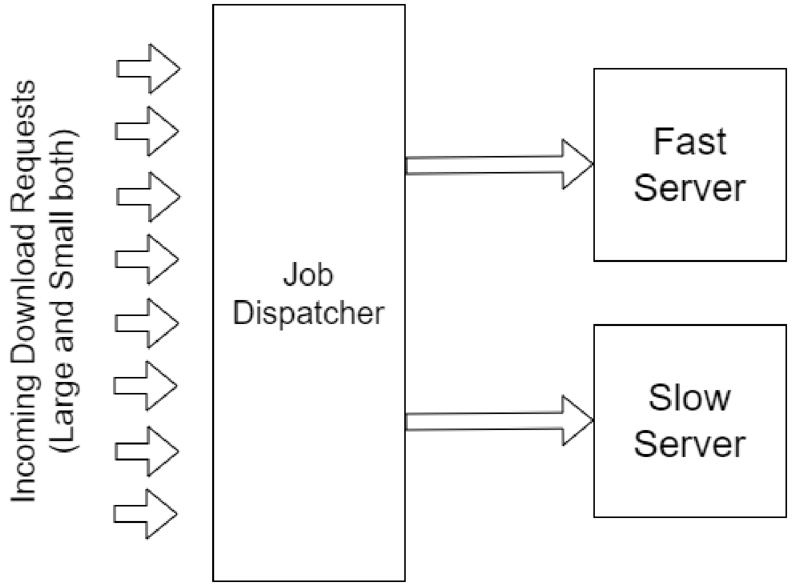
OPTIMAL ROUTING TO PARALLEL SERVERS

BTP-2 PRESENTATION

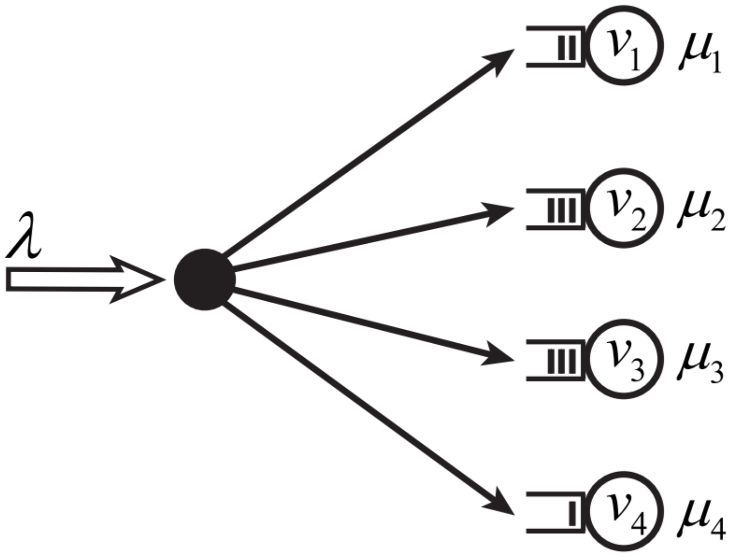
Project Guide: Prof. Rajarshi Roy
Rishikant Kashyap (20EC39028)

Server Optimization Problem



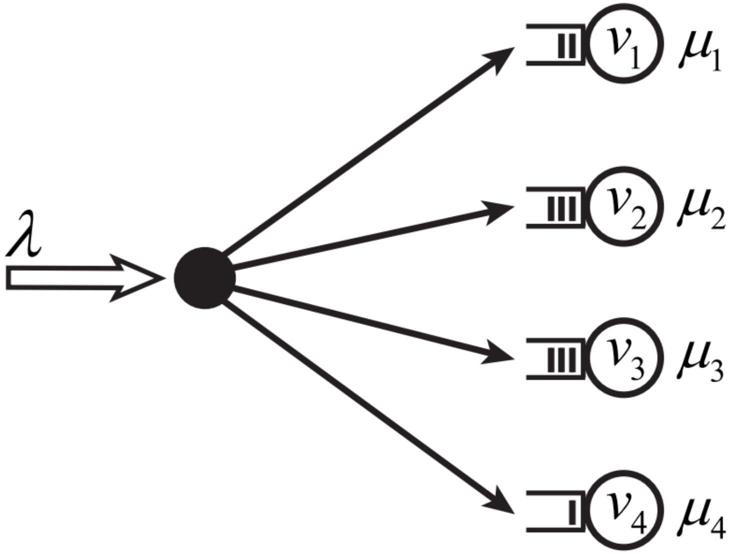
- Let us consider a system of downloading files with
  a. two types of requests: large and small files, and
  b. two types of servers: fast and slow for preprocessing
- We have to minimize the time taken for preprocessing
- Allocating servers based on just availability may lead to large files being allocated to slow server
- Slow server may take a lot of time to process large files resulting in suboptimal results
- Our aim should be to reduce the idleness of the best servers to improve the performance of the system

Optimal Routing Problem



- Let us consider the Optimal Routing Problem for a discrete-time system where a job dispatcher is connected to M parallel servers.
- At each time slot, $a(t)$ jobs arrive at the dispatcher, which is then allotted to the servers.
- The task at each server is managed in a queue that stores incoming jobs.
- Each server s_m has an underlying utility v_m, which is obtained after the server completes a task.

Optimal Routing Problem



- For a sample path ω, under a generic policy π, the utility obtained is given by $\sum_{m=1}^{M} v_m \, C_m^{\pi}(\omega, \mathrm{T})$
- The expected utility of a policy π over the time horizon is given as $U_T(\pi) = \mathbb{E}[\sum_{m=1}^{M} v_m \, C_m^{\pi}(\omega, \mathrm{T})]$
- The regret of a policy π is defined as $R_T(\pi) = sup_{\pi^* \in \prod^*} U_T(\pi^*) - U_T(\pi)$
- Our goal is to design a policy that makes routing decisions to maximize the expected utility and minimize the regret

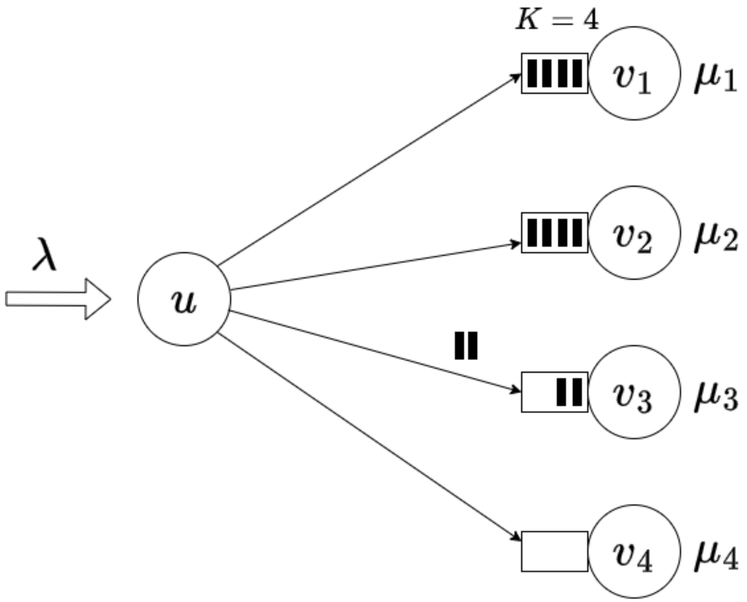
Here, v_m denotes the underlying utility of server s_m

$C_m^{\pi}(\omega, \mathrm{T})$ denotes the job completed by server s_m over the time T

$sup_{\pi^* \in \prod^*} U_T(\pi^*)$ denotes the supremum over all policies in $\prod^*$

The Priority-K Policy

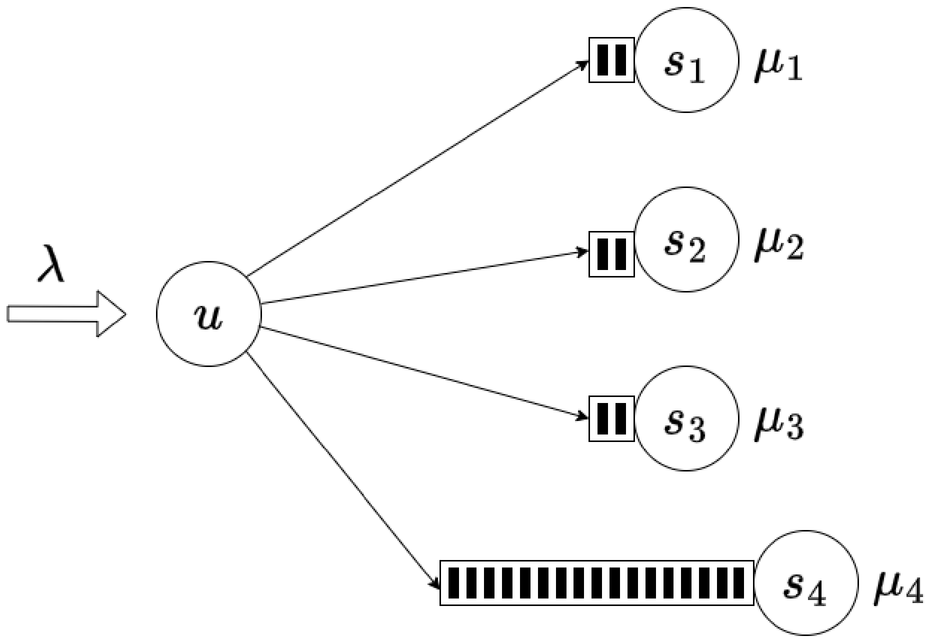
The Case of Known Utility Ordering



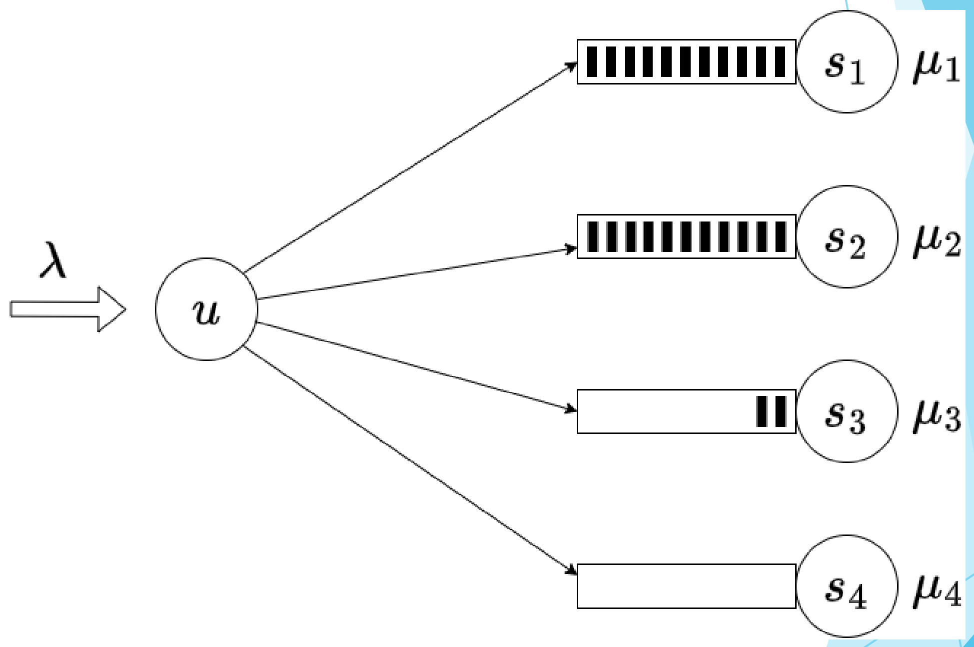
- Consider the servers to be arranged in decreasing order of their utility ordering: $v_1 > v_2 > \ldots > v_M$
- We define L as critical number of servers such that:
  $\sum_{m=1}^{L} \mu_m < \lambda < \sum_{m=1}^{L+1} \mu_m$
- To achieve low regret, we have to reduce idleness of top L servers
- Priority-K Policy prioritizes the top servers by assigning them the tasks first if queue length is less than K, else sends them to the next server

The Priority-K Policy

The Optimal Value of K



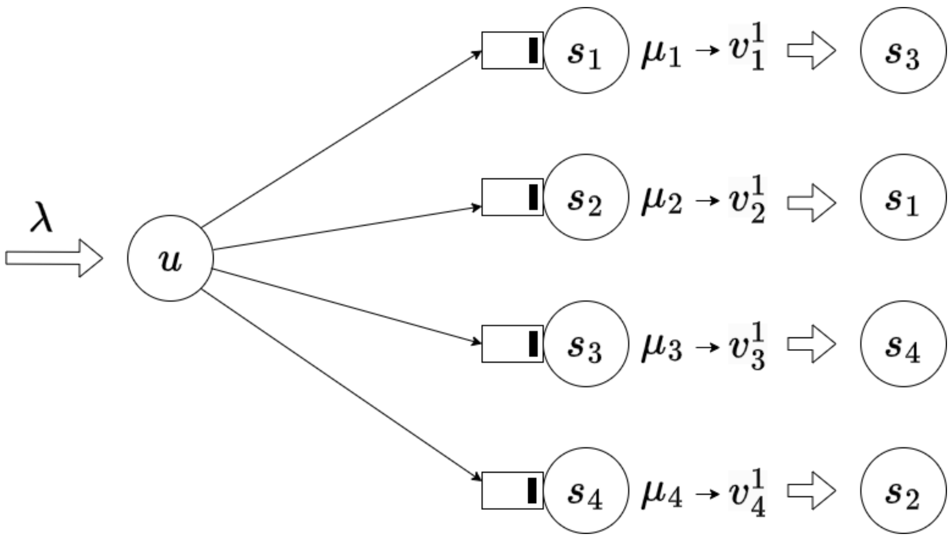
Idleness of the high utility servers due to small value of K



Overloading of the high utility servers due to large value of K

The Upper-Confidence Priority-K Policy

The Case of Unknown Utility Ordering



$$v_3^1 > v_1^1 > v_4^1 > v_2^1$$

- We follow an exploration-exploitation based approach for the unknown utility case
- First, some jobs are sent to all the servers to explore the servers' capabilities
- The order of the servers is then decided based on their feedback (observed utilities)
- Now, we follow Priority-K Policy according to the decided order to exploit the best servers
- With any change in order, we reiterate the exploration phase to get the best ordering of servers

Implementation

The Priority-K Policy

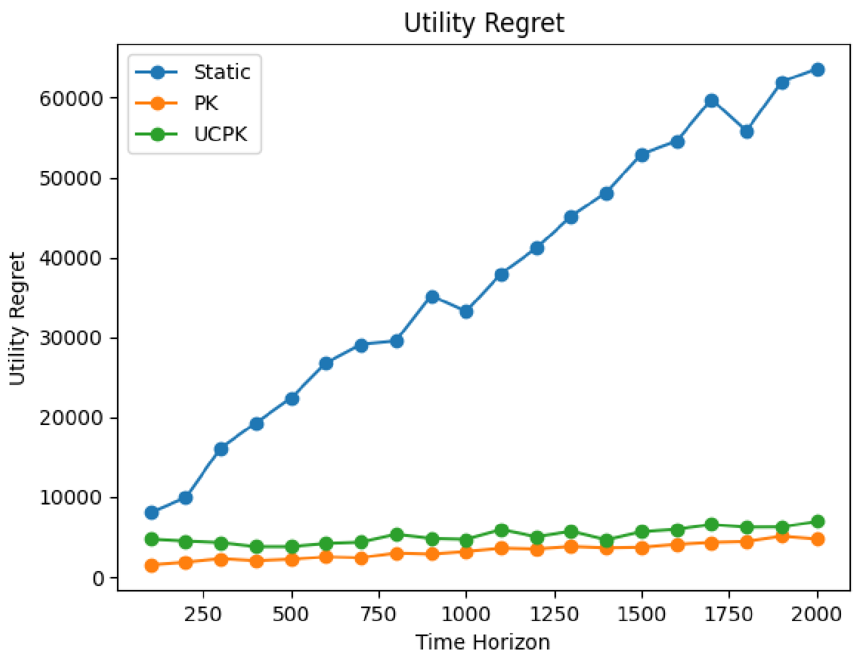
```
def priority_k(k, jobs, Ordering, servers):
    M = len(servers)
    for i in range(M):
        if jobs <= 0:
            break

        m = Ordering[i] - 1
        if servers[m].queue_length < k:
            available = k - servers[m].queue_length
            if available < jobs:
                servers[m].queue_length = k
                jobs -= available
            elif available >= jobs:
                servers[m].queue_length += jobs
                jobs = 0

    if jobs > 0:
        servers[M - 1].queue_length += jobs
        jobs = 0
```

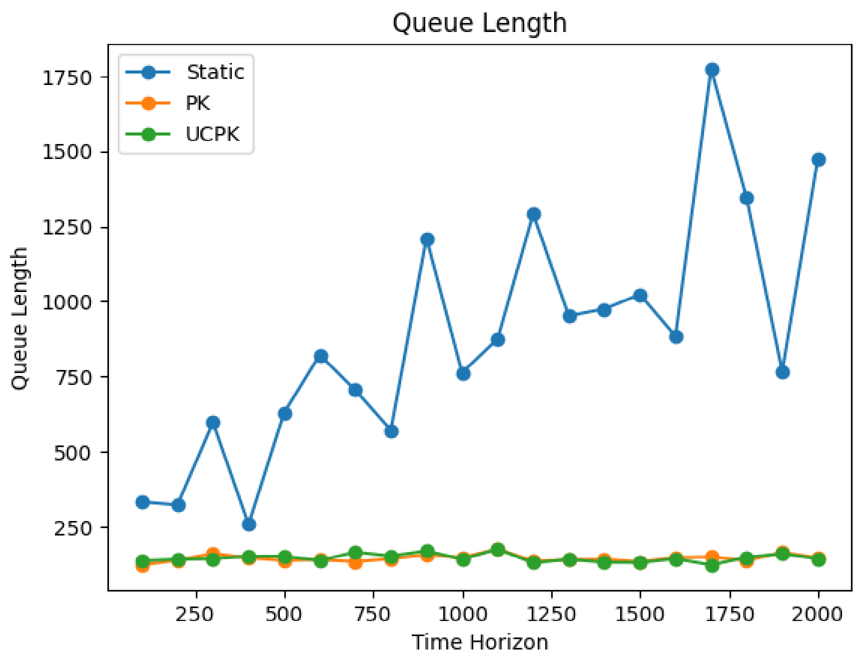
Implementation

The Upper Confidence Priority-K Policy

```
def upper_confidence_priority_k(k, Jobs, Ordering_UCPK, servers, T, Service_rates):
    Ordering_copy = [0] * len(Ordering_UCPK)
    M = len(servers)

    t = 0
    while t<T:
        if Ordering_UCPK == Ordering_copy:
            jobs = Jobs[t]
            priority_k(k, jobs, Ordering_UCPK, servers)
            for m in range(M):
                service_rate = random.choice(Service_rates)
                servers[m].process_files(service_rate)
            Ordering_UCPK = get_new_Ordering(servers)
            t = t + 1
        else:
            Ordering_copy = Ordering_UCPK
            m = 0
            jobs = Jobs[t]

            for m in range(M):
                if(m == M-1):
                    servers[m].queue_length += jobs % M
                else:
                    servers[m].queue_length += jobs // M
                service_rate = random.choice(Service_rates)
                servers[m].process_files(service_rate)

            t = t + 1
            Ordering_UCPK = get_new_Ordering(servers)
```

Simulation Results

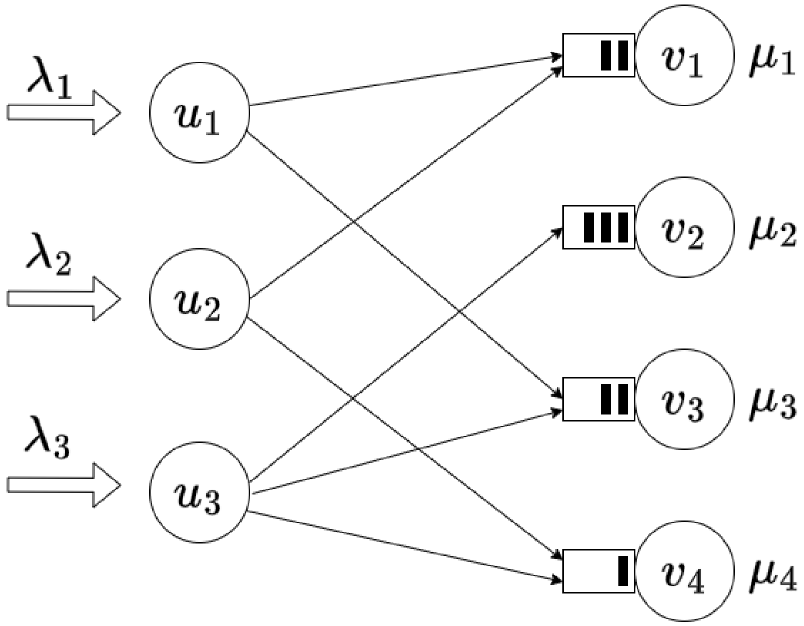




Utility Regret and Queue Length of Different Policies

Extension to general bipartite network

The Generalized Priority-K Policy



- Consider a system of N job dispatchers and M parallel servers
- The dispatcher and servers form a bipartite graph G
- We extend the proposed Priority-K policy and call it the Generalized Priority-K Policy
- Every job dispatcher adheres to localized Priority-K policy based on the order of the servers it is connected to
- We achieve logarithmic regrets for this generalized routing problem

Conclusion

- For designing efficient routing policies, we need to prioritize the usage of best servers and minimize their idle time
- The Priority-K based approach makes best possible use of the high-capacity servers while keeping in check the queue backlogs
- Using the exploration-exploitation approach with the Priority-K based approach, we come up with the Upper-Confidence Priority-K Policy
- We get logarithmic increase in value of regret for the Priority-K based policies
- This policy is then extended to the Generalized Priority-K Policy for the General Routing Problem with multiple job dispatchers

Thank you