COMPSCI 690QC: Quantum Communication Spring 2025

Final Project

Shymon Samsel

https://github.com/ssamsel/CS690QC-Final/

1 Introduction

In my project, I model a quantum network in which peer nodes are connected to a centralized node, the switch, which is responsible for distributing entanglement amongst the peers. I implement the simulation in Python using only basic scientific Python libraries and my own custom simulation framework. The switch generates entanglement between itself and the peers according to the Barret-Kok protocol. In order to entangle two peers, the switch must first generate entanglement between itself and the peers it wants to entangle, and then perform a Bell state measurement, much like a first generation repeater. Each peer submits request to become entangled with another peer to the switch, and the switch must determine the following:

- The order in which to fulfil requests
- When to generate entanglement between itself and a peer
- When/if to discard an entanglement between itself and a peer

In practice, quantum memories are subject to noise, so the entangled Bell pairs will degrade over time. I implement different algorithms that manage the above factors, and evaluate how they perform under varying network conditions. The key performance metrics I will analyze are:

- Fidelity of swapped Bell state
- Sojurn Time (i.e. latency)
- Throughput
- Relative Throughput (proportion of fulfilled requests to total number of requests seen in the system)

I will evaluate performance of different algorithms on varying network conditions such as:

- Link distance
- Request rates
- Symmetrical vs Asymmetrical networks (varied link distances and/or request rates)

2 Model and Assumptions

2.1 Fixed Parameters of the Physical System

- All links use the same type of fiber with $L_{att} = 22$ km and speed of light $c = 2 \times 10^8$ m/s
- Heralding stations are located exactly in the middle of each link.
- Success probability of an entanglement attempt is $\frac{\eta_i^2}{2}$ and $\eta_i = \exp(-\frac{L_i}{2L_{att}})$
- Each attempt takes $2\tau_i$ seconds, where τ_i is the propagation delay derived from L_i and c

2.2 Switch Capabilities

- 2 communication qubits totals, allowing for only 1 BSM/request served at a time (the simulation framework supports 1 communication per peer however).
- BSMs and quantum gates are perfect.
- Each BSM takes negligible time along with the algorithm's decision making steps.

2.3 Request Assumptions

- Request rates are predetermined and do not change over time.
- Requests from peer $i \rightsquigarrow j$ and considered the same as requests from $j \rightsquigarrow i$
- Requests are served in FIFO order within their class (but I only test global FIFO order algorithms).

2.4 Probabilistic Assumptions

- Inter-peer entanglement requests are Poisson processes
- Each Barret-Kok run is a Bernoulli process

2.5 Noise Model

- EPR pairs are time-evolved werner states: $\rho(t) = e^{-\Gamma t}|\Phi^+\rangle\langle\Phi^+| + (1 - e^{-\Gamma t})\frac{I_2}{2}$
- $\Gamma = \frac{1}{T_1} = \frac{1}{5ms} = .2$ (seems to by typical for NV centers from my understanding).

3 Implementation

Resource management algorithms and entanglement distribution algorithms interoperate, so they are combined into one algorithm implemented as a subclass of *Simulation*. Each simulation algorithm implements three methods: *matching_decision*, *entanglement_decision*, and *idle_decision*, which are invoked looping in that order until the time exceeds the max interval specified for the simulation. Since request arrivals are Poisson processes, the simulation operates in an approximation of continuous time, limited by floating point accuracy. At a high level, *matching_decision* decides which set of peers to entangle swap, *entanglement_decision* manages resources (e.g. entanglement generation, discarding past entanglement, etc.), and *idle_decision* handles moving forwards in time. The behavior of each method is not strict and I chose to implement the simulations this way to allow for reuse of code between different algorithms along with flexibility of algorithm design.

3.1 Approximating Continuous Time

Let T_{sim} be the current time at an instant in the simulation.

3.1.1 Entanglement Generation

Entanglement generation attempts are Bernoulli processes which means that sampling from the geometric distribution simulates the number of attempts before entanglement is generated. When the algorithm decides it wants to become entangled with peer i, it samples $T_{entgl}(i) = T_{sim} + 2\tau_i N$, where $N \sim$ Geometric$(\frac{\eta_i^2}{2})$. It stores the sampled $T_{entgl}(i)$, and does not consider itself entangled with the peer until the after $T_{engl}(i)$, which is approximated by $T_{sim} \geq T_{entgl}(i)$. $T_{engtl}(i)$ cannot be resampled until $T_{sim} \geq T_{entgl}(i)$ and the algorithm is restricted from using any $T_{engtl}(i)$ in any decision it makes, as a real implementation cannot see into the future, nor alter it.

3.1.2 Request Arrivals

Requests are considered symmetrical, which means we have request rates $\lambda_{i \leftrightsquigarrow j} = \lambda_{i \rightsquigarrow j} + \lambda_{j \rightsquigarrow i}$ by the additive property of Poisson processes. Each loop of the algorithm involves advancing forwards in time by some value, let that value be T_{adv}. For each request class k, $A_k \sim$ Poisson$(\lambda_k T_{adv})$ is sampled. Then to determine the arrivals times of each of the A_k requests, A_k samples are drawn uniformly in the interval $[0, T_{adv}]$, since arrivals are uniformly distributed in fixed intervals in Poisson processes. Each of these samples is incremented by T_{sim}, sorted, and placed into the queue corresponding to request class k. The algorithm does not "see" requests that have arrival times greater than T_{sim}.

3.1.3 Idling

The *idle_decison* method determines T_{adv}. The algorithm can idle until the next entanglement succeeds, until all entanglements succeed, a fixed amount of time, or until the next request arrives. If the algorithm's determination of T_{adv} computes to 0, the algorithm will idle until the next request arrives. If there are no requests in any queue, the simulation samples $T_{adv} \sim$ Exponential $\left(\frac{1}{\sum_k \lambda_k}\right)$, since Poisson interevent times exponentially distributed. It is possible that no requests will arrive in this sampled T_{adv}, but since exponential distributions are memoryless, such an event does not affect the distribution.

4 Results and Evaluation

4.1 Algorithms Tested

I tested the following algorithms and variants:

4.1.1 ParallelOnDemand

When a request is at the head of the queue, the switch starts attempting entanglement between relevant peers in parallel and makes the BSM immediately once both EPR pairs are created.

4.1.2 SequentialOnDemand

This algorithm is the same as *ParallelOnDemand*, except that it waits for one entanglement to succeed before starting to attempt to entangle the other.

4.1.3 QoS Variants

These algorithms maintaining the entanglement scheduling policy of their parents, but before making a BSM, the algorithm calculates the would-be post swap fidelity. If the fidelity is below a certain threshold (.8 in my tests) the bell state(s) are discarded and entanglement generation must rescheduled according to the parent algorithm. This provides a fidelity quality of service guarantee, hence the name.

4.1.4 SmartOnDemand

In a network where link lengths differ, entanglement times differ. If the times to entanglements differ a lot, it means one EPR pair will be decaying for a relatively long period while the system is waiting on the other EPR pair to be created. In these cases, fidelity will generally be higher if entanglement is done sequentially, starting with the longer link length. However, if the network also has some similar link lengths, sequential entanglement generation will negatively impact latency and throughput. *SmartOnDemand* attempts to remedy this by calculating the probability that parallel entanglement yields higher fidelity, and chooses either parallel or sequential entanglement based on which is more likely. The motivation, derivation, and justification of this algorithm are explained in detail in the appendix.

4.2 Results

I tested my algorithms against two network topologies, one with vastly varying link lengths (figure 1) and one with similar link lengths (figure 2). Within each topology, I ran 8 evenly spaced trials of differing arriving request rates chosen to show when and how different algorithms start to fall off in performance. Let $\mathcal{V}$ denote the varied topology and $\mathcal{S}$ denote the similar topology. The range for $\mathcal{V}$ is much smaller, which was expected since longer links take longer to entangle, meaning the service times are longer, thus leading to the system becoming overwhelmed sooner.

4.3 Sequential vs Parallel

In all topologies, *Parallel* is able to maintain the best performance in terms of latency and throughput, however performs by far the worst in terms of fidelity in $\mathcal{V}$. This was expected in terms of latency, but the lower fidelity was confusing at first and is what motived me to create *Sequential. Sequential* performs better overall in $\mathcal{V}$, sacrificing a little bit of latency and throughput as the arrival rates increase, but performs worse in every way in $\mathcal{S}$. Once I understood why this was the case (as explained earlier in the overview of *Smart*), I designed the *Smart* algorithm which ended up performing roughly the same as the better of *Sequential* and *Parallel* both topologies.

4.4 QoS Performance vs Smart

I expected that the *QoS* variants would have the best fidelity and worse latency than the standard variants, which is what was I observed. Although *QoS* variants do maintain the best fidelity, it is only a marginal improvement over the standard variants and *Smart*, except for *Parallel* in $\mathcal{V}$. The *QoS* discarding behavior adds a lot of latency to the system in both cases, with them performing much worse in $\mathcal{V}$. Despite maintaining close to the best fidelity in both topologies, *Smart* is also able to provide close to the best latency and throughput. This is the reason I did not bother testing

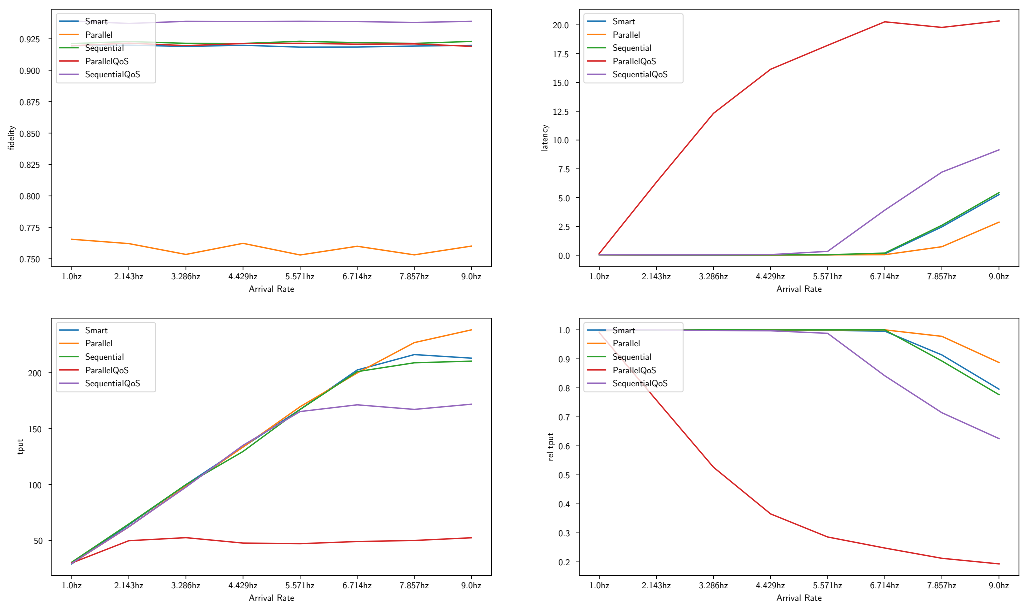

Figure 1: Asymmetrical/Varying Link Lengths: $[20, 11, 15, 50, 30, 10]$ km

a *QoS* variant of *Smart*, as any potential added benefits would likely be marginal and likely make latency/throughput worse.

5 Discussion and Conclusion

Overall, the *Smart* algorithm appears to perform the best. It can handle varying network topologies across different simulations, and presumably handle varying network topologies during the simulation. There is still a vast amount of testing that can be done, but is unfortunately too large of a task to take on for this project. I initially started my project testing switch variations that had 1 communication qubit per peer and could perform parallel BSMs, but was unable to find any particularly interesting results or come up with any novel algorithms. From that testing it was apparent how much more performant such a switch would be, with the maximum arrival rates before system instability being orders of magnitude higher than the single swap algorithms I presented. I left the parallel/multi-qubit algorithms, along with various tests and figures from those algorithms in my codebase.

5.1 Future Work and Improvements

To make my findings more thorough, more network topologies should be tested. A simulation can be run that averages the performances of the algorithms on the average of every possible link length permutation within some fixed set of bounds. I would have tried to implement this however such a test would take an excruciatingly long time to run and is why I decided not to.

As mentioned earlier, parallel BSM capable switches perform substantially better with high request rates, although I did notice they performed worse with low request rates. It would be interesting to see if a dynamic algorithm could be implemented that selects specifically the ideal number of BSMs to make at a given moment in time, and how such and algorithm would compare. If given more time I would want to test how different QoS policies would work, such as not discarding a lower fidelity state if the current observed load is high, for example. It would be interesting to see how this would compare to *Smart* and if there exist QoS policies that would either outperform

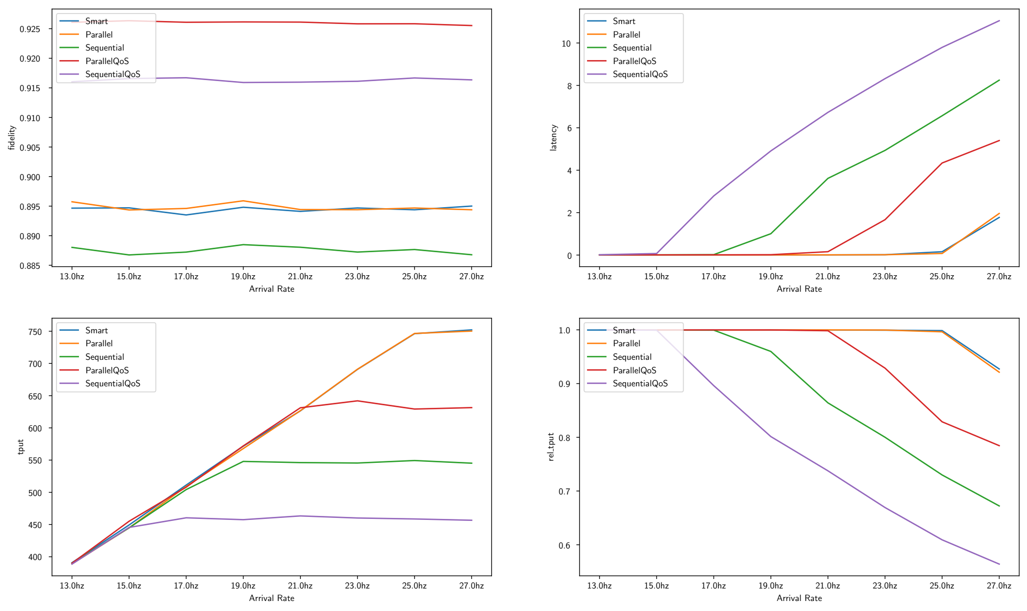

Figure 2: Similar link lengths: $[20, 19, 18, 20, 18, 20]$ km

Smart overall or specifically work well with it.

6 Appendix: Queuing Theory Analysis

These simulations can be modeled as M/G/1 queues with different service times for different classes of requests. Let X_i be the service time of class i, X be the average service time, and I be the current class being serviced.

6.1 M/G/1 Overview

$$\begin{aligned}
\mathbb{E}[X] &= \sum_i \mathbb{E}[X_i] P(I = i) \\
&= \sum_i \mathbb{E}[X_i] \frac{\lambda_i}{\sum_j \lambda_j} \\
&= \frac{\sum_i \lambda_i \, \mathbb{E}[X_i]}{\sum_j \lambda_j}
\end{aligned}$$

Let ρ be the utilization, $\lambda = \sum_j \lambda_j$. The system is stable when $\rho \leq 1$

$$\begin{aligned}
\rho &= \lambda \, \mathbb{E}[X] \\
\lambda \, \mathbb{E}[X] &\leq 1 \\
\mathbb{E}[X] &\leq \frac{1}{\lambda} \\
\frac{\sum_i \lambda_i \, \mathbb{E}[X_i]}{\lambda} &\leq \frac{1}{\lambda} \\
\sum_i \lambda_i \, \mathbb{E}[X_i] &\leq 1
\end{aligned}$$

The lower the expected service time is, the better performance we expect both in terms of fidelity and latency, although they are not directly correlated.

6.2 Sequential Entanglement

Recall that $N_i, N_j \sim$ Geom(p_i), Geom(p_j)

$$\begin{aligned} X_i &= 2\tau_{i_1}N_{i_1} + 2\tau_{i_2}N_{i_2} \\ \mathbb{E}[X_i] &= \mathbb{E}[2\tau_{i_1}N_{i_i} + 2\tau_{i_1}N_{i_2}] \\ &= \frac{2\tau_{i_1}}{p_{i_1}} + \frac{2\tau_{i_2}}{p_{i_2}} \end{aligned}$$

6.3 Parallel Entanglement

$$\begin{aligned} X_i &= \max\{2\tau_{i_1}N_{i_1}, 2\tau_{i_2}N_{i_2}\} \\ \mathbb{E}[X_i] &= 2\tau_{i_1}\,\mathbb{E}[N_{i_1}|\tau_{i_1}N_{i_1} > \tau_{i_2}N_{i_2}]P(\tau_{i_1}N_{i_1} > \tau_{i_2}N_{i_2}) \\ &\quad + 2\tau_{i_2}\,\mathbb{E}[N_{i_2}|\tau_{i_1}N_{i_1} \le \tau_{i_2}N_{i_2}]P(\tau_{i_1}N_{i_1} \le \tau_{i_2}N_{i_2}) \\ &= \frac{2\tau_{i_1}}{p_{i_1}}P(\tau_{i_1}N_{i_1} > \tau_{i_2}N_{i_2}) + \frac{2\tau_{i_2}}{p_{i_2}}P(\tau_{i_1}N_{i_1} \le \tau_{i_2}N_{i_2}) \\ &= \left(\frac{2\tau_{i_1}}{p_{i_1}} - \frac{2\tau_{i_2}}{p_{i_2}}\right)P(\tau_{i_1}N_{i_1} > \tau_{i_2}N_{i_2}) + \frac{2\tau_{i_2}}{p_{i_2}} \\ P(\tau_{i_1}N_{i_1} > \tau_{i_2}N_{i_2}) &= \sum_{n=1}^{\infty} P(\tau_{i_1}N_{i_1} > \tau_{i_2}n|\tau_{i_2}N_{i_2} = \tau_{i_2}n)P(\tau_{i_2}N_{i_2} = \tau_{i_2}n) \\ &= \sum_{n=1}^{\infty} P(N_{i_1} > \frac{\tau_{i_2}}{\tau_{i_1}}n|N_{i_2} = n)P(N_{i_2} = n) \\ &= \sum_{n=1}^{\infty}(1-p_{i_1})^{\left\lfloor n\frac{\tau_{i_2}}{\tau_{i_1}}\right\rfloor}(1-p_{i_2})^{n-1}p_{i_2} \\ &\approx \sum_{n=1}^{50}(1-p_{i_1})^{\left\lfloor n\frac{\tau_{i_2}}{\tau_{i_1}}\right\rfloor}(1-p_{i_2})^{n-1}p_{i_2}, \forall L \le 50 \\ &:= P_{par}(p_{i_1}, p_{i_2}, \tau_{i_1}, \tau_{i_2}) \\ \mathbb{E}[X_i] &\approx \left(\frac{2\tau_{i_1}}{p_{i_1}} - \frac{2\tau_{i_2}}{p_{i_2}}\right)P_{par}(p_{i_1}, p_{i_2}, \tau_{i_1}, \tau_{i_2}) + \frac{2\tau_{i_2}}{p_{i_2}} \end{aligned}$$

6.4 Smart Entanglement

6.4.1 SmartOnDemand Probability Derivation

Let F_i be the time the shorter/faster-in-expectation link takes to entangle in class i, and S_i be the longer/slower-in-expectation link's time in class i

$$X_i = \begin{cases} F_i & \text{sequential} \\ |S_i - F_i| & \text{parallel} \end{cases}$$

$$\begin{aligned} P(F_i > |S_i - F_i|) &= P(F_i^2 > F_i^2 - 2SF_i + S_i^2) = P(2F_i > S_i) \\ &= P(2\tau_{fast}N_{fast} > \tau_{slow}N_{slow}) \\ &= \sum_{n=1}^{\infty} P(2F_i > 2n\tau_{slow}|S_i = 2n\tau_{slow})P(S_i = 2n\tau_{slow}) \\ &= \sum_{n=1}^{\infty} P(F_i > n\tau_{slow})P(S_i = 2n\tau_{slow}) \end{aligned}$$

$$
\begin{aligned}
&= \sum_{n=1}^{\infty}(1-p_{fast})^{\left\lfloor n\frac{\tau_{slow}}{2\tau_{fast}}\right\rfloor}(0-p_{slow})^{n-1}p_{slow} \\
&\approx \sum_{n=1}^{50}(1-p_{fast})^{\left\lfloor n\frac{\tau_{slow}}{2\tau_{fast}}\right\rfloor}(1-p_{slow})^{n-1}p_{slow}, \forall L \leq 50 \\
&:= P_{smt}(p_{fast}, p_{slow}, \tau_{fast}, \tau_{slow})
\end{aligned}
$$

6.4.2 Expected Class Service Time

$$
X_i = \begin{cases} \underset{par}{X_i}, & P(F_i > |S_i - F_i|) > .5 \\ \underset{seq}{X_i}, & P(F_i \leq |S_i - F_i|) \leq .5 \end{cases}
$$

$$
\mathbb{E}[X_i] = \begin{cases} \mathbb{E}_{par}[X_i], & P(F_i > |S_i - F_i|) > .5 \\ \mathbb{E}\, seq[X_i], & P(F_i \leq |S_i - F_i|) \leq .5 \end{cases}
$$

This is dependent on network topology, and I am unsure of a good way to calculate this or approximate it in closed form unfortunately.

6.5 Determining which Algorithm is Statistically Best

As discussed earlier, there are some cases where parallel is better than sequential and vice-versa. From the expected class service time of *SmartOnDemand* we see that it is bounded by whichever entanglement generation strategy performs better on average for a given class. This means *SmartOnDemand* behaves optimally in the context of parallel vs sequential on the class level and thus the global system level. This is consistent with my findings and this analysis is what motived the design/implementation and testing of *SmartOnDemand*.