上 海 交 通 大 学 试 卷（期末 A 卷）

（2018 至 2019 学年　第 1 学期）

班级号_____ 学号_____ 姓名_____

课程名称_____计算机系统工程_____ 成绩_____

Problem-1: Network & Fault Tolerance (25')

Bob decided to develop an online voting service (OVS) to allow people to vote for things they like. The system will include a single-host server and a client-side mobile application. The interfaces between the client and server are shown below.

```
# client side:
   vote(item id);//vote for the item and get a response

   check(item id);//get the vote account of the item

# server side
   confirm();//server handle vote requests and check requests
               from clients
```

1. At first, Bob uses UDP for OVS. He finds that UDP does not handle packet loss, so he implements at-least-once for the mobile application. Do you think it is enough? Please give your reasons. (5')

Answer:

不够，还需要实现at-most-once。不然会出现重复计票

2. After that, Bob chooses TCP instead. TCP protocol needs a timer for each segment. He learns that negative acknowledge (NAK) helps in reducing the use of timer. Please explain how NAK works and why sender only needs one timer for the last segment of the stream. (Please give short answers with clear points.) (5')

Answer:

NAK通过发送哪些包没有收到来减少timer的数量。对于最后一个包则需要通过timer来保证一定收到response

3. As Singles' Day comes, the OVS will face huge voting requests from multiple users. Bob uses AIMD (Additive Increase, Multiplicative Decrease) as its congestion control solution. Please explain how AIMD works and how it can ensure fairness? (Please give short answers with clear points.) (5')

Answer:

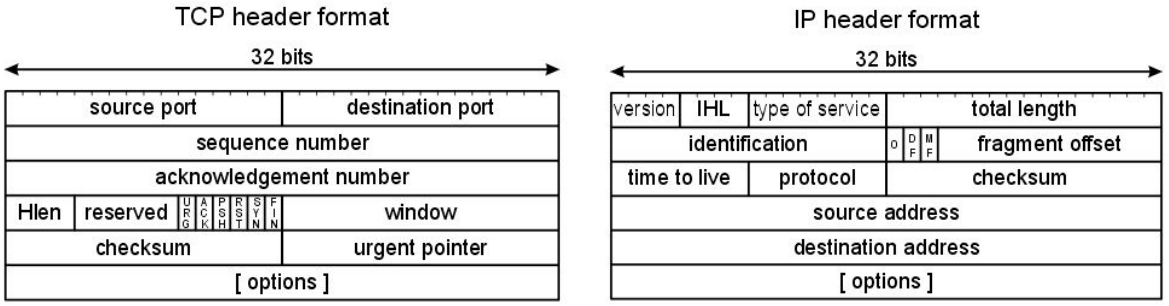
AIMD (Additive Increase, Multiplicative Decrease)。线性增加，倍数下降。保证fairness：当有多个sender的时候，最终这些sender的windows size会收敛到同一个值

4. One day the service crashed since the disks stopped working any more. Some data are lost. Bob was suggested to use RAID. Which type of RAID would you recommend to Bob? Please give your reasons. And, can RAID completely solve the problem of data loss? Why? (5')

Answer:

RAID5，因为voting主要是写请求，RAID5对于写并发支持最好，RAID不能完全解决数据丢失，如果所有磁盘都出现问题，数据依然会丢失

5. As the service scales, Bob buys more servers, which need to exchange data through network. However, using TCP/IP is space-consuming because of extra headers. Below are the packet formats of TCP and IP header.



Bob found that the server machines are directly connected with each other using cables that provide uncorrupted and ordered data transmission (i.e., the content and order of packets are **always** right). However, packets may still get lost due to congestion. Bob wants to design his own protocol based on TCP. Please help Bob optimize TCP/IP and describe at least three aspects. (5')

Ans:

1. 对TCP/IP 两层协议进行合并，减少重复的项
    a. IP层中的两个ip(src/dst)可以不要（在一个集群中，直接用mac地址）
    b. TTL 不需要（不会经过router去修改ttl）
    c. 只需要一个长度
2. 去掉checksum，因为不会corrupted
3. 不能去掉seq/ack，即使不会reorder，也可能会出现丢包

4. 对windows的timeout可以进行优化，当收到的包是1，2，3，5的时候，说明4一定已经丢了，直接要求sender发送4而不是等待timeout

Problem 2: Transaction (35')

A bank wants to implement a distributed user account system. The system records two values for each account: **balance** (which means the money remained in the account) and **withdrawn** (which means the total money withdrawn by the account). Each account is only stored on one machine.

In the system a `coordinator` can issue two kinds of transactions:

1. **`transfer`**`(from_id, to_id, value)`: transfer **value** of money.
2. **`read`**`(account_ids)`: return **balance** and **withdrawn** of multiple accounts.

For simplicity, we do not handle the case when money is not enough to transfer. Suppose there is only one machine which never crashes (coordinator and server are the same machine). Two-phase locking (2PL) is applied to above transactions.

```
# coordinator side:
transfer(from_id, to_id, value):
    ids = sort([from_id, to_id])
    for id in ids:
        server.lock(id)
    <from_b, from_w> = server.read(from_id)
    <to_b, to_w> = server.read(to_id)
    from_b -= value
    from_w += value
    to_b += value
    server.write(from_id, from_b, from_w)
    server.write(to_id, to_b, to_w)
    server.unlock(from_id)
    server.unlock(to_i)

read(account_id):
    server.lock(account_id)
    ret = server[account_id].read(account_id)
    server.unlock(account_id)
    return ret
```

```
# server side:
write(id, b, w):
    accounts[id].balance = b;
    accounts[id].withdrawn = w;
```

```
read(id):
    return <accounts[id].balance, accounts[id].withdrawn>

lock(id):
    lock(accounts[id].lock)
unlock(id):
    unlock(accounts[id].lock)
```

1. Please compare 2-phase locking and OCC (e.g., the cons, the pros, the suitable scenarios, etc.). (5')

2-phase locking is more suitable in high-contention scenarios. It includes expanding phase and shrinking phase.

OCC is more suitable in low-contention conditions. It may abort when contention arouses.

(Other reasonable answer is also acceptable)

2. During transfer, the code calls **`sort()`**. Why? (5')

   To avoid dead lock.

3. For the above implementation, there is one machine that never crashes. Now the bank uses multiple servers for better throughput. It is possible that the two accounts (e.g., **from_id** and **to_id**) reside on different servers. Thus, during a transfer, if one server crashes, the data may be partially updated. How to guarantee the data of the two accounts are updated atomically? (5')
   Use 2-phase commit and log to guarantee both operations on two machine are completed. (Other reasonable answer is also acceptable)

4. Based on the system implemented in question 3, now we need to improve the availability. The system now provides three replicas for each account and each replica.

   Consider the following design: If a transaction needs to modify accounts, it should read *the same data* from at least two replicas, and then write to at least two replicas. If a transaction is read-only, it can read data from only one replica. Is this design serializable? If yes, briefly explain it. If not, please give a counterexample. (5')
   It is not serializable.

   It is not serializable. For example, there are 3 servers A B C and 2 variables x=0 y=0 on 3 servers. The first transaction writes x=1 to AB, writes y=1 to BC. The second read-only transaction read x=1 y=0 from A, which means it is not serializable.

5. What is the relationship between conflict serializable, view serializable and final state serializable? Which one is used most? Why? (5')

Conflict serializable means it can be transferred to serial schedule by swapping con-conflict operations. View serializable means it is view equal to a serial schedule. View equal means Final state serializable means the final result is equal to that of a serial schedule. Conflict serializable is most commonly used because it is easy to detect conflict operations.

6. What categories are following schedules (conflict serializable, view serializable, final state serializable, none)? (6’)
    a. S1: R1(X) R1(Y) R2(X) R2(Y) W2(Y) W1(X)
    b. S2: R1(X) R2(X) R2(Y) W2(Y) R1(Y) W1(X)
    c. S3: W3(Z), R2(X), W2(Y), R1(Z), W3(Y), W1(Y)

    Final state serializable

    Conflict serializable

    Conflict serializable

7. Which category of serializability does the bank account system satisfy? Please give your reasons. (4’)

    Conflict serializable. Because it uses two-phase locking.

Problem 3: Security (16’)

1. Why CFI can defend against ROP attack? What are the limitations of CFI? Do you have some way to mitigate these limitations? (5’)

    CFI ensures that the control data like the return address no to be corrupted. ROP attack is based on hijacking the control flow by overwriting the return address so it can be defended by CFI.

    CFI cannot defend non-control data hijacking.

    Protect the non-control data use the mechanism CFI uses to protect the control data.

2. Isolation is one of the widely used technologies for security. Please give at least three ways for isolation (e.g., different layers or components), and describe their effects. (5’)

    Kernel - User isolation: prevent non-privileged user accessing privileged data.

    Process isolation: prevent one process from writing the data of other processes.

    VM isolation: prevent one VM from accessing the data of another VM.

3. Address Space Layout Randomization (ASLR) is a technique involved in preventing exploitation of memory corruption vulnerabilities. Kernel uses kASLR to prevent attacker to know the exactly locations of kernel objects. Every time the system reboots, the base of kernel's .text segment will be loaded to a different location.

   Consider the following kernel code, if you are an attacker, please bypass the kASLR (e.g., get the address of function **event_ioctl**). Please describe your approach, write your code if necessary, and give a patch to fix the problem. (6')

   event_ioctl(user_event, EVENT_INIT);

   event_ioctl(user_event, EVENT_INFO);

   The address of the function handler will be copied to user_event which can be read in user space.

   As the offset of other kernel objects' address to the function handler are fixed even KASLR is enabled. So the addresses of all other kernel objects can be calculated from the address of the function handler. KASLR is bypassed.

```
struct event {
    unsigned type;
    int (*handler)(void);
};
```

```
/* code in kernel */

struct event cse_event;
int handler() {...};

int event_ioctl (struct event *user_event, int cmd) {
   unsigned size = sizeof(struct event);
   switch (cmd) {
       case EVENT_INIT:
           if (copy_from_user(&cse_event, user_event, size))
              return -1;
           cse_event.handler = handler;
           break;
       case EVENT_INFO:
           copy_to_user(user_event, &cse_event, size);
           break;
       case EVENT_HANDLING:
           return cse_event.handler();
       default:
           return -1;
   }
```

```
   return 0;
}
```

```
event_ioctl is the only api that user program could access.

// copy kernel data of size s in buffer `from` to a user buffer `dst`.
copy_to_user(void *dst, void *from, unsigned s); //success - return 0

// copy user data of size s in buffer `from` to a kernel buffer `dst`.
copy_from_user(void *dst, void *from, unsigned size);
//success - return 0
```

Problem 4: VR & Paxos (24')

I. VR protocol (10')

Assume we have 5 machines. Single-way network latency is 1ms. All machines could learn the network partition and other machines' death immediately after it happens. Slaves will start ViewChange immediately when its connection to primary is lost. If not mentioned, the machines and network will work as specified in protocol.

1. Given the following network events (some but not all)

   Start at view 1, primary received **OP1** at t=0, **OP1** commit, S4 & S5 partitioned from others at t=3, primary received **OP2** at t=4, S3's response for **OP2** lost, primary received **OP3** at t=6, all **Prepare** for **OP3** lost, S1 & S2 dead at t=8

   1) Will **OP2** eventually commit after network heals and system recovers? Why? (4')

2. After 1., S4 dead at t=9, network healed at t=10, S2 recovered at t=10, S5 retried its **ViewChange** at t=10
   1) What will the viewstamp eventually be? (2')
   2) What's the final state of all machines? Please show the curV, lastV and the log of each machine. (2')
   3) What's the minimal commit latency of each **OP**? (2')

II. Paxos (8')

Assume we have 3 machines, machine A and B are proposers, and machine A, B, C are all acceptors.

The single-way network latency between proposers and acceptor is **1ms** and it is stable. After receiving the response of a **prepare** request, if no safe value is determined in previous ballot, it will take the proposer **10ms** to calculate the value it wants to propose.

We encode the machine id at bit 0 of ballot number, so the ballot number is globally unique. This means proposer A will propose ballot 0, 2, 4, … And proposer B will propose ballot 1, 3, 5, …

If an accept is ignored, the acceptor will still send a response to the proposer, and the proposer will start another ballot with a higher ballot number immediately.

1. Now assume proposer A starts to propose value 1 at t=0, and proposer B starts to propose value 2 at t=7ms, how will the system behave? Please explain the phenomenon in detail. (4')

   C receive prepare 0 from A at t=1ms

   A receive prepare OK at t=2ms

   C receive prepare 1 from B at t=8ms

   B receive prepare 1 OK at t=9ms

   A send accept 0 at t=12ms

   C receive and reject accept 0 at t=13ms

   A send prepare 2 at t=14ms

   B send accept 1 at t=19ms

   C receive and reject accept 1 at t=20ms

   B send prepare 3 at t=21ms

   …

   The system will be stuck and fail to agree on any value.

2. How to avoid the problem in best effort? (4')

   Add random back-off in proposer between two proposals

III. VR vs Paxos (6')

Assume we have a system start at following state:

S1	OP1	OP2	OP3	OP4
	OP1	OP5	OP6	
…	OP1	OP5		
	OP1			
S5	OP1			

After some operations, the system goes to the following state:

S1	OP1	OP5	OP6	OP4
	OP1	OP5	OP6	
…	OP1	OP5		
	OP1	OP7	OP8	

S5 OP1 OP7 OP8

1. Is this a paxos system or a VR system? (2’)

   Paxos system

   Because OP3 on S1 is overwriten by OP6 but OP4 is not.

2. If the clients won’t send new requests, which OPs will eventually and **ALWAYS** commit? (4’)

   NOTE: There won’t be more than 2 dead machines at any time.

   OP1, OP4, OP5

我承诺，我将严格遵守考试纪律。

承诺人：_____

题号									
得分									
批阅人(流水阅卷教师签名处)									