COL733 Major_Exams

Viraj Agashe

TOTAL POINTS

38 / 50

QUESTION 1

Virtualization 4 pts

1.1 Q 1.1 **1 / 1**

+ 0 pts Incorrect

✓ **+ 1 pts** *Correct*

1.2 Q 1.2 **1 / 1**

✓ **- 0 pts** *Correct*

- 1 pts Incorrect

1.3 Q 1.3 **1 / 1**

+ 0 pts Incorrect

✓ **+ 1 pts** *Correct*

1.4 Q 1.4 **1 / 1**

+ 0 pts Incorrect

✓ **+ 1 pts** *Correct*

QUESTION 2

Dynamo and Dotted Version vectors 4 pts

2.1 Q 2.1 **1.5 / 1.5**

✓ **+ 0.75 pts** *Under Network partition choose between availability and Consistency.*

✓ **+ 0.75 pts** *Else choose between low latency and consistency.*

+ 0 pts Incorrect

2.2 Q 2.2 **0.5 / 2.5**

✓ **+ 0.25 pts** *(a) v2 @ S1:1, S2: 0*

+ 0.5 pts (b) v3 @ S1: 0, S2: 1; v4 @ S1: 1, S2: 0.2

+ 0.25 pts (c) v3@S1:0, S2:1

+ 0.25 pts (d) same as (b)

+ 0.5 pts (e) v4 @ S1: 1, S2: 0.2, v5 @ S1:0, S2: 1.3

✓ **+ 0.25 pts** *(f) same as (d)*

+ 0.5 pts (g) v5@S1: 1, S2: 1.3; v6@S1: 1, S2: 2.4

+ 0 pts Incorrect

QUESTION 3

Spanner TrueTime 11 pts

3.1 Q 3.1 **0 / 1**

- 0 pts Correct

✓ **- 1 pts** *Incorrect*

3.2 Q 3.2 **1 / 2.5**

+ 0 pts Incorrect

✓ **+ 1 pts** *Slower*

+ 1.5 pts If reads are happening at TT.earliest(), servers servicing RW transactions will have safe time stuck for longer time

+ 1.5 pts Or if reads are happening at TT.latest(), we will have to wait for longer

3.3 Q 3.3 **2.5 / 2.5**

+ 0 pts Incorrect

✓ **+ 1 pts** *Unchanged*

✓ **+ 1.5 pts** *Safe time must have passed one day old timestamp*

3.4 Q 3.4 **2.5 / 2.5**

+ 0 pts Incorrect

✓ **+ 1 pts** *Unchanged*

✓ **+ 1.5 pts** *Transaction participants can immediately acquire write locks*

3.5 Q 3.5 **0 / 2.5**

✓ **+ 0 pts** *Incorrect*

+ 1 pts Slower

+ 1.5 pts Conflicting transaction will hold write locks for longer

QUESTION 4

CRDTs 11 pts

4.1 Q 4.1 **3 / 3**

+ 0 pts Incorrect

✓ **+ 1.5 pts** *Describes how update works*

✓ **+ 1.5 pts** *Describes how merge works*

4.2 Q 4.2 **3 / 3**

+ 0 pts Incorrect

✓ **+ 3 pts** *Both update and merge are monotonic on semilattice*

+ 3 pts All updates and merges are commutative

4.3 Q 4.3 **2 / 2**

+ 0 pts Incorrect

✓ **+ 2 pts** *Correctly shows a non-linearisable history*

4.4 Q 4.4 **3 / 3**

+ 0 pts Incorrect

✓ **+ 1.5 pts** *Maintain second CRDT to remember busy time*

✓ **+ 1.5 pts** *.get returns intervals that are in original CRDT but not in busy CRDT*

QUESTION 5

Distributed Transactions 4 pts

5.1 Q 5.1 **1 / 1**

+ 0 pts Incorrect

✓ **+ 1 pts** *Correct*

5.2 Q 5.2 **3 / 3**

+ 0 pts Incorrect

✓ **+ 1.5 pts** *Co-ordinator might have acknowledged T1 to client after hearing prepare yes*

✓ **+ 1.5 pts** *Participant might serve another conflicting transaction T2 after reboot*

QUESTION 6

6 Chain Replication **0 / 5**

✓ **+ 0 pts** *Incorrect*

+ 3 pts Create five different chains with 5 different tail servers, assign each key out of 1M total keys to a different chain

+ 2 pts Correctly argues about linearizability of new design

+ 0 pts Tail is still busier than other servers

+ 0 pts Design breaks linearizability

+ 1 pts Design reduces tail load but still maintains multiple versions/tail load is still higher than other servers

+ 1 pts Design slows down writes by locking

QUESTION 7

7 Zookeeper **4 / 4**

+ 0 pts Incorrect

✓ **+ 2 pts** *Correct history with an older read following a read that was not possible in original zookeeper*

✓ **+ 2 pts** *Correct explanation. Failover between two reads.*

+ 0 pts History also possible in original zookeeper

+ 0 pts History not possible with the modified zookeeper

QUESTION 8

Raft 7 pts

8.1 Q 8.1 **2 / 2**

+ 0 pts Incorrect

✓ **+ 1 pts** *Upholds election safety*

✓ **+ 1 pts** *Each server only votes once*

8.2 Q 8.2 **3 / 3**

+ 0 pts Incorrect

✓ **+ 1 pts** *Need to randomise timeouts*

✓ **+ 2 pts** *Correctly shows a pathological sequence of events*

8.3 Q 8.3 **2 / 2**

+ 0 pts Incorrect

✓ **+ 1 pts** *Takes longer*

✓ **+ 1 pts** *Always need more than 1 candidate*

Name: VIRAJ AGASHE Entry number: 2020CS10527

I do hereby undertake that as a student at IIT Delhi:
(1) I will not give or receive aid in examinations, and
(2) I will do my share and take an active part in seeing to it that others as well as myself uphold the spirit and letter of the Honour Code.

Q1 [4 marks] Virtualization

Q1.1 [1 mark] [True/False] When guest OS changes CR3, it must trap into hypervisor when we are using extended page tables for memory virtualization.

FALSE (for EPT, we use EPTP for the host P.T.)

Q1.2 [1 mark] [Tick correct answer] What is the Popek-Goldberg Theorem's requirement for building a trap-and-emulate hypervisor?

[ ] Privileged instructions ⊆ Sensitive instructions
[ ] Sensitive instructions = Privileged instructions
[✓] Sensitive instructions ⊆ Privileged instructions
[ ] Privileged instructions ≠ Sensitive instructions

Q1.3 [1 mark] [True/False] Original x86 architecture followed Popek-Goldberg Theorem.

FALSE

Q1.4 [1 mark] [True/False] QEMU on x86 runs in ring-0.

FALSE

Q2 [4 marks] Dynamo and Dotted version vectors

Q2.1 [1.5 mark] What is PACELC theorem?

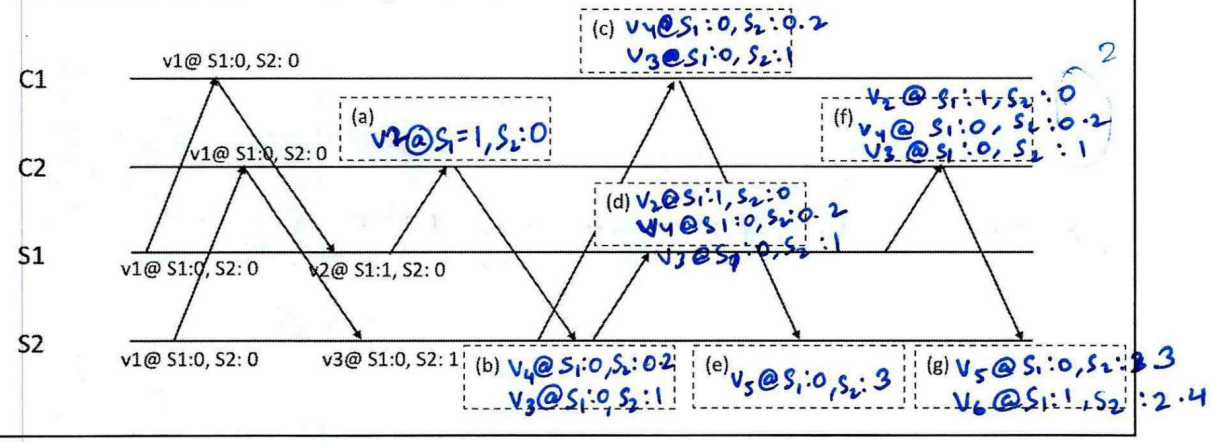
In the presence of partitions, system must choose between availability & consistency, else latency & consistency.

Q2.2 [2.5 marks] Assume that S1 and S2 are two dynamo servers holding values for the key "K". For the following sequence of GET and PUT events on the same key "K", fill the value(s) and their timestamps.

Recall that each server may maintain multiple values of the key and that the GET may return those multiple values. Further, the next PUT is assumed to do "read repair" that

Name: _____ Entry number: _____

merges the read values. Assume that each PUT writes a new value like v4, v5 and so on. Assume that we are maintaining versions as dotted version vectors.



Q3 [11 marks] Spanner TrueTime

Google observed the worst case clock drift of 200 microseconds per second in their servers. Each server synchronises with time master server(s) every 30 seconds to keep TrueTime uncertainty in between 1 and 7 milliseconds.

Let us say that we change this behaviour to make every server synchronise its time with time master server(s) every 10 minutes instead.

Q3.1 [1 mark] What will be the maximum TrueTime uncertainty? Assume that all synchronisation attempts are successful, i.e, no network or server failure.

$= 200 \times 10^{-6} \times 10 \times 60 = 120$ ms.

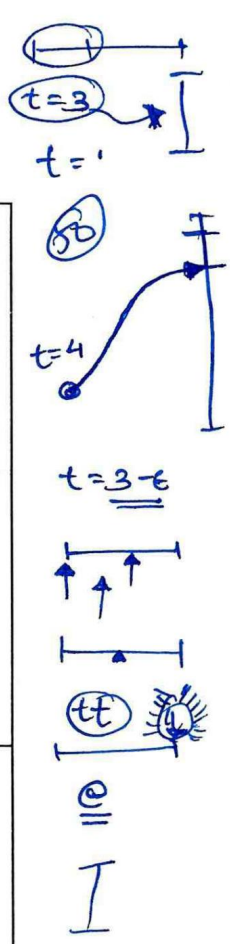
Let us now examine the impact of this change. For each of the following workload, mention if the workload is expected to become slower or will it remain unchanged. Justify your answer.

Q3.2 [2.5 marks] Snapshot read at the latest timestamp.

Slower.

Before satisfying the read, we need to ensure that the timestamp of the read $\leq t_{safe} + \epsilon$. Since ϵ is increased to 120 ms from 6 ms, this will be slower.

Name: VIRAJ AGASHE Entry number: 2020CS10362



Q3.3 [2.5 marks] Snapshot read at a one day old timestamp (for backup and audit purposes).

Same.

Since the worst case $\delta = 120$ ms, for a day-old timestamp, we know that the safetime will be past it safely, and it is outside our range of TrueTime uncertainity.

Q3.4 [2.5 marks] Prepare phase of read-write transaction with no other conflicting transactions.

Same.

Since no conflicting transaction, we can start prepare right away. Prepare phase does not use any notion of time/ safetime.

Q3.5 [2.5 marks] Prepare phase of read-write transaction with other conflicting transactions.

Same.

~~Again since prepare phase does not~~

Even with conflicting transactions, we may ~~not have~~ prepare does not involve δ time notions. (we simply abort other txns)

Name: _____ Entry number: _____

This page is intentionally left blank. You may use it for rough work.

Q4.2. We define the merge operation as.

S_1.merge(S_2) :

for e in S_2 :

if e in S_1 : $S_1[e]$.append($S_2[e]$)

else : S_1.add(e, $S_2[e]$)

We claim that our CRDT forms a partial order under the operations request, get & merge.

We define the operator $\prec$ over two sets S_1, S_2 of our CRDT as,

$S_1 \prec S_2$ iff. $\forall e \in S_1$, ($e \in S_2$ and $S_1[e] \subseteq S_2[e]$)

This is very similiar to the regular set operations.

Further, updates (request) are monotonic and so are merges (can be seen by the defn. above of $\prec$).

$\Rightarrow$ The CRDT is SEC, i.e. will show convergence ↓ To LUB of Lattice.

Name: VIRAJ AGASHE Entry number: 2020CS10562

Q4 [11 marks] CRDTs

Let us say there is a huge lecture hall room in which we are going to conduct all the major examinations. We want to assign TAs to invigilation duties. The only operations we want to support are `request(ta_entry_num, start_time, stop_time)` and `get(ta_entry_num) -> list[start_time, stop_time]`. These operations may go to different servers which might be temporarily partitioned from one another.

Q4.1 [3 marks] Suggest a *state-based* CRDT to implement these operations.

Hint: `start_time` and `stop_time` are in the unit of hours like 1pm to 3pm, so we need not worry about clock drift.

We can keep a simple set-like CRDT.

~~Datatype: {(Entry No, List[start-time, end-time])}~~

We will store a set of tuples of the form (Entry No, List[start-time, stop-time]) ← Tuple

- For a request(e, st, et), we append (st, et) to the list corresponding to e (entry no)
- For a get(e), we simply return the list, i.e. the 2nd element of the tuple of element with entry no. e. If no key, return []

If tuple set with entry no. e does not exist add (e, [(st, et)]) to the set

Q4.2 [3 marks] Justify why your system provides strong eventual consistency.

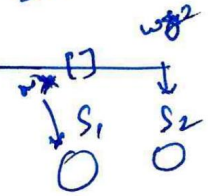
For the merge function, we simply do as follows:

merge(S_1, S_2):
  $S = \{\}$; for all $e \in S_1 \cap S_2$:
    $L = S_1[e] \cup S_2[e]$
    S.add(L)
  for all $e \in S_1 \setminus S_2$:
    $L = S_1[e]$; S.add(L)
  for all $e \in S_2 \setminus S_1$:
    $L = S_2[e]$; S.add(L)
  return S;

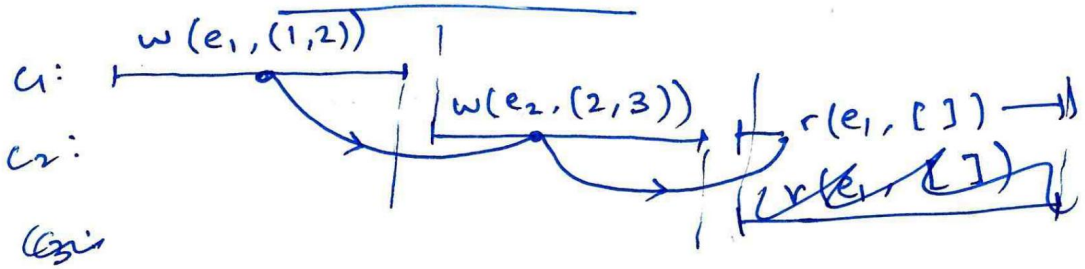
IMPORTANT
Q4.2 done on LEFT PAGE

To prove eventual consistency we just need to define a partial order over the CRDT and prove that

See last page for Q4.2



Q4.3 [2 marks] Give an example of an observable history with your system that is NOT linearizable but exhibits strong eventual consistency.



Clearly Not linearizable, But it could be that both writes went to different servers. Ultimately as proven above, after anti-entropy they will be consistent.

Name: _____ Entry number: _____

Q4.4 [3 marks] Now let us say we want to further support `busy(ta_entry_num, start_time, stop_time)` where the TA can mark themselves as busy. For example, let us say we call `busy(1234, 1pm, 2pm)` and `request(1234, 12:30pm, 3pm)`. After convergence, `get(1234)` returns `[(12:30pm, 1pm), (2pm, 3pm)]`.

We can keep 2 sets; Available sets and busy sets, both having the same semantics as our previous CRDT — request modifies the available set, and busy modifies the busy set

(Similiar to ~~ddd~~ delete sets — to ensure commutativity)

- When the client requests a ~~read~~ get (entry_num), we return to them as,

~~SAMPLE~~

Say S_A: Avl. set, S_B: busy set.

```
L = []
for el in S_A[entry_num]
    for el' in S_B[entry_num]
        temp = temp \ el'
    L.append(temp)
return L
```

Here, we take each tuple (s, e) in S_A and take the diff. with each element (tuple) in S_B (busy set)

↓

Ensures that we get the correct data S_B availability

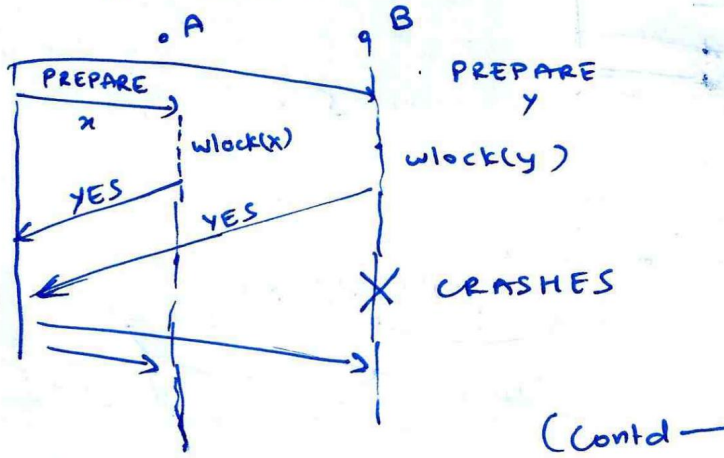
Q5 [4 marks] Distributed transactions

Q5.1 [1 mark] [True/False] When read-write transactions are not conflicting, optimistic concurrency control is expected to be faster than pessimistic concurrency control.

TRUE

Q5.2 [3 mark] In a two-phase commit, why do transaction participants need to write-ahead prepare yes into their disk before responding with a yes to the prepare request from transaction coordinator. Use a concrete example to explain what goes wrong if a participant does not write prepare yes to disk.

Suppose transaction participants (TP) do not write ahead log PREPARE YES. Then,



(Contd — next page)

Name: VIRAJ AGASHE Entry number: 2020CH10527

Q 5.2 (Contd.) This page is intentionally left blank. You may use it for rough work.

Now when B comes up again, it must remember that it was in the middle of a transaction. If it has not logged YES <txid> in the WAL, then when it comes back up, it has no way to remember this, and it could say yes to PREPARE(y) for another transaction T_2.

If it logs, then we can ensure B applies logs before saying YES to any COMMIT or PREPARE messages.

Now, T_1 will keep trying COMMIT(T_1) to B, but now it may be already in another transaction (T_2).

So, A has new value of x, B has older y and cannot be updated ⇒ NON-SERIALIZABILITY.

1

set_union.

Name: _____ Entry number: _____

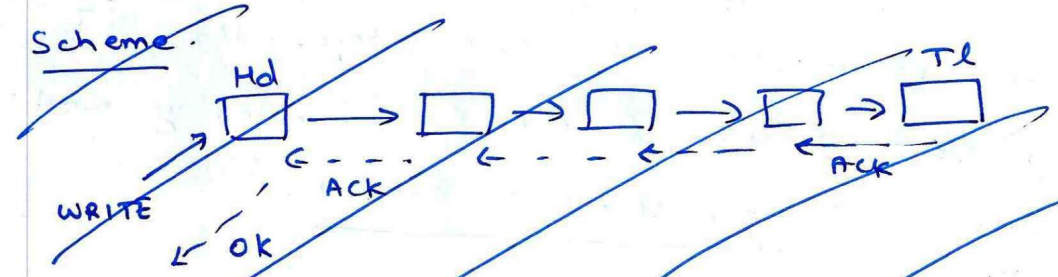
Q6 [5 marks] Chain replication

Let us say that we have 5 servers maintaining a linearizable key-value store with 1 million keys. Our key-value store only supports `get(key)->val` and `put(key, val)` operations. Further assume that each pair of servers has identical network bandwidth and latency.

CRAQ reduces the load on the tail server by letting non-tail servers serve reads. This also improves locality of reads by letting clients read from a closer server. But CRAQ needs to maintain multiple versions of each key.

[5 marks] Let us say we don't care about improving locality and we do not want to maintain multiple versions of each key due to its implementation complexity. Suggest a scheme to distribute read load across the servers while maintaining linearizability and 5x replication of the key-value store.

Scheme.



In this scheme, we keep the same first step, i.e. writes flow through the chain, and ACKs flow back through the chain.

Q7 [4 marks] Zookeeper

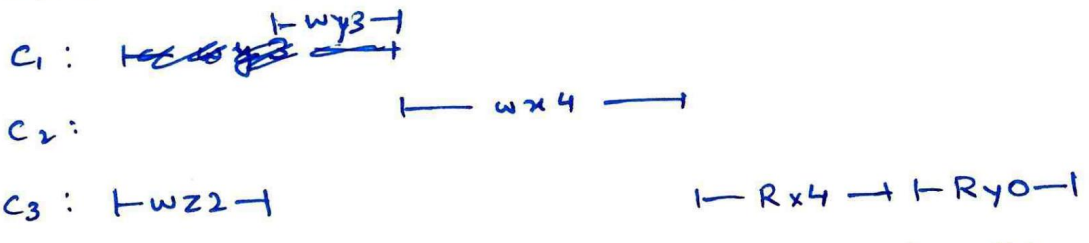
In Zookeeper, every read and write returns a "zxid" which is sent by the client in the next request. The server is allowed to respond only after its commitIndex is greater than the

Name: VIRAJ AGASTE Entry number: 2020CS10527

request's zxid. Let us say we change this behaviour such that only the writes return a zxid; reads do not return a zxid. In each request, the client sends the largest zxid it knows about.

[4 marks] Show a history that can be observed by clients in this modified Zookeeper, but that canNOT be observed in the original Zookeeper. Justify why this history is now observable.

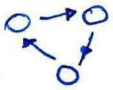
Suppose value of $x=0$, $y=0$ initially. Then,



This cannot be seen in original zookeeper since the read Rx4 would return zxid, ~~of the~~ and since wx4 is committed, wy3 must also be committed. So we would always read ~~0~~y3. In new zookeeper, we only have zxid of our earliest write, wz2. So we if our second read Ry0 goes to a different server, we can read Ry0.

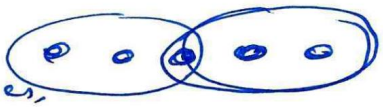
Q8 *[7 marks]* Raft

Let us say we modify Raft as follows: each server votes for the *second* vote request. Hence, a candidate may not vote for itself as soon as it becomes a candidate. If the candidate had already received a requestVote RPC, it will vote for itself. Followers cast their vote to the second requestVote RPC that they receive.



Q8.1 [2 marks] Does this change still uphold Election Safety, i.e, at most one leader can be elected in a given term? Justify your answer.

YES. Same concept as regular RAFT: if there are 2 majorities, they must have a common node (PHP) i.e. Some node voted twice. However, after voting if there is another election timeout, we need to simply increment term no. So vote only once per term ⇒ Election Safety.



Q8.2 [3 marks] Remember that in the original Raft, we could have a pathological sequence of events where a leader never gets elected even though a majority of servers are able to talk to each other. Original Raft solved this by randomising election timeouts. Do we still need to randomise election timeouts with our change to voting mechanism? Justify your answer.

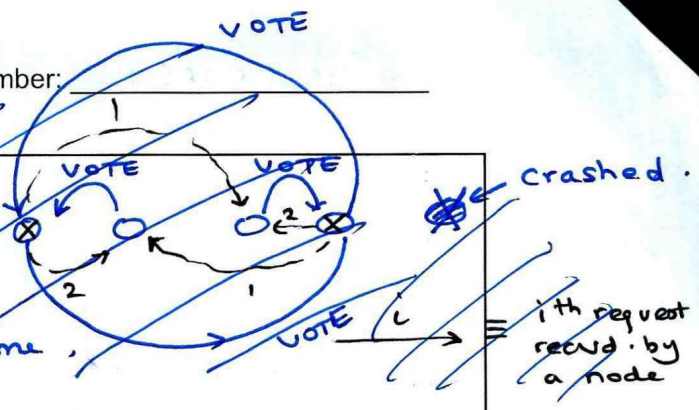
Name: _____ Entry number: _____

Consider this scenario:

⊗ ≡ Up to date nodes

In this case, there is a tie.

If election timeouts are same,



Q8.3 [2 marks] Is this mechanism expected to take longer than the original Raft election mechanism to elect a new leader? Justify your answer.

8.3. Longer.

(In the infinite loop case, obviously)

Even if we use randomized timeouts, before voting for itself, candidates will need to wait to receive a vote request from someone else. So it will take slightly longer ($\sim RTT/2$ longer per election period)

This area is intentionally left blank. You may use it for rough work.

Q8.2 In this figure, ⊗ are the up-to-date nodes.

If election timeout is the same, the scenario can keep repeating in an infinite loop as well.

(Since candidates can still end up voting for themselves as long as there is one more up to date candidate)