CS162
Operating Systems and Systems Programming
Lecture 18
TCP's Flow Control, Transactions

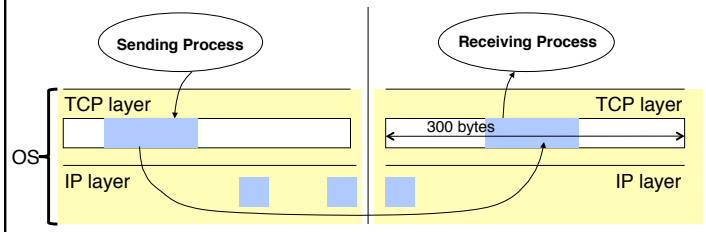
October 31, 2012
Ion Stoica
http://inst.eecs.berkeley.edu/~cs162

Goals of Today's Lecture

- TCP flow control (continued)
- Transactions (ACID semantics)

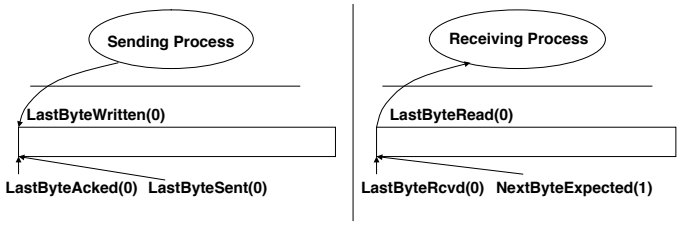
Note: Some slides and/or pictures in the following are adapted from lecture notes by Mike Franklin.

TCP Flow Control



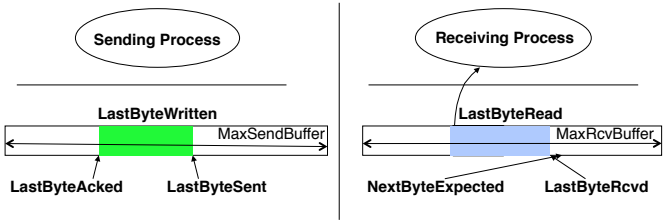
- Example assumptions:
  - Maximum IP packet size = 100 bytes
  - Size of the receiving buffer (MaxRcvBuf) = 300 bytes
- Recall, ack indicates the next expected byte in-sequence, not the last received byte
- Use circular buffers

TCP Flow Control



- LastByteWritten: last byte written by sending process
- LastByteSent: last byte sent by sender to receiver
- LastByteAcked: last ack received by sender from receiver
- LastByteRcvd: last byte received by receiver from sender
- NextByteExpected: last in-sequence byte expected by receiver
- LastByteRead: last byte read by the receiving process

TCP Flow Control



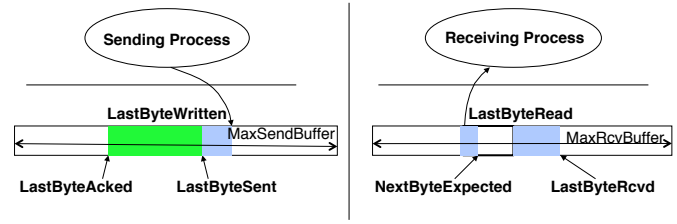
- AdvertisedWindow: number of bytes TCP receiver can receive

  AdvertisedWindow = MaxRcvBuffer – (LastByteRcvd – LastByteRead)

- SenderWindow: number of bytes TCP sender can send

  SenderWindow = AdvertisedWindow – (LastByteSent – LastByteAcked)

TCP Flow Control



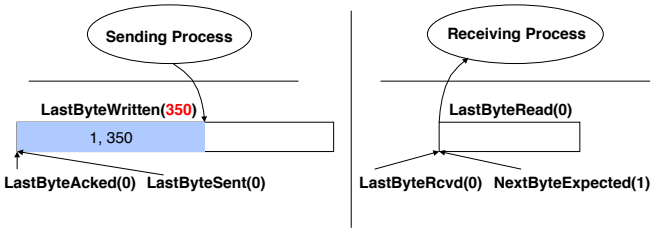
- Still true if receiver missed data….

  AdvertisedWindow = MaxRcvBuffer – (LastByteRcvd – LastByteRead)

- WriteWindow: number of bytes sending process can write

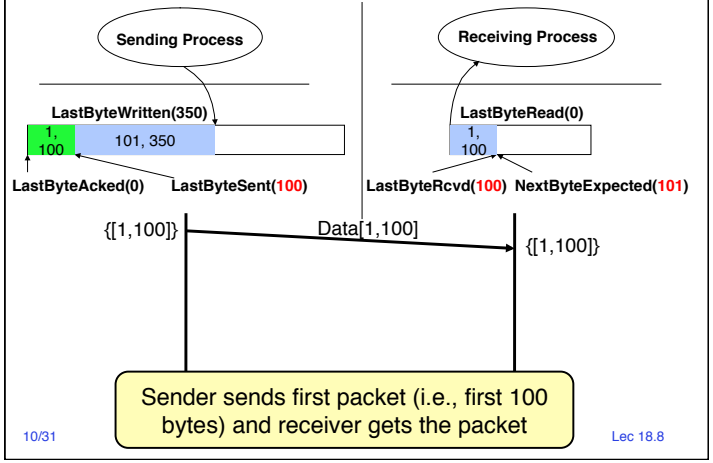
  WriteWindow = MaxSendBuffer – (LastByteWritten – LastByteAcked)

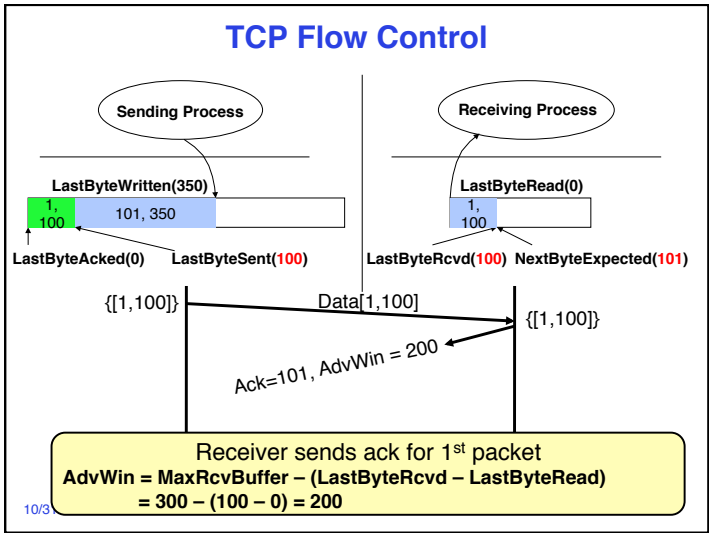
TCP Flow Control

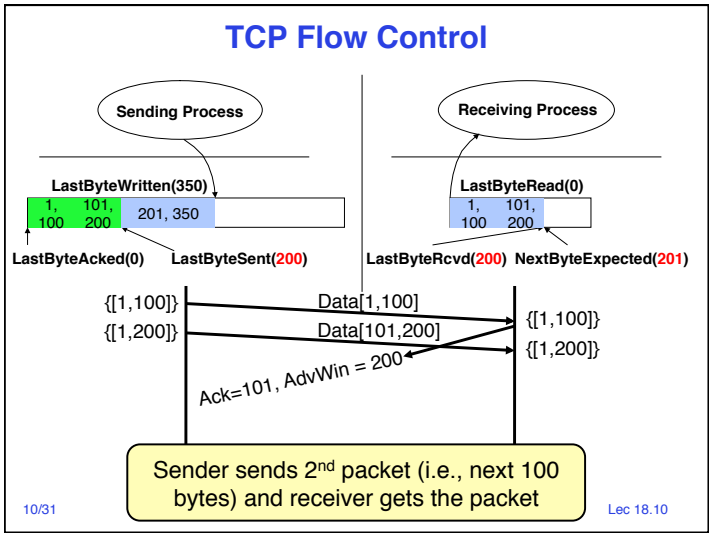


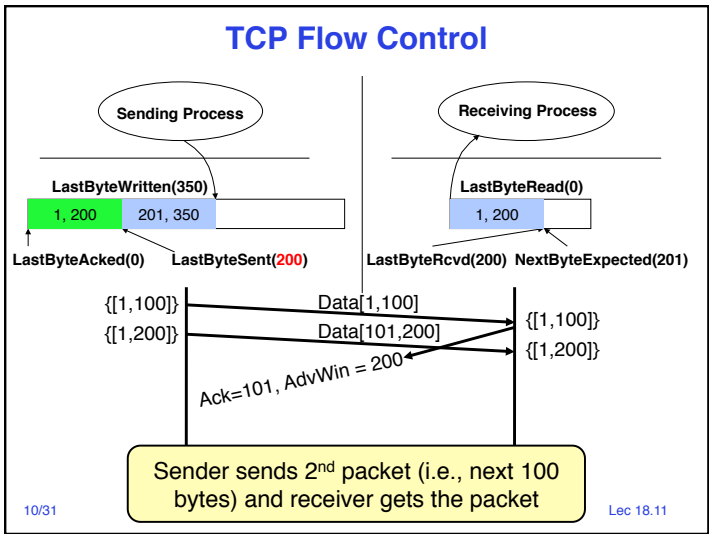
- Sending app sends 350 bytes
- Recall:
  - We assume IP only accepts packets no larger than 100 bytes
  - MaxRcvBuf = 300 bytes, so initial Advertised Window = 300 bytes

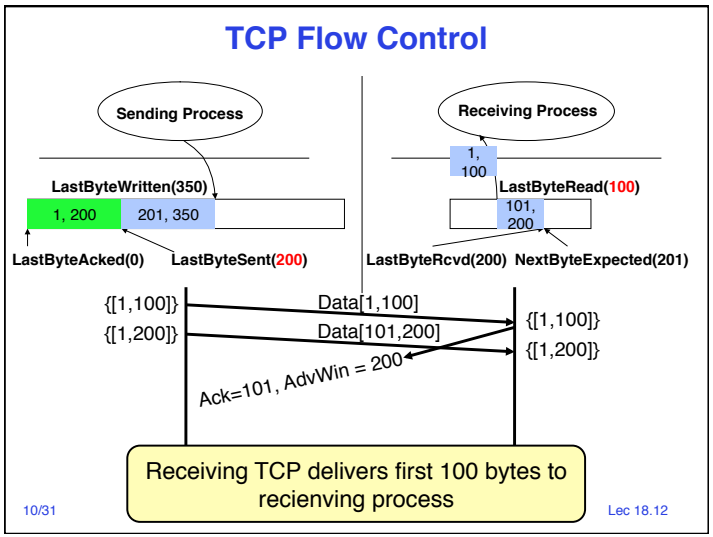
TCP Flow Control

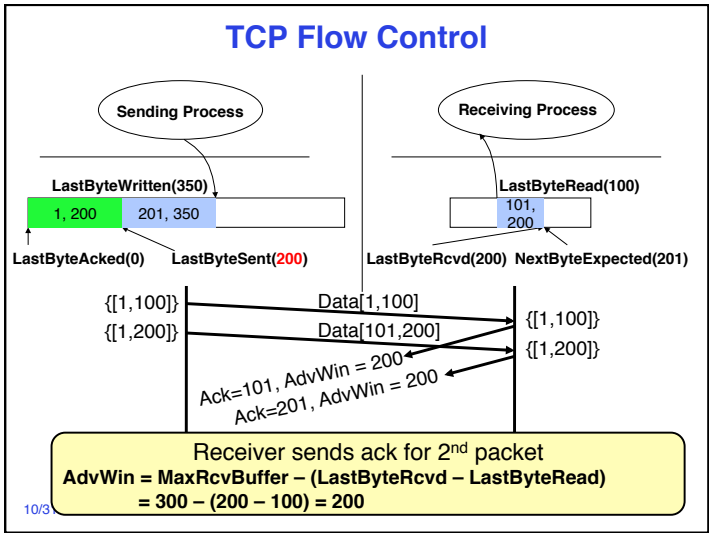


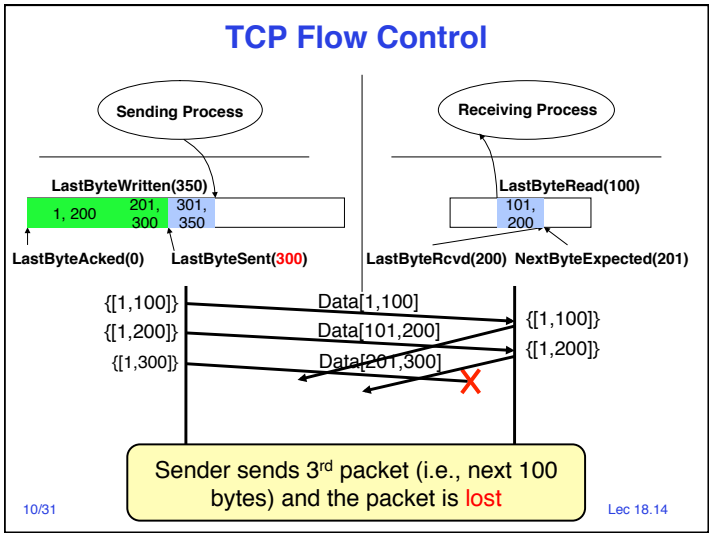

TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
LastByteRead(0)
1, 100
101, 350
1, 100
LastByteAcked(0)
LastByteSent(100)
LastByteRcvd(100)
NextByteExpected(101)
{[1,100]}
Data[1,100]
{[1,100]}
Ack=101, AdvWin = 200
Receiver sends ack for 1st packet
AdvWin = MaxRcvBuffer – (LastByteRcvd – LastByteRead)
= 300 – (100 – 0) = 200
10/31

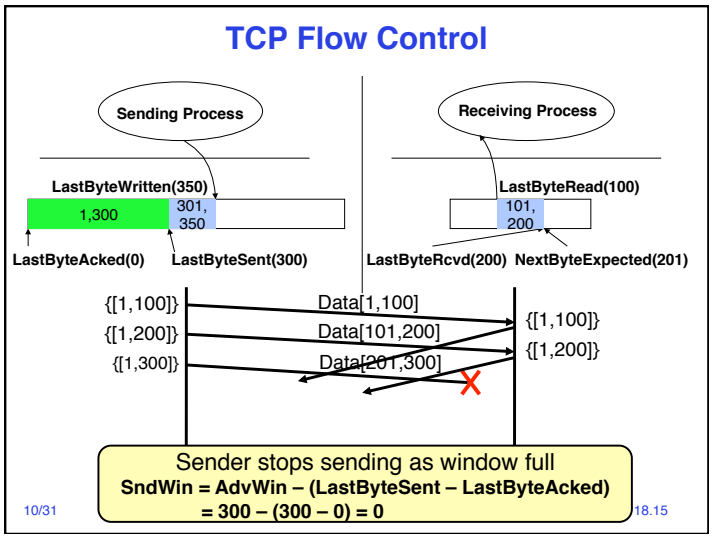

TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
LastByteRead(0)
1, 100
101, 200
201, 350
1, 100
101, 200
LastByteAcked(0)
LastByteSent(200)
LastByteRcvd(200)
NextByteExpected(201)
{[1,100]}
{[1,200]}
Data[1,100]
Data[101,200]
{[1,100]}
{[1,200]}
Ack=101, AdvWin = 200
Sender sends 2nd packet (i.e., next 100 bytes) and receiver gets the packet
10/31
Lec 18.10

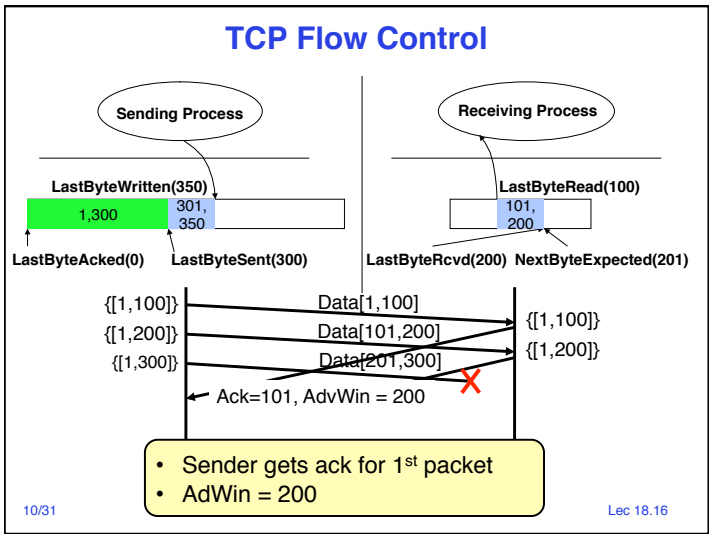

TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
LastByteRead(0)
1, 200
201, 350
1, 200
LastByteAcked(0)
LastByteSent(200)
LastByteRcvd(200)
NextByteExpected(201)
{[1,100]}
{[1,200]}
Data[1,100]
Data[101,200]
{[1,100]}
{[1,200]}
Ack=101, AdvWin = 200
Sender sends 2nd packet (i.e., next 100 bytes) and receiver gets the packet
10/31
Lec 18.11

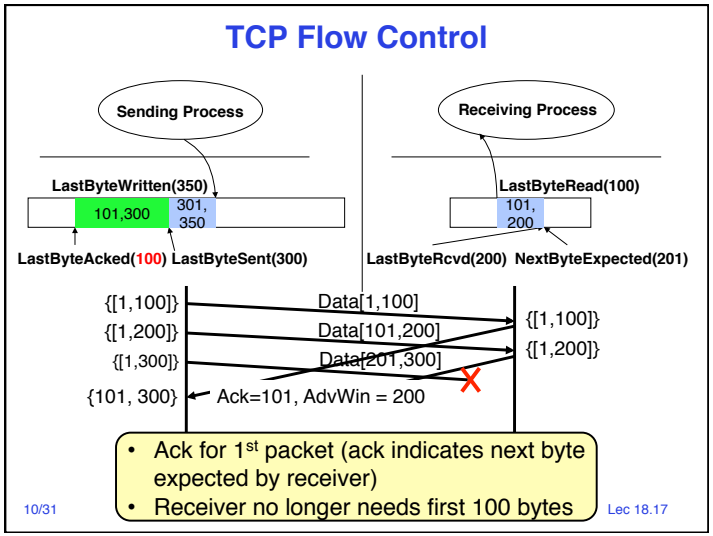

TCP Flow Control
Sending Process
Receiving Process
1, 100
LastByteWritten(350)
LastByteRead(100)
1, 200
201, 350
101, 200
LastByteAcked(0)
LastByteSent(200)
LastByteRcvd(200)
NextByteExpected(201)
{[1,100]}
{[1,200]}
Data[1,100]
Data[101,200]
{[1,100]}
{[1,200]}
Ack=101, AdvWin = 200
Receiving TCP delivers first 100 bytes to recienving process
10/31
Lec 18.12

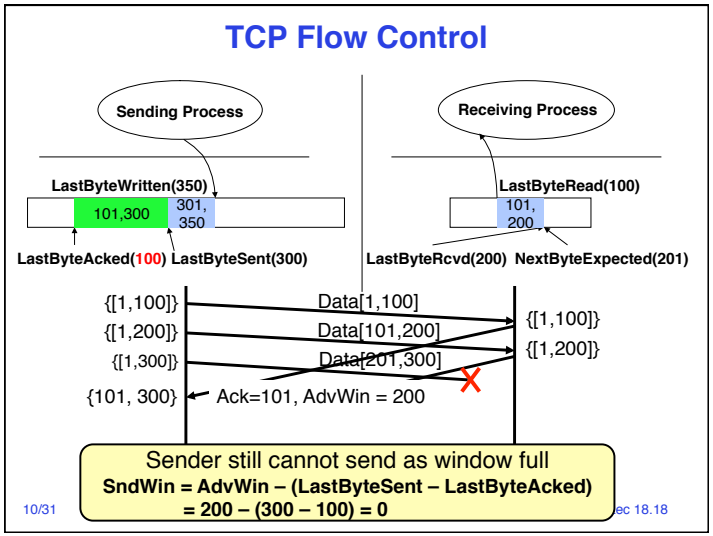

TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
1, 200
201, 350
LastByteRead(100)
101, 200
LastByteAcked(0)
LastByteSent(200)
LastByteRcvd(200)
NextByteExpected(201)
{[1,100]}
{[1,200]}
Data[1,100]
Data[101,200]
{[1,100]}
{[1,200]}
Ack=101, AdvWin = 200
Ack=201, AdvWin = 200
Receiver sends ack for 2nd packet
AdvWin = MaxRcvBuffer – (LastByteRcvd – LastByteRead)
= 300 – (200 – 100) = 200
10/3

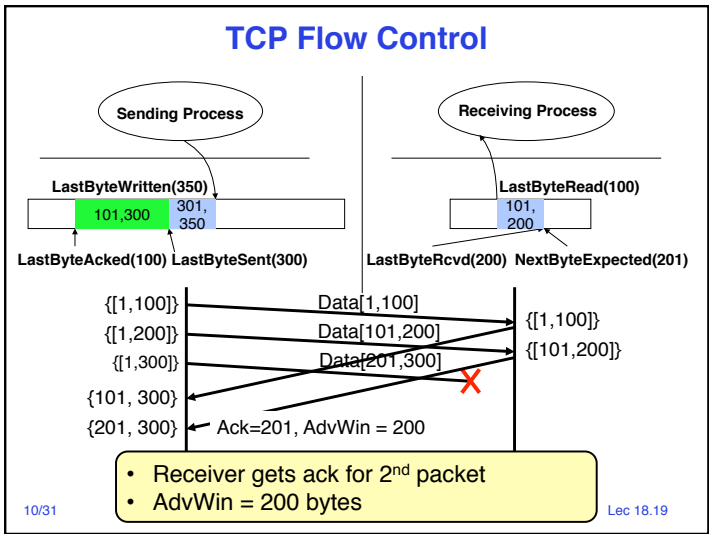

TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
1, 200
201, 300
301, 350
LastByteRead(100)
101, 200
LastByteAcked(0)
LastByteSent(300)
LastByteRcvd(200)
NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
Data[1,100]
Data[101,200]
Data[201,300]
{[1,100]}
{[1,200]}
Sender sends 3rd packet (i.e., next 100 bytes) and the packet is lost
10/31
Lec 18.14

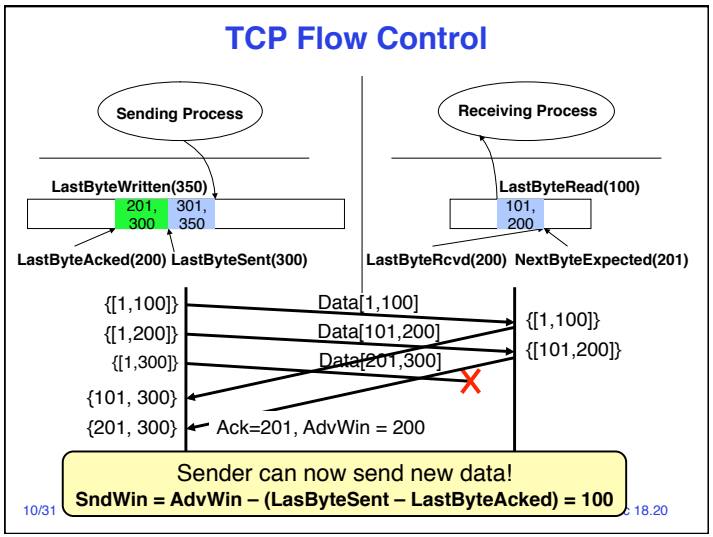

TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
1,300
301, 350
LastByteRead(100)
101, 200
LastByteAcked(0)
LastByteSent(300)
LastByteRcvd(200)
NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
Data[1,100]
Data[101,200]
Data[201,300]
{[1,100]}
{[1,200]}
Sender stops sending as window full
SndWin = AdvWin – (LastByteSent – LastByteAcked)
= 300 – (300 – 0) = 0
10/31
18.15

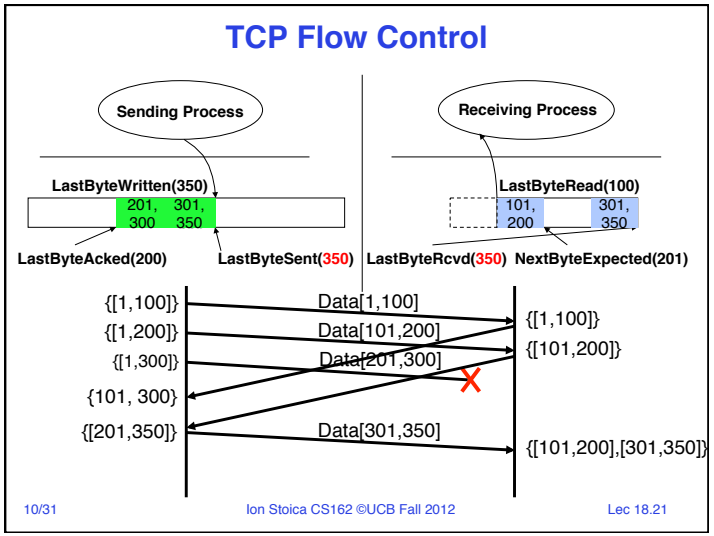

TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
1,300
301, 350
LastByteRead(100)
101, 200
LastByteAcked(0)
LastByteSent(300)
LastByteRcvd(200)
NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
Data[1,100]
Data[101,200]
Data[201,300]
{[1,100]}
{[1,200]}
Ack=101, AdvWin = 200
• Sender gets ack for 1st packet
• AdWin = 200
10/31
Lec 18.16

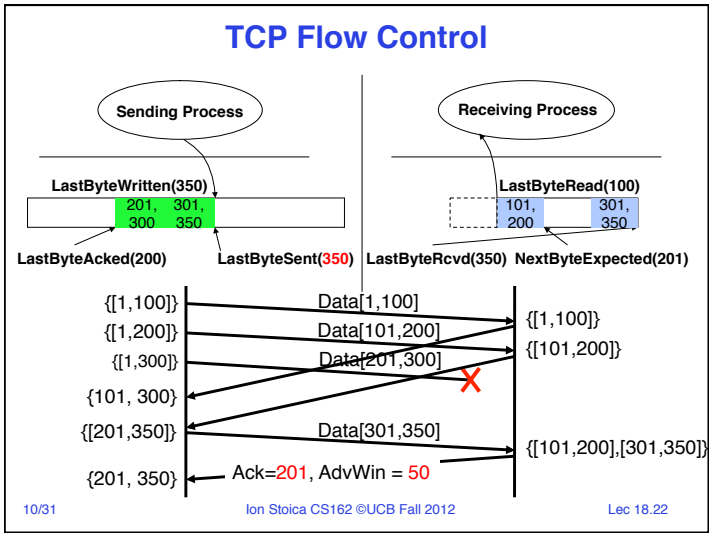

TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
101,300
301, 350
LastByteRead(100)
101, 200
LastByteAcked(100) LastByteSent(300)
LastByteRcvd(200) NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
{101, 300}
Data[1,100]
Data[101,200]
Data[201,300]
Ack=101, AdvWin = 200
{[1,100]}
{[1,200]}
• Ack for 1st packet (ack indicates next byte expected by receiver)
• Receiver no longer needs first 100 bytes
10/31
Lec 18.17

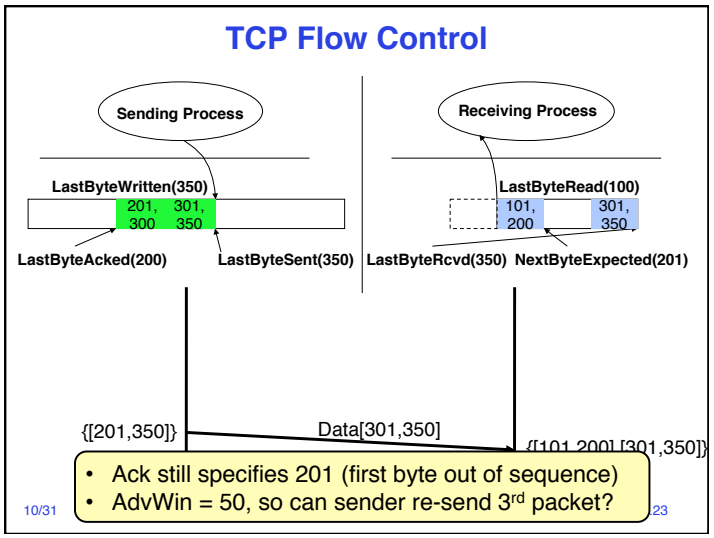

TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
101,300
301, 350
LastByteRead(100)
101, 200
LastByteAcked(100) LastByteSent(300)
LastByteRcvd(200) NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
{101, 300}
Data[1,100]
Data[101,200]
Data[201,300]
Ack=101, AdvWin = 200
{[1,100]}
{[1,200]}
Sender still cannot send as window full
SndWin = AdvWin – (LastByteSent – LastByteAcked)
= 200 – (300 – 100) = 0
10/31
ec 18.18

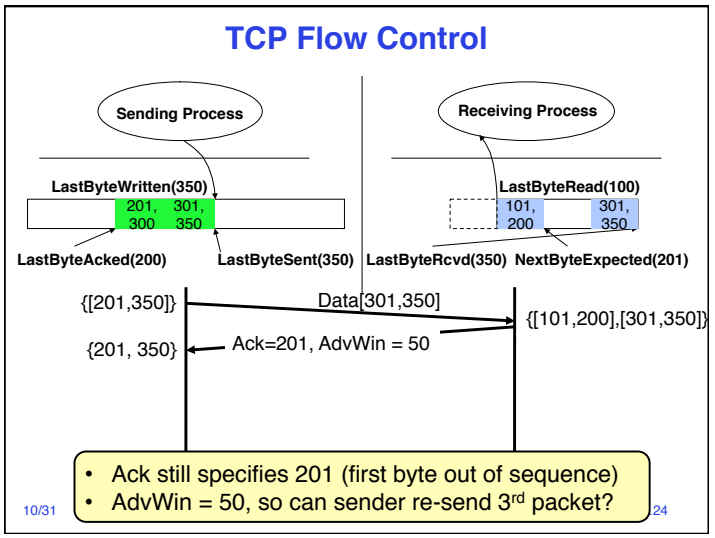

TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
101,300
301, 350
LastByteRead(100)
101, 200
LastByteAcked(100) LastByteSent(300)
LastByteRcvd(200) NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
{101, 300}
{201, 300}
Data[1,100]
Data[101,200]
Data[201,300]
Ack=201, AdvWin = 200
{[1,100]}
{[101,200]}
• Receiver gets ack for 2nd packet
• AdvWin = 200 bytes
10/31
Lec 18.19

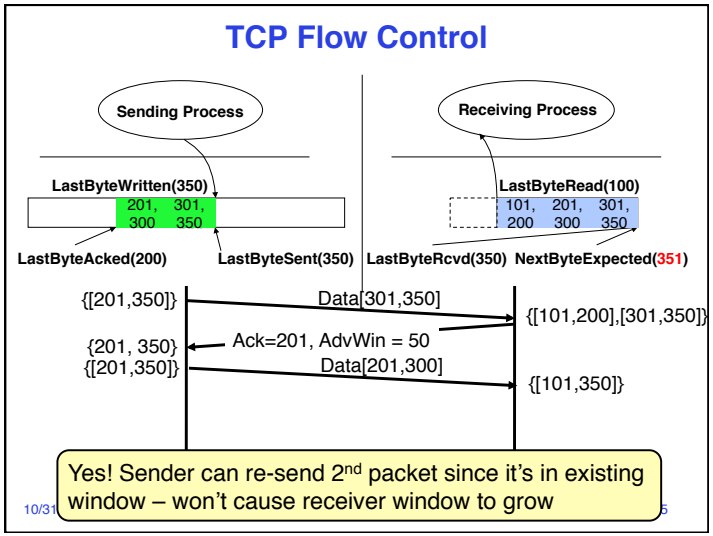

TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
201, 300
301, 350
LastByteRead(100)
101, 200
LastByteAcked(200) LastByteSent(300)
LastByteRcvd(200) NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
{101, 300}
{201, 300}
Data[1,100]
Data[101,200]
Data[201,300]
Ack=201, AdvWin = 200
{[1,100]}
{[101,200]}
Sender can now send new data!
SndWin = AdvWin – (LasByteSent – LastByteAcked) = 100
10/31
c 18.20

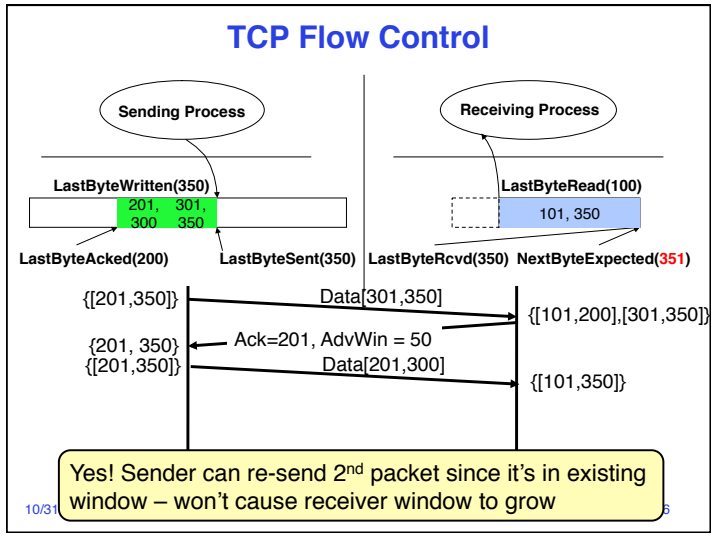

TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
201, 300
301, 350
LastByteRead(100)
101, 200
301, 350
LastByteAcked(200)
LastByteSent(350)
LastByteRcvd(350)
NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
{101, 300}
{[201,350]}
Data[1,100]
Data[101,200]
Data[201,300]
Data[301,350]
{[1,100]}
{[101,200]}
{[101,200],[301,350]}
10/31
Ion Stoica CS162 ©UCB Fall 2012
Lec 18.21

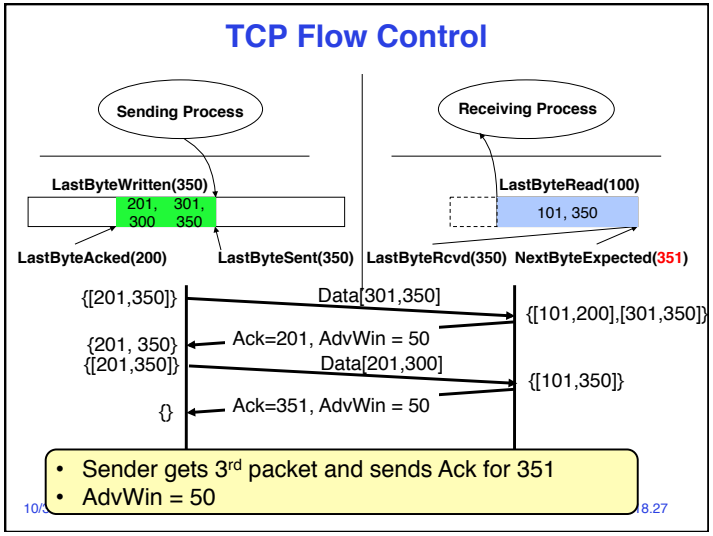

TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
201, 300
301, 350
LastByteRead(100)
101, 200
301, 350
LastByteAcked(200)
LastByteSent(350)
LastByteRcvd(350)
NextByteExpected(201)
{[1,100]}
{[1,200]}
{[1,300]}
{101, 300}
{[201,350]}
{201, 350}
Data[1,100]
Data[101,200]
Data[201,300]
Data[301,350]
Ack=201, AdvWin = 50
{[1,100]}
{[101,200]}
{[101,200],[301,350]}
10/31
Ion Stoica CS162 ©UCB Fall 2012
Lec 18.22

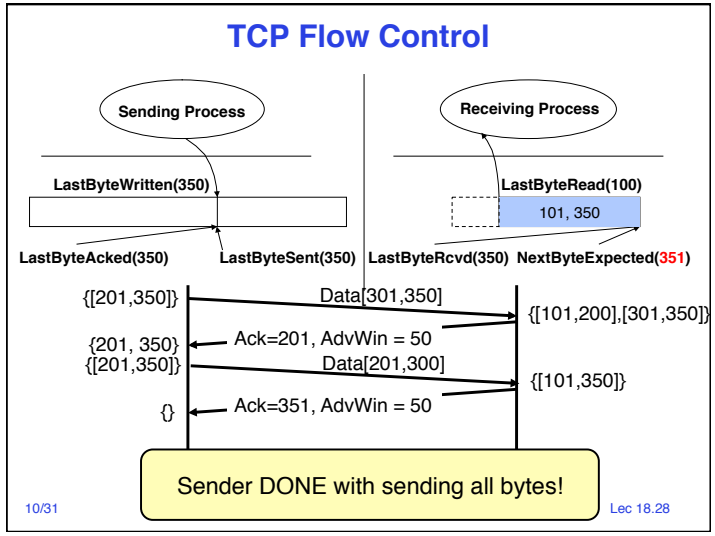

TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
201, 300
301, 350
LastByteRead(100)
101, 200
301, 350
LastByteAcked(200)
LastByteSent(350)
LastByteRcvd(350)
NextByteExpected(201)
{[201,350]}
Data[301,350]
• Ack still specifies 201 (first byte out of sequence)
• AdvWin = 50, so can sender re-send 3rd packet?
10/31
23


TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
201, 300
301, 350
LastByteRead(100)
101, 200
301, 350
LastByteAcked(200)
LastByteSent(350)
LastByteRcvd(350)
NextByteExpected(201)
{[201,350]}
{201, 350}
Data[301,350]
Ack=201, AdvWin = 50
{[101,200],[301,350]}
• Ack still specifies 201 (first byte out of sequence)
• AdvWin = 50, so can sender re-send 3rd packet?
10/31
24


TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
201, 300
301, 350
LastByteRead(100)
101, 200
201, 300
301, 350
LastByteAcked(200)
LastByteSent(350)
LastByteRcvd(350)
NextByteExpected(351)
{[201,350]}
Data[301,350]
{[101,200],[301,350]}
{201, 350}
{[201,350]}
Ack=201, AdvWin = 50
Data[201,300]
{[101,350]}
Yes! Sender can re-send 2nd packet since it's in existing window – won't cause receiver window to grow
10/31


TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
201, 300
301, 350
LastByteRead(100)
101, 350
LastByteAcked(200)
LastByteSent(350)
LastByteRcvd(350)
NextByteExpected(351)
{[201,350]}
Data[301,350]
{[101,200],[301,350]}
{201, 350}
{[201,350]}
Ack=201, AdvWin = 50
Data[201,300]
{[101,350]}
Yes! Sender can re-send 2nd packet since it's in existing window – won't cause receiver window to grow
10/31


TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
201, 300
301, 350
LastByteRead(100)
101, 350
LastByteAcked(200)
LastByteSent(350)
LastByteRcvd(350)
NextByteExpected(351)
{[201,350]}
Data[301,350]
{[101,200],[301,350]}
{201, 350}
{[201,350]}
Ack=201, AdvWin = 50
Data[201,300]
{[101,350]}
{}
Ack=351, AdvWin = 50
• Sender gets 3rd packet and sends Ack for 351
• AdvWin = 50
10/31


TCP Flow Control
Sending Process
Receiving Process
LastByteWritten(350)
LastByteRead(100)
101, 350
LastByteAcked(350)
LastByteSent(350)
LastByteRcvd(350)
NextByteExpected(351)
{[201,350]}
Data[301,350]
{[101,200],[301,350]}
{201, 350}
{[201,350]}
Ack=201, AdvWin = 50
Data[201,300]
{[101,350]}
{}
Ack=351, AdvWin = 50
Sender DONE with sending all bytes!
10/31
Lec 18.28

Quiz 18.1: Flow-Control

- Q1: True _ False _ Flow control is responsible with detecting packet losses and retransmissions
- Q2: True _ False _ Flow control always allows a sender to resend a lost packet
- Q3: True _ False _ With TCP, the receiving OS can deliver data to the application out-of-sequence (i.e., with gaps)
- Q4: True _ False _ Flow control makes sure the sender doesn't overflow the receiver

Quiz 18.1: Flow-Control

- Q1: True _ False X Flow control is responsible with detecting packet losses and retransmissions
- Q2: True X False _ Flow control always allows a sender to resend a lost packet
- Q3: True _ False X With TCP, the receiving OS can deliver data to the application out-of-sequence (i.e., with gaps)
- Q4: True X False _ Flow control makes sure the sender doesn't overflow the receiver

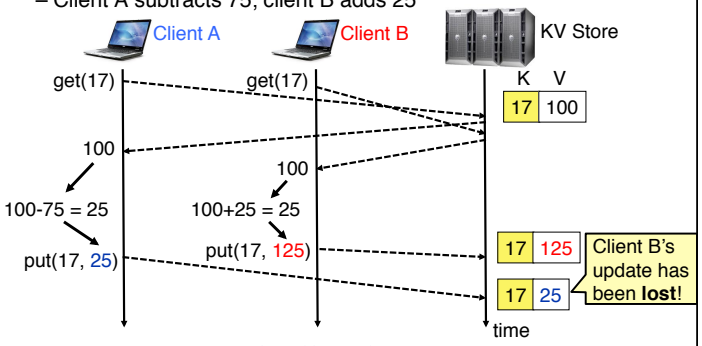
Summary: Reliability & Flow Control

- Flow control: three pairs of producer consumers
  - Sending process → sending TCP
  - Sending TCP → receiving TCP
  - Receiving TCP → receiving process
- AdvertisedWindow: tells sender how much new data can the receiver buffer
- SenderWindow: specifies how many more bytes can sender sent
  - Depends on AdvertisedWindow and on data sent since sender got AdvertisedWindow

5min Break

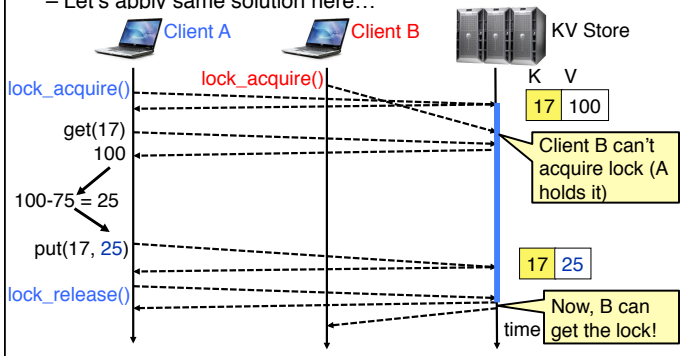
Need for Transactions

- Example: assume two clients updating same value in a key-value (KV) store at the same time
  - Client A subtracts 75; client B adds 25



Solution?

- How did we solve such problem on a single machine?
  - Critical section, e.g., use locks
  - Let's apply same solution here…



Discussion

- How does client B get the lock?
  - Pooling: periodically check whether the lock is free
  - KV storage system keeps a list of clients waiting for the lock, and gives the lock to next client in the list
- What happens if the client holding the lock crashes?
- Network latency might be higher than update operation
  - Most of the time in critical section spent waiting for messages
- What is the lock granularity?
  - Do you lock every key? Do you lock the entire storage?
  - What are the tradeoffs?

Better Solution

- Interleave reads and writes from different clients
- Provide the same semantics as clients were running one at a time
- **Transaction** – database/storage sytem's abstract view of a user program, i.e., a sequence of reads and writes

"Classic" Example: Transaction

```
BEGIN;    --BEGIN TRANSACTION
 UPDATE accounts SET balance = balance -
   100.00 WHERE name = 'Alice';

 UPDATE branches SET balance = balance -
   100.00 WHERE name = (SELECT branch_name
   FROM accounts WHERE name = 'Alice');

 UPDATE accounts SET balance = balance +
   100.00 WHERE name = 'Bob';

 UPDATE branches SET balance = balance +
   100.00 WHERE name = (SELECT branch_name
   FROM accounts WHERE name = 'Bob');
COMMIT;    --COMMIT WORK
```

Transfer $100 from Alice's account to Bob's account

The ACID properties of Transactions

- **Atomicity:** all actions in the transaction happen, or none happen
- **Consistency:** transactions maintain data integrity, e.g.,
  - Balance cannot be negative
  - Cannot reschedule meeting on February 30
- **Isolation:** execution of one transaction is isolated from that of all others; no problems from concurrency
- **Durability:** if a transaction commits, its effects persist despite crashes

Atomicity

- A transaction
  - might *commit* after completing all its operations, or
  - it could *abort* (or be aborted) after executing some operations
- Atomic Transactions: a user can think of a transaction as always either *executing all its* operations, or *not executing any* operations at all
  - Database/storage system *logs* all actions so that it can *undo* the actions of aborted transactions

Consistency

- Data follows integrity constraints (ICs)
- If database/storage system is consistent before transaction, it will be after
- System checks ICs and if they fail, the transaction rolls back (i.e., is aborted)
  - A database enforces some ICs, depending on the ICs declared when the data has been created
  - Beyond this, database does not understand the semantics of the data (e.g., it does not understand how the interest on a bank account is computed)

Isolation

- Each transaction executes as if it was running by itself
  - It cannot see the partial results of another transaction
- Techniques:
  - Pessimistic – don't let problems arise in the first place
  - Optimistic – assume conflicts are rare, deal with them *after* they happen

Durability

- Data should survive in the presence of
  - System crash
  - Disk crash → need backups
- All committed updates and only those updates are reflected in the database
  - Some care must be taken to handle the case of a crash occurring during the recovery process!

This Lecture

- Deal with **(I)solation**, by focusing on **concurrency control**
- Next lecture focus on (A)tomicity, and partially on (D)urability

Example

- Consider two transactions:
  - T1: moves $100 from account A to account B

```
T1:A := A-100; B := B+100;
```

  - T2: moves $50 from account B to account A

```
T2:A := A+50;  B := B-50;
```

- Each operation consists of (1) a read, (2) an addition/ subtraction, and (3) a write
- Example: A = A-100

```
Read(A); // R(A)
A := A - 100;
Write(A); // W(A)
```

Example (cont'd)

- Database only sees reads and writes

	Database View
T1: A:=A-100; B:=B+100; →	T1:R(A),W(A),R(B),W(B)
T2: A:=A+50; B:=B-50; →	T2:R(A),W(A),R(B),W(B)

- Assume initially: A = $1000 and B = $500
- What is the legal outcome of running T1 and T2?
  - A = $950
  - B = $550

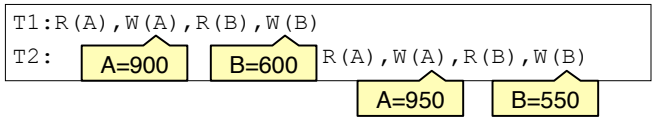
Example (cont'd)

T1: A:=A-100; B:=B+100;

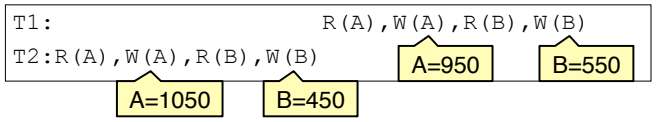
T2: A:=A+50; B:=B-50;

Initial values:
A:=1000
B:=500

- What is the outcome of the following execution?



- What is the outcome of the following execution?



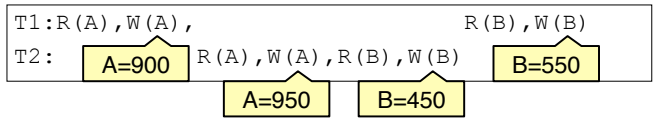
Example (cont'd)

T1: A:=A-100; B:=B+100;

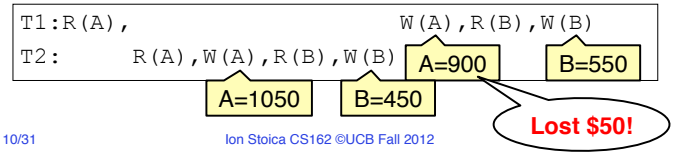
T2: A:=A+50; B:=B-50;

Initial values:
A:=1000
B:=500

- What is the outcome of the following execution?



- What is the outcome of the following execution?

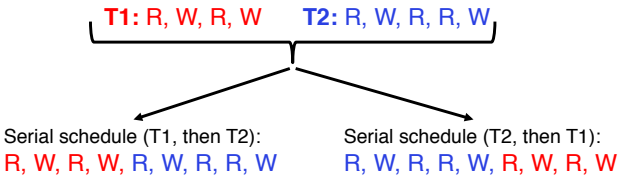


Transaction Scheduling

- Why not run only one transaction at a time?
- Answer: low system utilization
  - Two transactions cannot run simultaneously even if they access different data

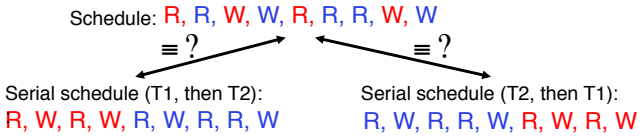
Goals of Transaction Scheduling

- Maximize system utilization, i.e., concurrency
  - Interleave operations from different transactions
- Preserve transaction semantics
  - Semantically equivalent to a **serial schedule**, i.e., one transaction runs at a time



Two Key Questions

1) Is a given schedule equivalent to a serial execution of transactions?



2) How do you come up with a schedule equivalent to a serial schedule?

Summary

- Transaction: a sequence of storage operations
- ACID:
  - Atomicity: all operations in a transaction happen, or none happens
  - Consistency: if database/storage starts consistent, it ends up consistent
  - Isolation: execution of one transaction is isolated from another
  - Durability: the results of a transaction persists