TCP flows hoping to NOT find even a single instance of reneging

$$\mathrm{P}(\mathrm{k} = 0 \mid H_0) < .05$$

$$p_n(0) = (1 - 10^{-5})^n$$

$$(1 - 10^{-5})^n < 0.05$$

- Using MAPLE, n ≥ 299,572

Questions?

Data reneging in OSes

- Reneging in Linux (version 2.6.28.7)
  - *tcp_prune_ofo_queue*() deletes out-of-order data
- Reneging in FreeBSD, Mac OS
  - *net.inet.tcp.do_tcpdrain* *sysctl* turns reneging on/off
  - *tcp_drain*() deletes out-of-order data

Data reneging in Linux

```c
/*
 * Purge the out-of-order queue.
 * Return true if queue was pruned.
 */
static int tcp_prune_ofo_queue(struct sock *sk)
{
	struct tcp_sock *tp = tcp_sk(sk);
	int res = 0;

	if (!skb_queue_empty(&tp->out_of_order_queue)) {
		NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_OFOPRUNED);
		__skb_queue_purge(&tp->out_of_order_queue);

		/* Reset SACK state.  A conforming SACK implementation will
		 * do the same at a timeout based retransmit.  When a connection
		 * is in a sad state like this, we care only about integrity
		 * of the connection not performance.
		 */
		if (tp->rx_opt.sack_ok)
			tcp_sack_reset(&tp->rx_opt);
		sk_mem_reclaim(sk);
		res = 1;
	}
	return res;
}
```

3. Inferring the state of receive buffer

TCP Segments with n SACK options	Enough space for another SACK option	Not enough space for another SACK option
n=1	~88%	0%
n=2	~11%	0%
n=3	0.7%	0.20%
n=4	n/a	0.15%
Total number of TCP segments		780,798 (100%)

3. Inferring the state of receive buffer

TCP Segments with n SACK options	Enough space for another SACK option	Not enough space for another SACK option
n=1	~88%	0%
n=2	~11%	0%
n=3	0.7%	**0.20%**
n=4	n/a	**0.15%**
Total number of TCP segments		780,798 (100%)

Misbehaviors in SACK generation

- 7 misbehaviors are observed in CAIDA traces
- We designed TBIT tests to verify SACK generation
- 27 OSes are tested
- RenegDetect is updated to identify those misbehaviors

Example TBIT test