Zachary Siciliani (260845463)

Eric Pelletier (260895863)

COMP 512

November 4th, 2022

Programming Assignment 2: Performance Analysis

First, we analyzed how the rate of moves being accepted is impacted by the number of players present in the game. Our hypothesis was that more players in the game would like to a smaller percentage of moves being accepted. Our data is shown in Figure 1.

Number of Players	Acceptance Rate							
	Player 1	Player 2	Player 3	Player 4	Player 5	Player 6	Player 7	Player 8
2	0.909091	1						
3	0.769231	0.909091	1					
5	0.120482	0.15873	0.196078	0.666667	0.5			
6	0.078125	0.121951	0.151515	0.263158	0.344828	0.555556		
8	0.055249	0.073529	0.098039	0.10101	0.09901	0.16129	0.263158	0.344828

Figure 1: Rate of moves being accepted vs number of players

We found that our results essentially matched what we expected. To obtain the data shown in Figure 1, we set the interval to 500 and the maximum number of moves to 10. We tried using multiple different intervals but found that changing the interval had relatively little impact on the data that was obtained. We believe that this is because the game is behaving properly and is never submitting a second move before the first one has been accepted, even if it takes multiple tries for the first one to be accepted. Because of this, no matter what the interval is, each process is only trying to push at most one move. If this system behaved improperly and the process tried to propose a new move before the previous one was fully handled, then a smaller interval would quickly lead to increasing moves being proposed simultaneously and a snowballing effect where more moves are being rejected; however, that was not what occurred in this case.

At a fixed interval, it is clear that increasing the number of players had a very significant effect on the acceptance rate. Acceptance rate was measured by tracking and logging each time a process starts a new proposal with a ballotId. We then did 10 (the number of accepted moves) divided by the number of proposals by that process to determine its acceptance rate. When there are only two processes proposing moves, it is quite plausible for a very high percentage of moves to be accepted. Even in allegedly simultaneous processes, there will likely be a small gap in time between messages being sent. Collisions therefore occur less frequently, and more proposals are likely to lead to acceptances. With more players, on the other hand, this is less likely. Once 8 processes are running, it is likely that multiple of them are at any given stage at the same time. One of those will then knock down the other, leading to a much lower acceptance rate.

The next question to address very heavily ties into the first one. We analyzed whether there is a given process that is more favored by our Paxos algorithm than the others. Our data in Figure 1

shows very strongly that there is; the greater the player number, the higher the acceptance rate. This is fully logical based on the way that we implemented a unique ballotId. We tracked a ballot sequence, which would always be a multiple of 10. Then, each process would add their player number to that ballotSequence to create a ballotId. For example, if player 5 has previously received ballotId 7, it will set its ballotSequence to 10 and add 5 to get a new ballotId of 15. It uses the ballotSequence to ensure the proposal is of a higher value, and its own player number to ensure the proposal is of a unique value. However, one side effect of this algorithm is that larger player numbers will lead to larger ballotIds for the same ballotSequence, and if both those players propose at the same time, one of them will be favored over the other. The occurrence of this phenomenon is very visible in our data, often with a sharp increase in success rate for the final player. If a fair game was among our priorities, we would have needed a different, more complex approach to ensuring uniqueness of ballotIds while also not favoring one process over another.

Finally, we analyzed the average runtime per move as the interval changed. One move in this analysis is defined as an entire round, where all processes play one move. Results can be seen in Figures 2 and 3.

Interval	Total Runtime (s)	Runtime per move (s)
100	15.48482	1.548482
250	16.609809	1.6609809
500	13.92378	1.392378
1000	15.360875	1.5360875
2000	20.735487	2.0735487
4000	40.810392	4.0810392
8000	80.710738	8.0710738

Figure 2: Table of runtime vs interval

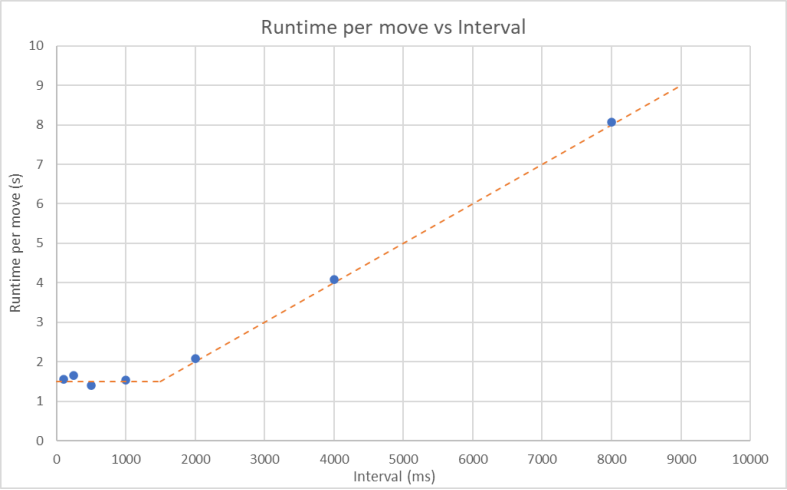

Figure 3: Scatter plot of runtime vs interval. The blue dots represent real data points, while the orange dotted line is a manually determined trendline.

As you can see in Figures 2 and 3, the average time per move decreases as the interval decreases. This continues until around an interval of 1500 ms, at which point the runtime stabilizes. While the runtime is decreasing, the limiting factor is the interval. The runtime per move can never be shorter than the interval, as the program will wait until that time has completed before attempting to deliver the next move. However, eventually the interval gets to be too short and it is no longer significant. The average turn will take longer than the interval, and the next move will be submitted as soon as the previous one is processed. The runtime per turn therefore stabilizes at around 1500 ms and stays there as the interval continues to approach 0. The data in Figures 2 and 3 is taken from a 2-player game. When repeating this experiment with more processes, the

results are similar. The pattern will be the same, but the value at which the runtime becomes the limiting factor increases with the number of players.