A Very Exclusive Club

6. (5 points) You want to choose a way to provide mutual exclusion for the following scenario: You have 2 very high quality, fast computers both connected to the same Ethernet switch. Requests are sent to *both* computers, and only one should act upon the request. When the request arrives, it should be assigned to one or the other computer, which then processes it. Exactly one computer should handle each request.

   What mutual exclusion protocol would you use? Justify your answer in terms of its performance, implementation complexity, and other factors you feel are important. A few sentences will do.

Solution:

The best solution here is probably a token passing scheme. It's not robust to failures without some extra machinery, but our machines are pretty reliable. The latency is close to optimal: Whichever computer is holding the token when a request arrives services the request and then passes the token. (What's cool about this is that the mutex algorithm actually runs *in advance* of the request arriving: other schemes could, of course, be modified to do this as well - but here the engineering is easy.) The number of messages sent is small, only one per request.

7. (6 points) Instead of 2 computers, you now want to build the same system using 20 computers that you found in the "free computers" bin in Wean hall. These computers are cheap, pretty unreliable, and all have varying speeds. Don't worry about a computer crashing once a request was received, but now what mutual exclusion protocol would you use? Justify *numerically* in terms of the number of messages, the latency, reliability, etc., compared to the protocol you picked for the previous part:

Solution:

Eek, we better use something that's robust to failures. And with 20 computers, we probably don't want to risk setting up a linear token passing scheme because the latency will be too high. We'd probably start with a majority-based broadcast scheme based upon the bakery algorithm and see if the traffic is too high. It provides low latency (a few RTTs), it does send a lot of messages (20 per request), but it's robust. If that proves

8. (9 points) A friend who took 15-440 last year proposes a new protocol for you. He wants to get the efficiency of Maekawa's quorum system but with robustness to node failures. He calls it the *Cluster Mutex, Ultra*, and it works like this:

   Arrange the nodes in a grid, just like Maekawa's algorithm. Assume there are an exact power of two number of nodes for simplicity.

   `mutex_acquire`: The requesting node r picks *two* rows and *two* columns to send its requests to. (Recall that Maekawa just sent to one row and one column). It broadcasts a message to the $2\sqrt{N}$ nodes in its rows and the $2\sqrt{N}$ nodes in its columns, saying REQUEST(time, r).

   `recv REQUEST`: If node hasn't granted its VOTE to anyone yet, send VOTE(time, r) back to r. Otherwise it's voted for s, so send back SORRY(time, r, s).

   `recv VOTE`: When a node receives at least $\sqrt{N}$ column votes (from nodes in the columns it picked) and $\sqrt{N}$ row votes (from nodes in the rows it picked), it has the lock and can proceed.

   `mutex_release`: Broadcast RELEASE(r) to all of the row and column nodes in its quorum.

   (a) Explain why or why not this protocol provides each of the following properties of a distributed mutual exclusion protocol. Be succinct; you may refer to known behaviors of other protocols we discussed in class if you wish.

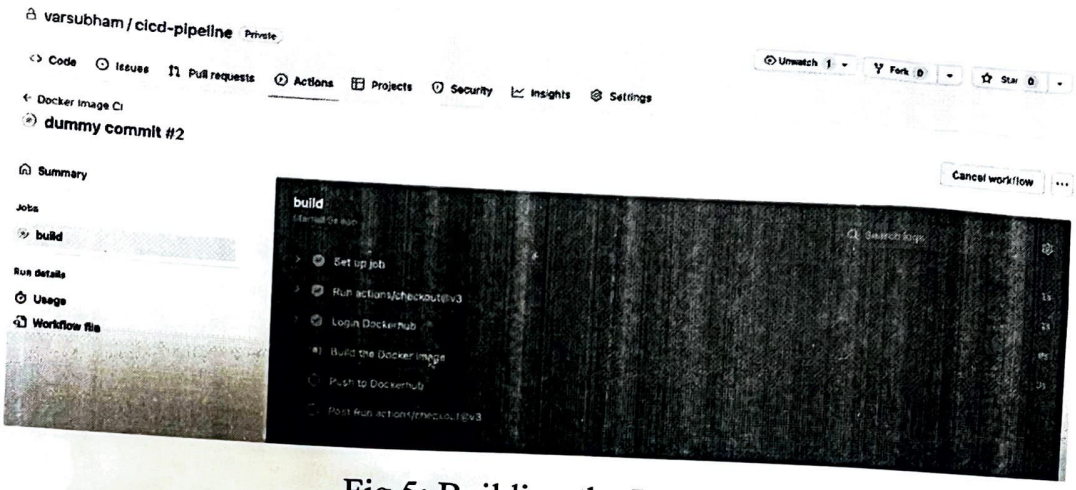

Fig 5: Building the Docker Image

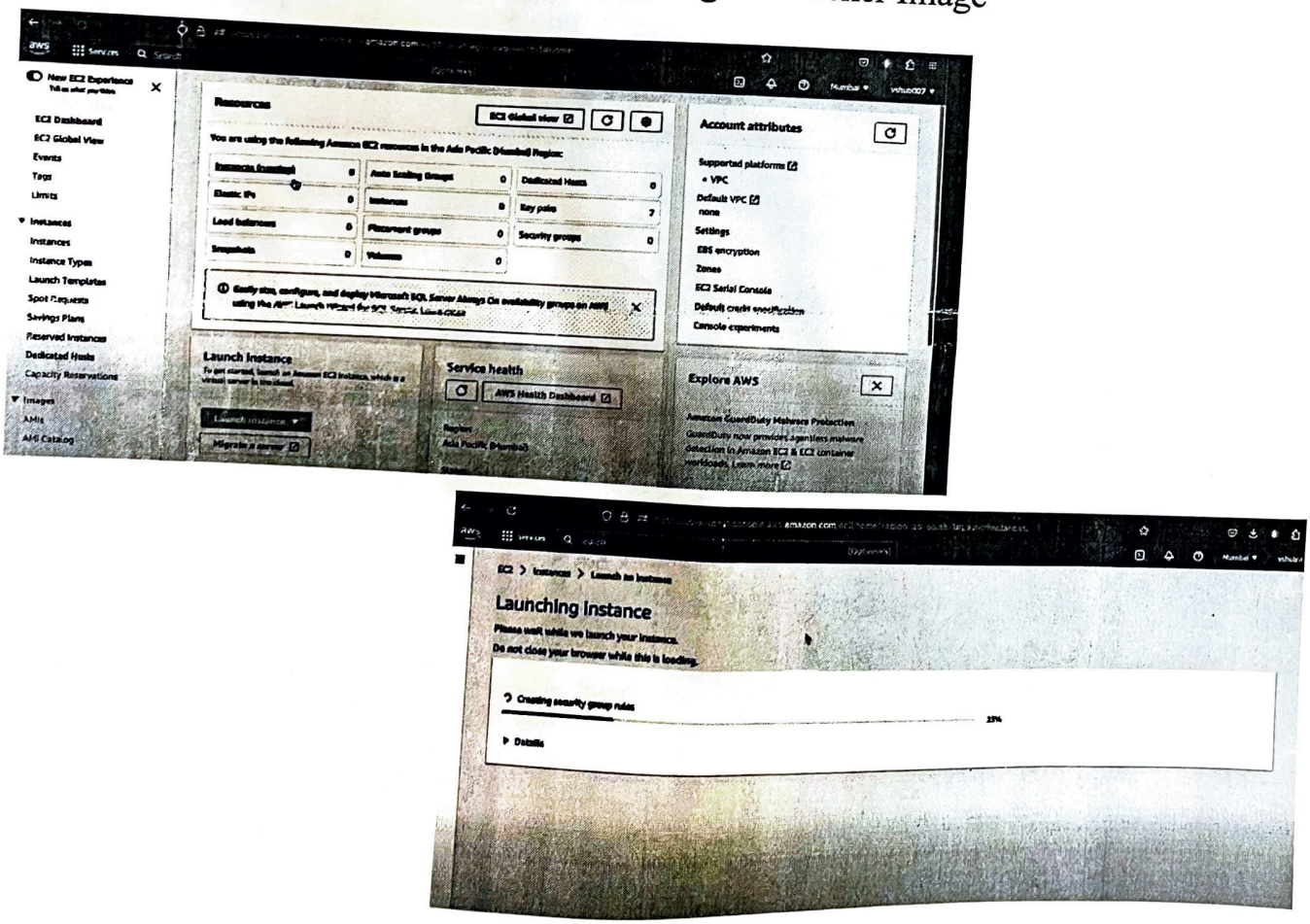

Fig 6: Launching AWS EC2 instance

- Mutual exclusion

Solution: The protocol is broken: It doesn't provide mutual exclusion. Consider what happens in this case where two clients are trying to grab the lock. "o"s represent responses sent to client 1 and "."s represent responses sent to client 2:

o	.	o	.	o	.	o	.
.	o	.	o	.	o	.	o
o	.						
.	o						
o	.						
.	o						
o	.						
.	o						

Both clients receive an "OK" response from $\sqrt{N}$ people in a column and in a row, just as the protocol said – but they have a completely non-overlapping set, and so fail to achieve mutual exclusion.

- Fairness

Solution: The protocol is not fair. Even if it worked. it has the same fairness problem that the basic Maekawa protocol does: there is no lamport timestamping of the requests to ensure that they are sequenced by the time at the requestor.

- Bounded waiting

Solution: The protocol does not guarantee bounded waiting. It's possible that a node could have to wait forever because other nodes keep winning the lock. (There are no queues for keeping track of who wanted the lock, and so everyone has to contend when the lock is released. This contention is *likely* to be bounded in practice, but in theory, it's not.)

(b) (8 points) There is a very bad problem with this protocol that is specific to the *changes* from the basic Maekawa protocol (e.g., not one of the things you may have mentioned above that are known problems with Maekawa). Propose a fix by describing the changed parts of the protocol, and sketch a brief explanation (think "proof" but not formal) of why your revised protocol works.

Solution: Instead of requiring $\sqrt{N}$ from any row and any column, require $\sqrt{N}$ responses *all in the same row* (and all in the same column). This works because it turns into running two copies of the Maekawa algorithm using a permuted set of rendezvous nodes. Either one alone is enough to provide mutual exclusion properly. The protocol is less robust than your friend hoped, however: it can only handle a few failures.

A Very Exclusive Club

6. (5 points) You want to choose a way to provide mutual exclusion for the following scenario: You have 2 very high quality, fast computers both connected to the same Ethernet switch. Requests are sent to *both* computers, and only one should act upon the request. When the request arrives, it should be assigned to one or the other computer, which then processes it. Exactly one computer should handle each request.

   What mutual exclusion protocol would you use? Justify your answer in terms of its performance, implementation complexity, and other factors you feel are important. A few sentences will do.

Solution:

The best solution here is probably a token passing scheme. It's not robust to failures without some extra machinery, but our machines are pretty reliable. The latency is close to optimal: When a computer is holding the token when a request arrives, it services the request and then passes the token. What's cool about this is that the mutex algorithm actually runs *in advance* of the request arriving. (Other schemes could, of course, be modified to as well – but here the engineering is easy.) The number of messages sent is small, only one per request.

7. (5 points) Instead of 2 computers, you now want to build the same system using 20 computers that you found in the "free computers" bin in Wean hall. These computers are cheap, pretty unreliable, and all have varying speeds. Don't worry about a computer crashing once a request was received, but now what mutual exclusion protocol would you use? Justify *numerically* in terms of the number of messages, the latency, reliability, etc., compared to the protocol you picked for the previous part:

Solution:

Ok, we better use something that's robust to failures. And with 20 computers, we probably don't mind the traffic is too high. It provides low latency (a few RTTs), it does send a lot of messages (20 per request), but it's robust. If that proves

8. (9 points) Robert Tappan 15-440 last year proposes a new protocol for mutual exclusion. He wants to get the efficiency of Mackawa's quorum system but with robustness to node failures. He calls it the *Cluster*

va's

two

plum

g R

E to

s).

col

ed),

l of

rovi

mutual exclusion protocol. Be succinct; you may refer to known behaviors of other protocols we discussed in class if you wish.