MS Project Report

Fall 2013

Reinforcement Learning for P2P Backup Applications

Shikhar Mall

Date: October 18, 2013

Abstract

A five year study of file-system metadata shows that the number of files increases by 200% and only a select few file-types contribute for over 35% of the files that exist on a file-system.[1] It is difficult to point out a permanent selection of files that a user really cares about. This project uses reinforcement learning (RL[12]) to exploit the correlation between file-types, their usage patterns, multiple revisions etc., to extract out a selection of files which are "important" for an individual user. In this project, we integrate this file-selection approach with an open-source P2P backup application called CommunityBackup. With this approach, such a backup application can auto-configure its file selection without the user having to update the selection every time s/he relocates his/her folders.

The survey also points out that most file-systems are only half-full on average, independent of the user job category.[2] A P2P[10] backup application allows peers to share this average half-empty file-system to maintain redundancy over a backup network. This project collects features that CommunityBackup can utilize for its *peer selection*[14] using Q-learning algorithm to find out the geographically sparse, safe and consistent backup peers over a high-latency network. Another model presented in this project shows the use of an RL approach to improve the data-transfer throughput by adaptively raising the concurrency index to get around the ISP bottlenecks during backup application runtime.

Background & Applications

CommunityBackup

This project is a part of CommunityBackup, which is an open-source initiative by Zest Softech Pvt. India Ltd.; it allows peers to share their space and form a large virtual file-system. The peer network (as in theory) automatically decides what to backup, where to backup, how much to backup using information available from the target machine, for example, its location, capacity, bad-sectors etc. This project seeks to increase the overall probability of "recovery" in case of data-loss at any peer.

Data Backup & Security

Data backup is a crucial part of enterprise security. It needs to be a reliable source of recovery in case of a system breakdown. Data is generally backed up in external/tape drives or even on remote locations to make them geographically secure. This also involves maintaining redundancy across multiple locations. Various commercial applications are available which provide data backups on *remote locations*[16], for example, Mozy (by EMC) is an encrypted online storage, Amazon S3, Carbonite, SpiderOak (zero-knowledge security) etc.

Goal

There are three main goals of the project, to design and develop,

1. An agent that can find all the important files inside a computer.
2. A system that can match a suitable backup peer for all the data segments that a source peer has.
3. An agent that can efficiently utilize concurrency during I/O to retrieve data segments from chosen backup peers.

Backup peer: Machines in the network which are CommunityBackup (CBK, REF: 1) servers.
Source Peer: Machines in the network which are CommunityBackup (CBK, REF: 1) clients.
Data Segments: Piecewise data generated by CBK from "important" (REF: 2) files kept in a machine.
Network: Large virtual high-latency distributed backup network prepared by CBK.

Every peer is a CBK server and a CBK client.

Background and challenge

Due to the exponential increase in the number of files, it is not possible today to enumerate all the files inside a hard-disk and point out the most important ones. There is no single criteria to distinguish between files which are important to the user and ones which are not. The user might not be able to point out a permanent selection of files which s/he really cares about. Similar increase in the number of interconnected networks makes it impossible today to enumerate all the possible backup peers on this planet and point out the healthy ones. There is no single criteria to distinguish between peers which are a reliable backup node for the user and ones which are not. The user might not be able to point out a permanent selection of peers worldwide which s/he trusts and could rely upon for keeping backups safe.

The intuition behind Important Files

Files which are generally in use, occasionally modified and accessed, transferred/moved/relocated a number of times, showing a definite size change pattern and which are not system or "well known" files, are called "important" files. Like: Billing receipts, Banking credentials, Project reports, Code-work, Presentations etc. Some not necessarily our important files might be Backups, Logs, Temporary files, Configuration files.

Well known files: Files which have a strong web presence.
System files: Files related to the Operating System.
Generally in use and occasionally modified/accessed: Depends upon the users computing style.

The intuition behind Suitable Backup Peers

Peers which are generally online, occasionally receiving backups and contributing in recoveries, having transferred/moved/relocated to different networks a number of times, showing an acceptable data loss pattern, are responsive (good network), holding a good geographical dispersion index, have enough storage space, and which are low on crash counts are called "good" peers. So, a "suitable" peer is one which is the best deal available for particular kind of a data segment. Note that not every "good" peer is "suitable" for a data segment. Like: storage servers (good for very important data segments, which do not care much about network speed), home computers (good for regular backups), personal laptops (good for

data backups which need to explore new networks), dedicated backup peers (good for all kinds of backups), storage clusters (best for data segments which might need quick recoveries) etc. Some peers, not necessarily, are "suitable" peers for a particular type of a data segment, for example: Honeypot servers, peers located nearby, peers in the same network, peers with good network access but cheap hardware etc.

Geographical dispersion index: A measure of how spread the backup peers are on the globe, takes into account of "windowed data" of known coordinates of a subpopulation of peers. So, it is an approximate measure.
Crash counts: Number of times a CBK engine has crashed, to an extent that it loses any data it ever backed up.
Online availability: Depends upon the users computing style.

The main aim of the third of this project is to design and develop a data retrieval mechanism that can adapt to the continuous contraction and expansion of the network bottleneck so that an optimal concurrency index can be maintained at any time during the data retrieval process.

The intuition behind I/O concurrency

The data transfer mechanisms typically employed in web-browsers, dedicated download managers, servers, etc are typically meant to utilize a single socket over the network. The diagram below illustrates how the network channel is utilized during a data retrieval on a single socket.

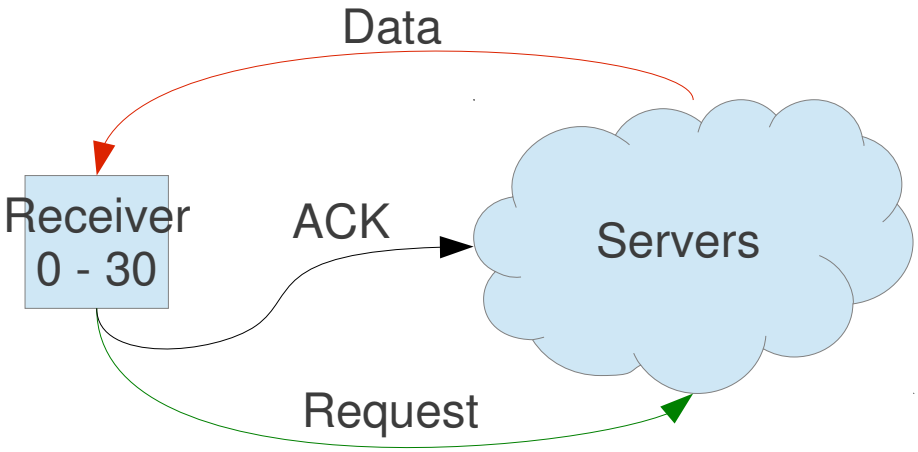


Figure 1: Single socket data retrieval

A receiver connects to network server(s) and initiates a data transfer stream. In the above diagram, ACK stands for acknowledgment, which is sent by a client after a segment of data has been received successfully. This traditional request-send-acknowledge policy is very conservative and hampers the performance of highly data-intensive applications. The underlying network usually maintains a transfer window (TCP Protocol[5]) which has a fixed maximum size. As a data transfer initiates, the window size is continuously incremented till it starts noticing data overflow, after which the window size is reduced to its half and the process is continued again.

Problem is that the data loss/overflow can be due to various reasons and, so, the network is never able to utilize the full available bandwidth.

The downgrade in performance due to data transfer window resize process can be overcome using parallel data streams.[3,4] An illustration of such a transfer is shown below:

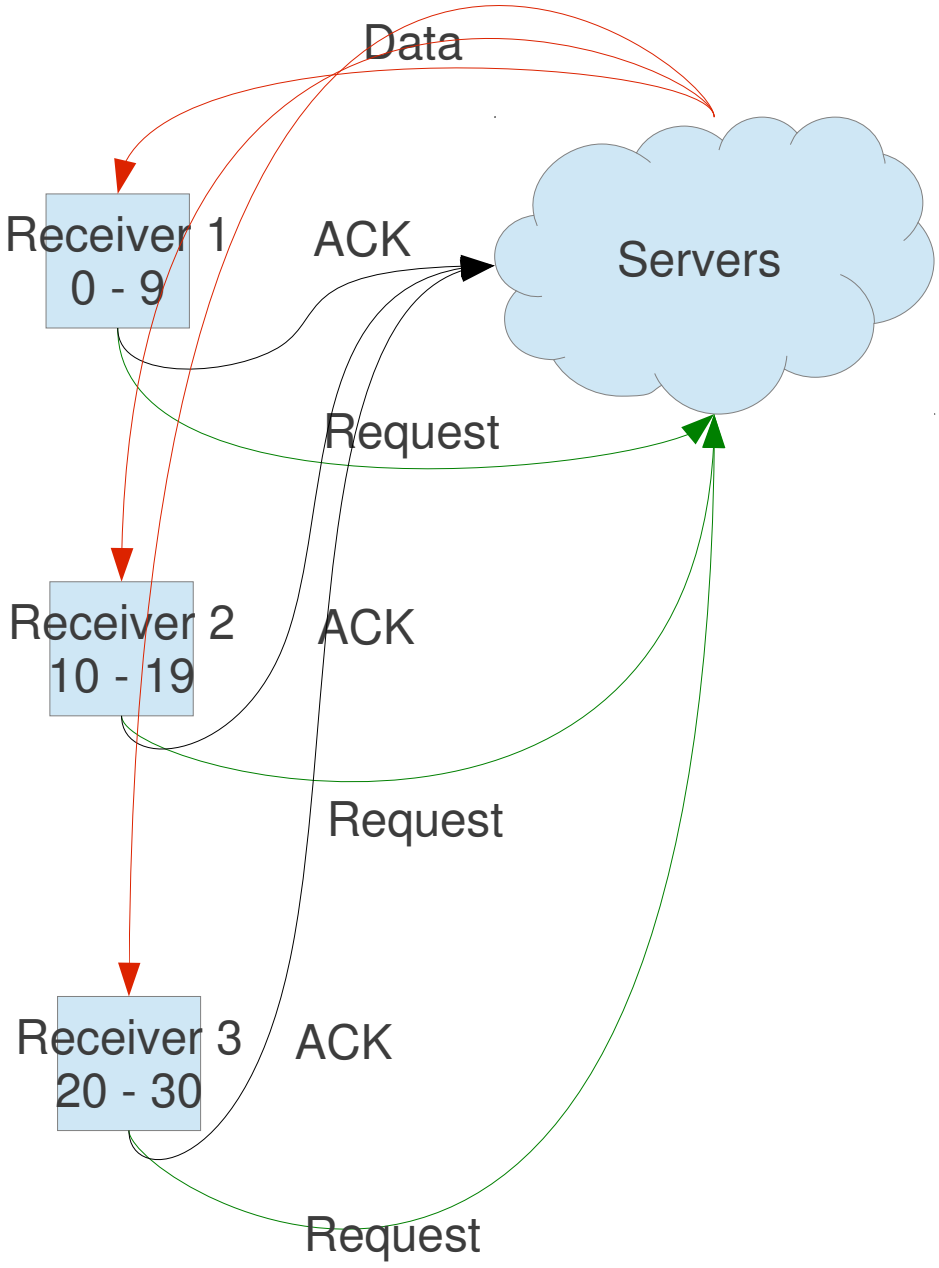


Figure 2: Multi-socket data retrieval

Concurrent I/O is initialed by dividing the entire data stream into smaller segments, then, each segment is retrieved on its own network (TCP) connection. However, if we strain the socket too much by initiating more than optimal number of current I/O, then, the overall performance of the

mechanism starts reducing. This happens because each I/O request adds a considerable amount of overhead to the retrieval process.

Some advanced multi-threaded file downloaders utilize this scheme by transferring data in smaller concurrent streams. This works much better than single socket data transfer mechanisms. Still a problem which this approach is that the concurrency index is never adjusted according to the current network conditions, instead, they are set to a static constant value throughout the retrieval process. This project demonstrates a mechanism by which the concurrency index can be adaptively changed on-the-fly during the file-retrieval process, according to the current network performance. This in-turn maximizes the network throughput.

Designing the MDPs

Before starting the description of the MDP[21] there is a need to define the two concepts I've used in the project:

- Scan Window
- Intelligent Data

These approaches have been selected so as to convert the POMDPs into fully observable MDPs.

Scan Window

It is a list of file selection using FIFO sorting based on the entry time in the change log. The agent talks to the environment simulator and gets a list of files and their feature scores. These scores are generated after an initial screening done by the simulation engine upon the file system statistics. This selection of files is called the "Scan Window". Each location in the scan window is an object containing a file along with its feature values.

Intelligent Data

The "travelling data" has been used as an intelligent agent for this project. The backup source not only selects and generates an important data segment, but, it also wraps it up into an intelligent packet which is aware of the backup network and is an RL agent itself. This packet then moves around in the network in search of a "better" host for itself. The agent talks to the environment simulator and gets the next available peer and its feature scores. These scores are generated after an initial screening done by the simulation engine upon the network and peer statistics. The project calls this agent, as "Intelligent Data". Each such agent in the network is an object containing the actual binary data along with its own feature values. (Data features are different from that of peer, REF 2)

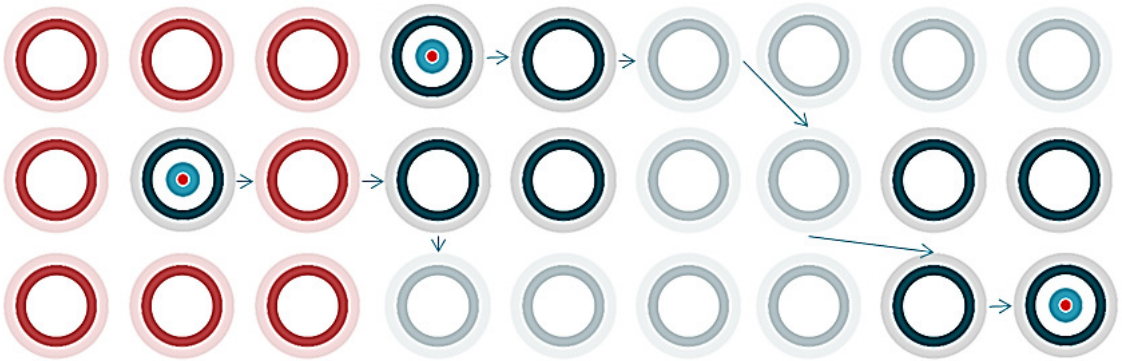


Figure 3, An RL Agent taking "move" action in a backup network.

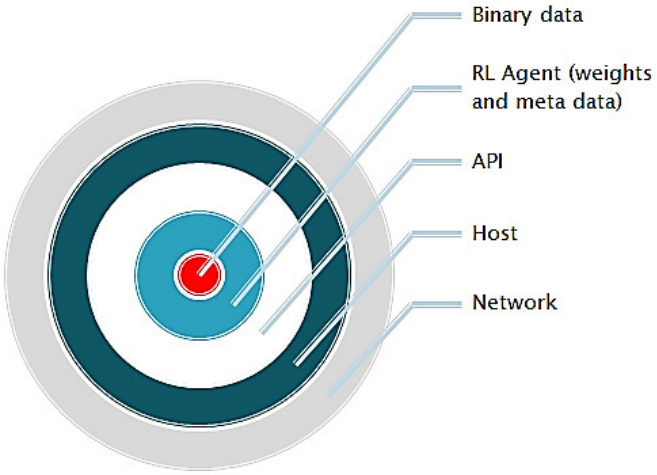


Figure 4, Pictorial representation of single RL agent carrying binary data

File Selection

State

A state is defined by window location, status of the selection engine, backup status of the file in the current window location. Every state is a unique combination of the following parameters:

Selection status of the selection engine (SS)

- When the agent has selected any file, this parameter is True.
- When the agent has NOT selected any file, this parameter is False.

Backup status of the file (BS)

- If the file was backed up previously then this parameter is True.
- If the file was NOT backed up previously then this parameter is False.

Window object location (WL)

- Location of the object at which the selection agent is currently concentrating upon.

Actions

Action Skip: Skip the current file and move on to the next file in the current window.
Action Backup: Select the current file and do a backup call to the simulator.

Rewards

Skip cost: Incurs a negative reward for moving on to the next file (-100).
Backup Reward: The reward function for the backup action is dynamic. We used several online APIs to generate reward for each backup action

Reward for backing up "well known" files

The simulator looks for the web presence of the file, and if it finds out that the file is a common entity over the internet then associates a negative reward factor with the file or else there is a positive factor.

- If file is a system/system-generated file, negative reward factor, else positive.
- If file is an ad-ware, negative reward factor, else positive.
- If file is spy-ware, negative reward factor, else positive.
- If file is a virus, negative reward factor, else positive.
- If file is a PUP or Trojan, negative reward factor, else positive.
- If file is a common executable, script or binary content then a negative reward factor, else positive.
- If file is a web-downloadable content then a negative reward factor, else positive.

Reward for backing up a possible system file

- If the location of the file is within one of the system folders, then a small negative factor or else positive.
- If the location of the file is within one of the home folders, then a small positive factor or else negative.

Reward for backing up a file Nth time

Depending upon the value of the reward factor, if it is > 0:

- Starting from +600 linearly reducing backup reward till a max negative value.
  Example, for increasing N values: 600, 480, 320, 120, -10, -120, -320, -480, -600, -720..... max –ve.
- Starting from -200 exponentially reducing backup reward till a max negative value.
  Example, for increasing N values: -200, -400, -800, -1600, -3200, -6400..... max –ve.

Peer Selection

State

A state is defined by agent's current host, status of the selection engine, response of the currently selected peer. Every state is a unique combination of the following parameters:

Selection status of the selection engine (SS)

- When the agent has selected any peer, this parameter is True.
- When the agent has NOT selected any peer, this parameter is False.

Response status of the peer (RS)

- If the peer has denied the backup proposal this parameter is True.
- If the file was NOT backed up previously then this parameter is False.

Backup Network (location)

- Network at which the selection agent is currently concentrating upon. This depends upon the current host.

Actions

Action Skip: Skip the current peer and move on to the next peer in the current network.
Action Backup: Select the current peer and call "move" using the simulator.

Reward

Skip cost: Incurs a negative reward for moving on to the next peer (-7).

Move reward: The reward function for the "move" action is dynamic. I have also used several online APIs as well to generate the reward for each "move" action.

Reward for moving to "well known" peers

The simulator looks for the web and network presence of the peer, and if it finds out that the peer is a common entity over the internet then associates a negative or positive reward factor with the peer depending upon the review it gets. Please note that some terminologies and text here have been lifted from an internet resource[13,20]:

- If peer is a known Anti-Infringement agency node, negative reward factor, else positive.
- If peer is a part of government, military, law enforcement, & intelligence agency networks, negative reward factor, else positive.
- If peer is a known P2P client or a tracker, big positive reward factor, else negative.
- If peer is a spammer, negative reward factor, else positive.
- If peer is / belongs to one of the following:
    - Level 1 (very high negative reward)

        - Companies or organizations who are clearly involved with trying to stop file sharing.
        - Companies which anti-p2p activity has been seen from.
        - Companies that produce or have a strong financial interest in copyrighted material.
        - Government ranges or companies that have a strong financial interest in doing work for governments.
        - Legal industry ranges.
        - IPs or ranges of ISPs from which anti-p2p activity has been observed.
    - Level 2 (reasonable negative reward)
        - General corporate ranges.
        - Ranges used by labs or researchers.
        - Proxies.
    - Level 3 (this is a paranoid list of peers, so, very small but a positive reward)
        - Many portal-type websites.
        - ISP ranges that may be dodgy for some reason.
        - Ranges that belong to an individual, but which have not been determined to be used by a particular company.
        - Ranges for things that are unusual in some way.
- If peer is / belongs to a known Educational Institution, small positive reward, else big positive.
- If peer is marked suspicious and is under investigation, very small positive reward, else big positive.
- If peer is an advertising tracker then a negative reward factor, else positive.
- If peer has been reported for bad deeds in p2p then negative reward, else positive.
- If peer is a known web-spider then negative reward else positive.
- If peer is a known hijacked node (used to deliver spam), then negative reward else positive.
- If peer is a known hacker and belong to such people, then negative reward else positive.
- If peer is a bad proxy (known for SEO hijacks, unauthorized site mirroring, harvesting, scraping, snooping and data mining / spy bot / security & copyright enforcement companies that target and continuously scan webservers) then negative reward else positive.
- If peer is a / belongs to a node of the following types, then strong positive rewards else regular positive rewards:
    - Dedicated CBK servers
    - Dedicated CBK clusters
    - Dedicated CBK organizations

Reward for moving to a known local peer

- If the location of the peer is within the current "local" network, then a small negative factor or else positive.
- If the peer has not been recently seen into the "local" network, then a small positive factor or else negative.

Reward for moving N^{th} time

- Depending upon the value of the reward factor, if it is > 0:
    - Starting from +600 linearly reducing backup reward till a max negative value. Example, for increasing N values: 600, 480, 320, 120, -10, -120, -320, -480, -600, -720..... max –ve.

- If reward factor is < 0
    - Starting from -200 exponentially reducing backup reward till a max negative value. Example, for increasing N values: -200, -400, -800, -1600, -3200, -6400..... max –ve.

Data Retrieval

States

Number of states are practically infinite, but for demonstration purposes there is a way to limit them. A state is defined by the current concurrency index and a success/failure indicator:

State parameters:

- Concurrency index
- Improved/deprecated boolean

Concurrency index indicates the number of concurrent I/O streams and "improved/deprecated" boolean value indicates whether after coming to this state there was a performance gain or not.

Actions

Add Segment

Before the data retrieval mechanism begins, the data is split into smaller segments as shown in the diagram below:

———

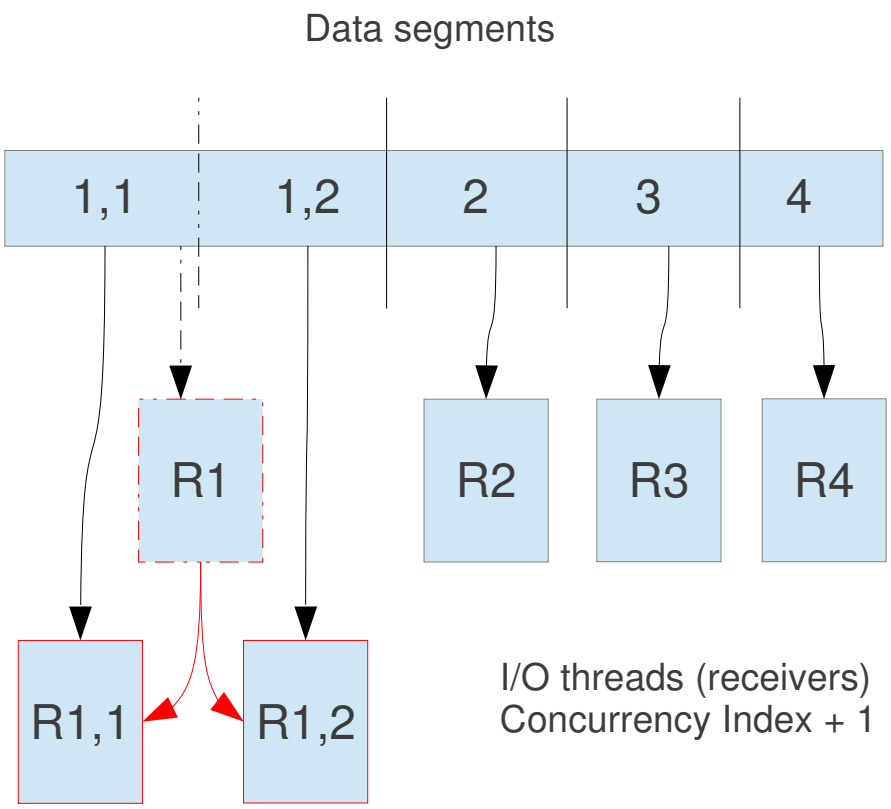


Figure 5: Adding a new segment

Adding a segment uses the following approach:

1. Find out the biggest segment that an I/O thread is downloading currently (segment 1 from illustration 3)
2. Split that segment into half (1,1 and 1,2 from illustration 3)
3. Shrink the segment of the original I/O thread to the dimensions of the first half. (R1,1)
4. Start a new I/O thread whose job is to download the second half segment. (R1,2)

Essentially, we pick up the largest existing segment and split it into half to create another I/O thread, which means increasing the concurrency index.

Remove Segment

Removing a segment calls for finding the smallest existing segment (target segment) and merging it with its left adjacent segment in-turn stopping the I/O thread which was downloading the target segment. This process is illustrated in the diagram below:

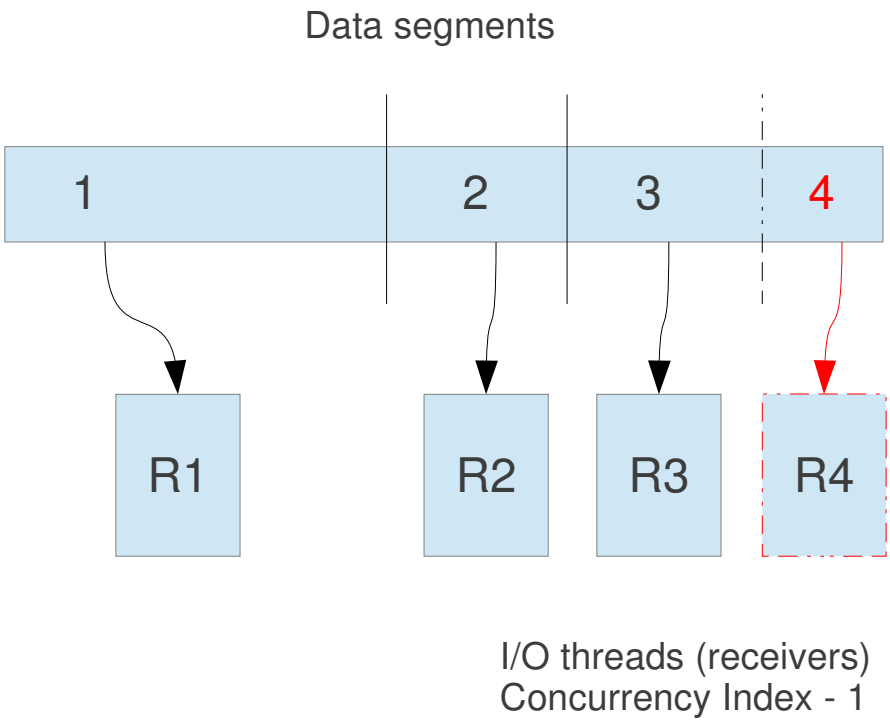


Figure 6: Removing an existing segment

Removing a segment uses the following approach:

1. Find out the smallest download segment and call it target segment (segment 4 in illustration 4)
2. Merge the target segment into the left adjacent segment
3. Stop the I/O thread corresponding to the target segment, in-turn reducing the concurrency index by 1. (R4 is removed)

Essentially, this action picks up the smallest segment and merges it with the left-adjacent segment. Note that it is not possible to merge into the right segment because each data segment is downloaded sequentially from left to right, and merging with the right segment means extending the left boundary of the data segment. Unfortunately, due to technical limitations, a thread cannot be restarted to download from the new segment beginning, unless we want it to download the entire segment again. Some improvements are possible here, but, it is a topic out of the scope of this project. Also, note that since removing a segment requires an existing left-adjacent segment to merge with, the first segment can never be removed because it doesn't have a left-segment available.

No Action

This action does nothing but waits for the next iteration.

Reward

Rewards are dynamic depending under what conditions any action is performed. If we enter into an improved state, there is a sure positive reward with varying amount, otherwise a sure negative reward with varying amount.

Improved state

A state is called "improved" if after taking an action the system shows a positive change in network throughput. This can happen in the following two ways:

1. Adding a new segment to utilize an expanded network bottleneck.
2. Removing an existing segment to stop straining the network socket which in-turn results in improved utilization (low overhead).

The magnitude of reward is dynamic and is directly proportional to the observed change in throughput.

Transitions

For file selection, it depends upon the selection and backup status of the file. The Agent starts from the first object in the scan window list. Skip action will can only be taken on files where the Selection Status (SS) value is False and will change the state where Backup Status (BS) value do not change .If current object is the last element in the scan window agent jumps to the first element in the scan window or else agent moves to the next element in the scan window. Backup action will change the state to a terminal state where the Selection Status (SS) is true.

Transition to a new state in peer selection always depends upon the selection status and response of the peer[15]. The Agent retrieves the first peer from the network. Skip action can only be taken in a state where the Selection Status (SS) value is False and will change the state where Response Status (RS) value do not change .If current object is the last remaining peer in the network then agent jumps to the first peer in the network. Select action will change the state to a terminal state where the Response Status (SS) is positive and the agent starts all over again.

The following events trigger transitioning into a new state for data retrieval mechanism:

- Add a new segment (concurrency index + 1)
- Remove a new segment (concurrency index - 1)
- I/O Thread stops (finishes downloading, concurrency index - 1)

Essentially, any change into the system that alters the concurrency index causes the state to change.

Q-Learning for file selector agent

The first method we tried is the use of Q-learning on a small state space with files ranging from 10 to 20 and Q-Learning with GLIE Policy II (Boltzmann Exploration) explore/exploit was implemented for this part. Learning performance or different number of training epochs is shown in Figure 3. The maximum reward for any file was up to 600 and would reduce linearly with each repetitive backup. To evaluate the average reward the agent was tested on a new set of files after training.

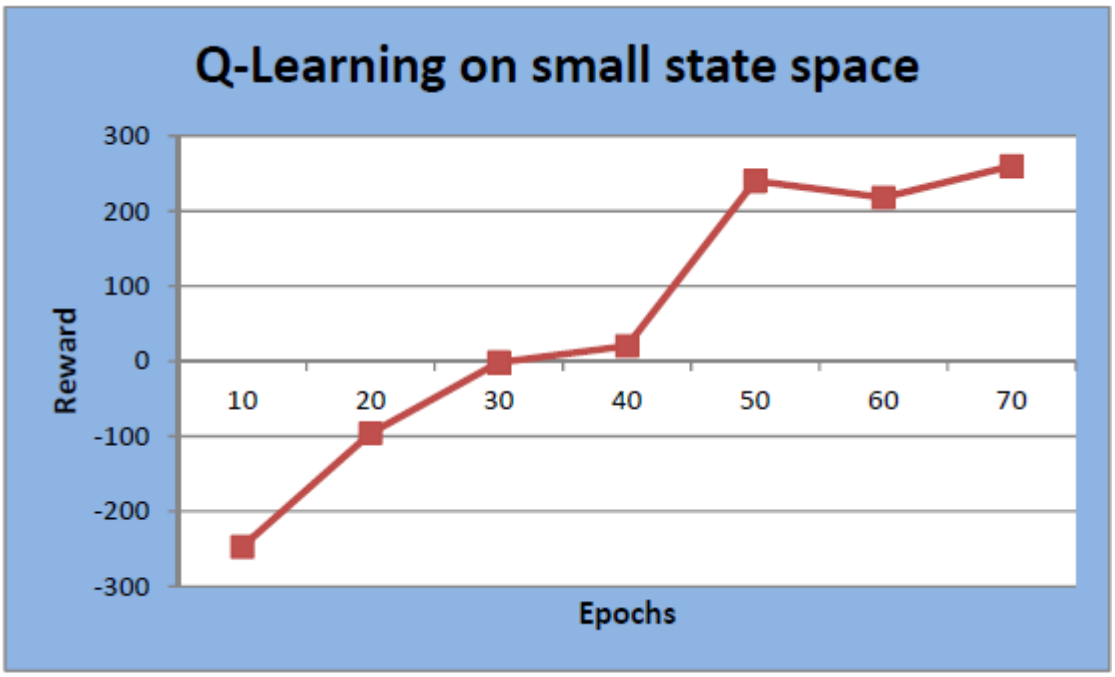


Figure 7, Average reward on small state space over training epochs

The agent starts with a negative average reward with small number of iterations and reaches the optimal performance with about 70 training epochs in a state space of made from 10 files.

Q-Learning for Intelligent Data agent

The first method I tried is the use of Q-learning on a small state space with peers ranging from 50 to 70 and Q-Learning with GLIE Policy I (eGreedy) explore/exploit was implemented for this part. Learning performance of different number of training epochs is shown in Figure 4. The maximum reward for any peer was up to 600 and would reduce linearly with each "move" action. To evaluate the average reward the agent was tested on a new network simulator after training.

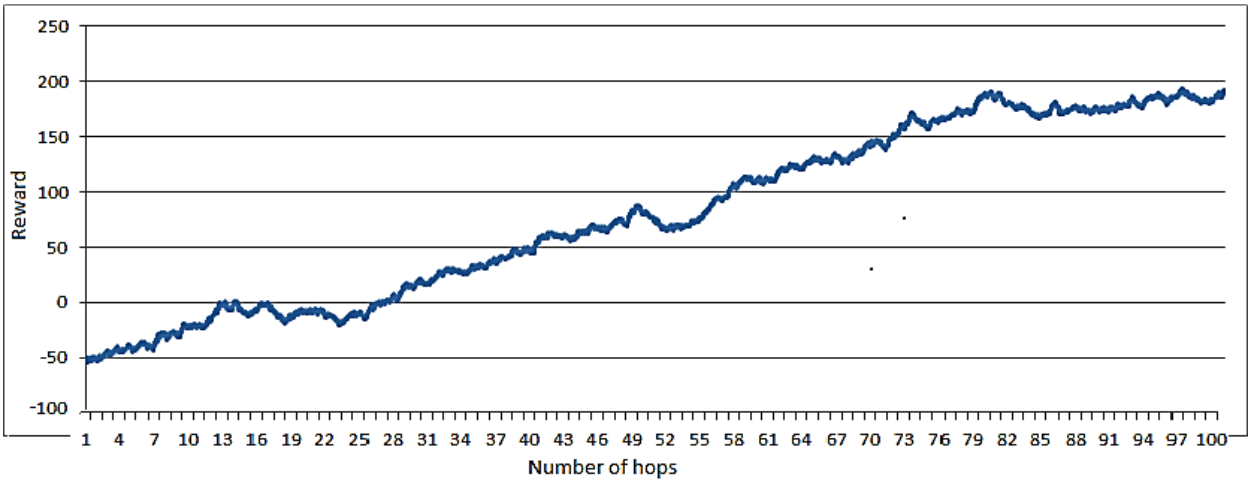


Figure 8, Q-Learning on small state space

The agent starts with a negative average reward with small number of iterations and reaches the optimal performance with about 85 training epochs in a state space made from 60 peers.

Q-function Approximation

While the above technique works for small state space with small numbers of peers (50-70) in the entire network, it is obvious that for a large number of peers in the network the state space would be very large (for every peer we need to know all the sixteen features of the peer and for data) and maintaining Q-value for each state would be extremely difficult. That leads to the implementation of Q-function Approximation for both peer selection and file selection agents.

Data Features for Function Approximation

Below are the features used for this file selection. The detailed justification for our selection is provided at the end of the report in appendix II.

We rank every feature value and scale down to some max value to take control the number of states.

1. **File size / Average change in file size.**
    - Assumptions:
        - Zero length files usually are of no importance.
        - Large files (> 100MB) usually are media, installation or database files and might not be of much importance.
2. **File Type**
    - Assumptions:
        - All system/executable file types score lower when compared to document, music, image or text types.
3. **Average modification interval:**
    - Assumptions:
        - We assume that files which have been modified most number of times, recently, are the ones important to the user.
4. **Average File Usage: Number of times, a file has been found loaded inside RAM.**
    - Assumptions:
        - We do not want to backup files which have never been opened by the user.
5. **Backup Times: Number of times, a file has been backed up.**
    - Assumptions:
        - We do not want to backup same files again and again.

While with q-learning even in very small state space it took about 50 training epochs before the agent started gaining average high reward, approximate function learning converges in 20~30 iterations.

The below table contains the Θ values for Q function approximation at 40 iterations.

Θ_1	Bias	0.8
Θ_2	File Type Popularity	2.4221
Θ_3	Average modification interval	23.0529
Θ_4	Average File Usage	1.2439
Θ_5	Average File Size Change	40.0322
Θ_6	Number of duplicate backups	48.9141

From the above table we can conclude that features like Average modification interval, Average File Size Change, Number of duplicate backups contribute more in the final file selection. This configuration of feature weights solely depends upon individual user track.

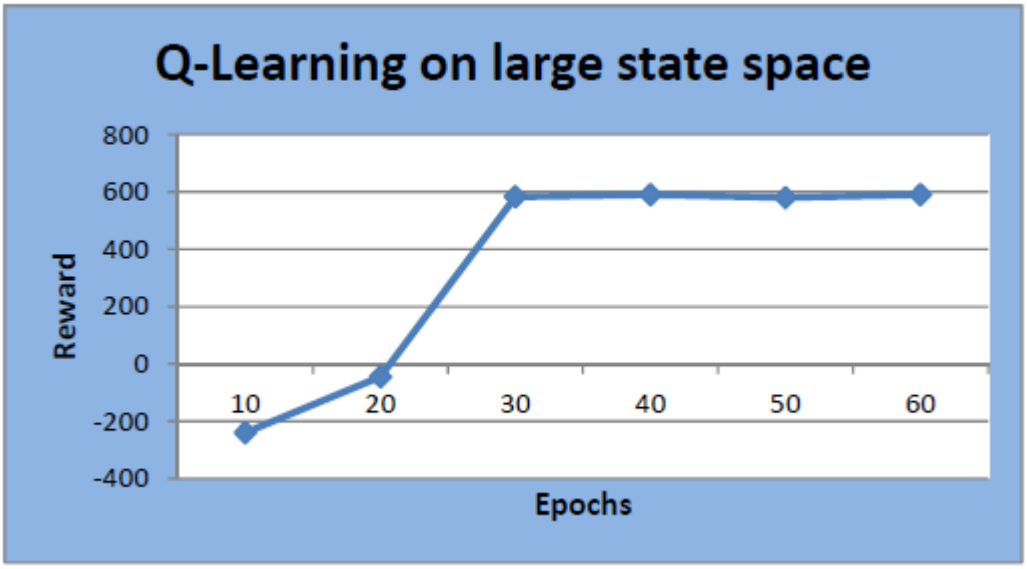


Figure 9, Average reward on large state space over training epochs

Peer Features for Function Approximation

Below are the features used for this peer selection. The detailed justification for our selection is provided at the end of the report in appendix II.

We rank every feature value and scale down to some max value to take control of the number of states.

1. **Recovered Important Data**
    - Assumptions:
        - Recovering any amount of useless data shouldn't impact a decision.
        - We use the product of data importance measure received from file selection engine in data features (REF: 2) and data size to calculate this peer feature
2. **Geographical dispersion index**
    - Assumptions:
        - Backing up to nearby peers doesn't help in case of a natural disaster.
        - This is measure of how spread the backup peers are on the globe, takes into account of "windowed data" of known coordinates of a subpopulation of peers. So, it is an approximate measure.

3. **Data hop count**
    - Assumptions
        - We want a stable matching solution of peer and data combinations, but, we also want to put a restrain on the number of "move" actions an agent takes.
        - Gets incremented by one with every hop the "intelligent data" makes in the network.
4. **Data size**
    - Assumptions
        - Even if an "intelligent data" is important, a peer might not be able to accept its backup proposal because of lack of available space.
5. **Data importance**
    - Assumptions
        - We prefer backing up important data than regular backups.
        - A sum of all the features of the "intelligent data"
6. **Target peer network latency**
    - Assumptions
        - Peer with high access time would be troublesome and should be avoided whenever possible.
7. **Data redundancy index**
    - Assumptions
        - The system prefers data backups which are less redundant.
        - With every "move" action, the previous host can choose to retain the copy and improve redundancy. (this feature is included, but not used, as there is no implementation to control data redundancy yet)
8. **Target peer availability (relative to the source)**
    - Assumptions
        - Even if a peer is available mostly, but, both the source and target peer should have consistent schedules of their online presence.
        - This feature measures the relative presence of the current host and target peer.
        - This also covers "time-zone effect" (problems caused due to peers residing in different time zones)

9. **Target peer's free storage space**
    - Assumptions
        - If the storage space of the target peer is not enough, there is no point in making a backup proposal as it will ultimately get rejected.
        - This is not an exact measure, only shows a percentage value.
10. **Target peer's used storage space**
    - Assumptions
        - This feature helps giving low preference to fresh/blank peers, because they might not have been tested yet.
11. **Target peer activation time-stamp**
    - Assumptions
        - This depicts a relative index on how old the current peer is.
        - The system should prefer an old peer over new ones.
12. **Target peer data loss index**
    - Assumptions
        - Depending upon data importance, losing a very importance data segment will raise the index much higher than that when losing a segment of low importance.
        - A peer which has high loss index should be avoided.
13. **Peer acceptance rate (of backup proposal)**
    - Assumptions
        - There can be various reasons to why a backup peer is not accepting backups, this feature help evaluate all such reasons into a single measure.
14. **Environmental feature 1: Lightning patterns at target peer**
    - This feature is collected from a third party resource.
    - Assumptions
        - It is not recommended to use a backup peer which resides at a location where there is expected lightning disaster.
15. **Environmental feature 2: Wind speed measure at target peer**
    - This feature is collected from a third party resource.
    - Assumptions
        - It is not recommended to use a backup peer which resides at a location where there is expected cyclone disaster.
16. **Environmental feature 3: System temperature at target peer**
    - This feature is calculated from the system hardware.
    - Assumptions
        - It is not recommended to use a backup peer which has poor heat sink and is bound to crash very soon due to system hardware failure.

While with q-learning even in very small state space it took about 85 training epochs before the agent started gaining average high reward, approximate function learning converges in 40~50 iterations.

The below table contains the Θ values for Q function approximation at 60 iterations:

Θ_1	Bias	0.8
Θ_2	System temperature at target peer	-8.73549
Θ_3	Data size	-0.59676
Θ_4	Lightning patterns at target peer	-8.13568
Θ_5	Geographical dispersion index	6.032791
Θ_6	Target peer availability	1.241027
Θ_7	Data redundancy index	-1.10085
Θ_8	Peer acceptance rate	6.862445
Θ_9	Target peer network latency	-9.70861
Θ_{10}	Target peer free storage space	10.0713
Θ_{11}	Target peer activation time-stamp	1.101388
Θ_{12}	Data importance	14.09615
Θ_{13}	Target peer data loss index	-8.14309
Θ_{14}	Data hop count	-3.47902
Θ_{15}	Recovered Important Data	3.588342
Θ_{16}	Peer acceptance rate	1.404046
Θ_{17}	Wind speed measure at target peer	-5.52948

From the above table we can conclude that features like Data importance, Target peer free storage space, Recovered Important Data, etc. contribute more in the final peer selection. This configuration of feature weights solely depends upon individual data track.

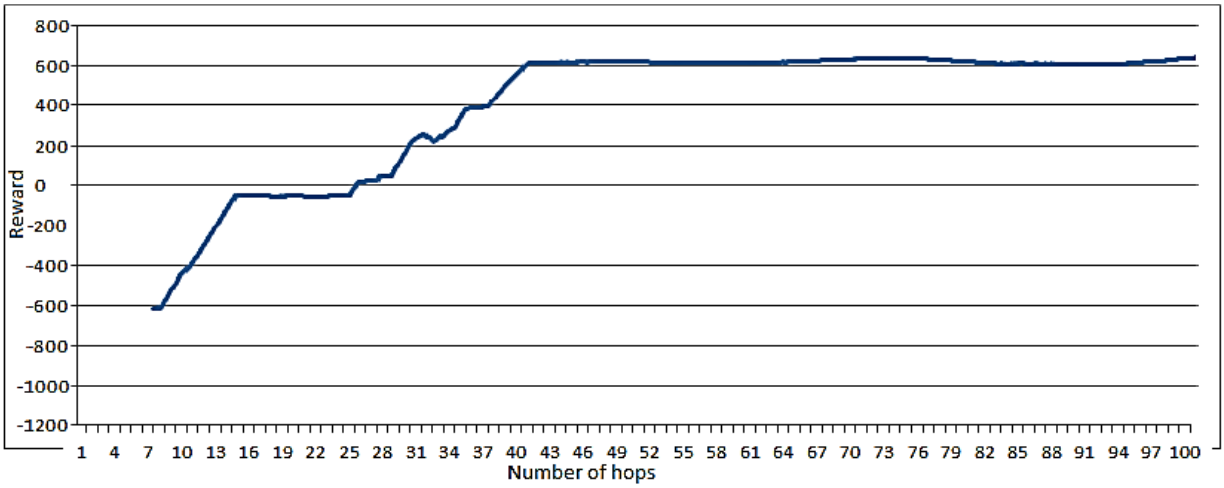


Figure 10, Average reward on large state space over training epochs (Q-Learning on large state space)

Comparison with Greedy Agent

The Q-learning agent was compared against a greedy agent. A greedy agent picks the backup peer greedily at any state while q-learning agent uses q-learning to pick an action. Figure 7 & 8 show an average reward accumulated by both the Greedy agents. Notice that the accumulated reward for the Q learning agent is less than the greedy agent, this is because of the fact that during the initial iterations the data moved quite a lot and the greedy agent did not take skip actions which has a negative reward and as a result the greedy agent was accumulating higher rewards than the Q learning agent. However, in both the cases, with increased number of iterations the greedy agent ends up in a suboptimal solution and hence the rewards were moving towards negative.

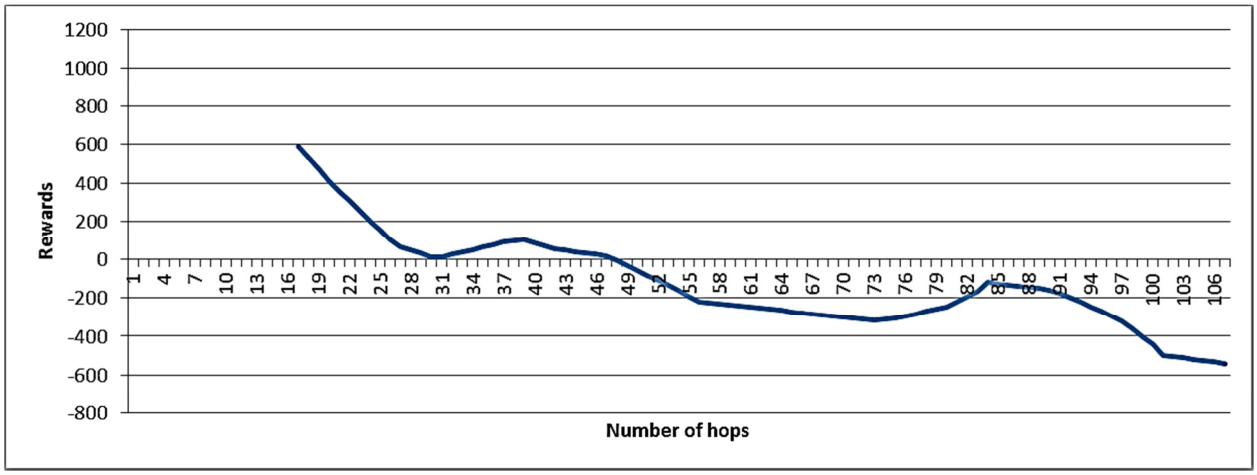


Figure 11, Average reward on small state space over training epochs for Peer Selection Greedy agent (Greedy-Learning on small state space)

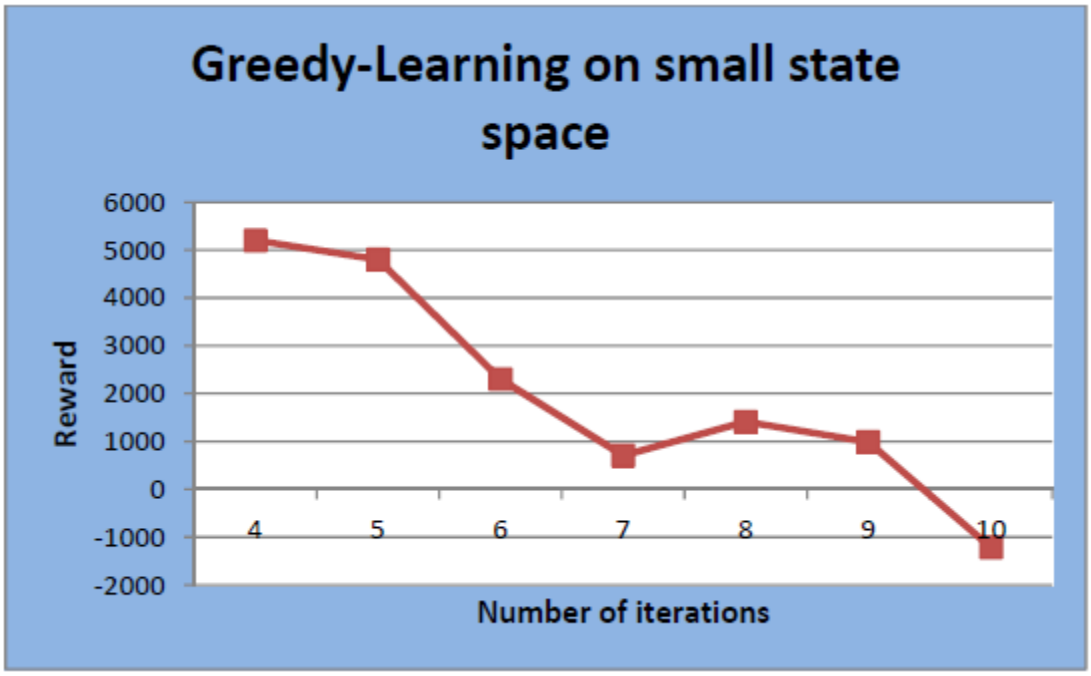


Figure 12, Average reward on small state space over training epochs for File Selection Greedy agent

Data Retrieval Features for Function Approximation

There is only one feature (sensor) required for the third part of the project. It is called network-throughput. Any action that the system takes, directly affects the overall performance of the data transfer. This feature has a last 60 seconds averaging window which is used to calculate the change in data retrieved over the last one minute.

Learning Agent

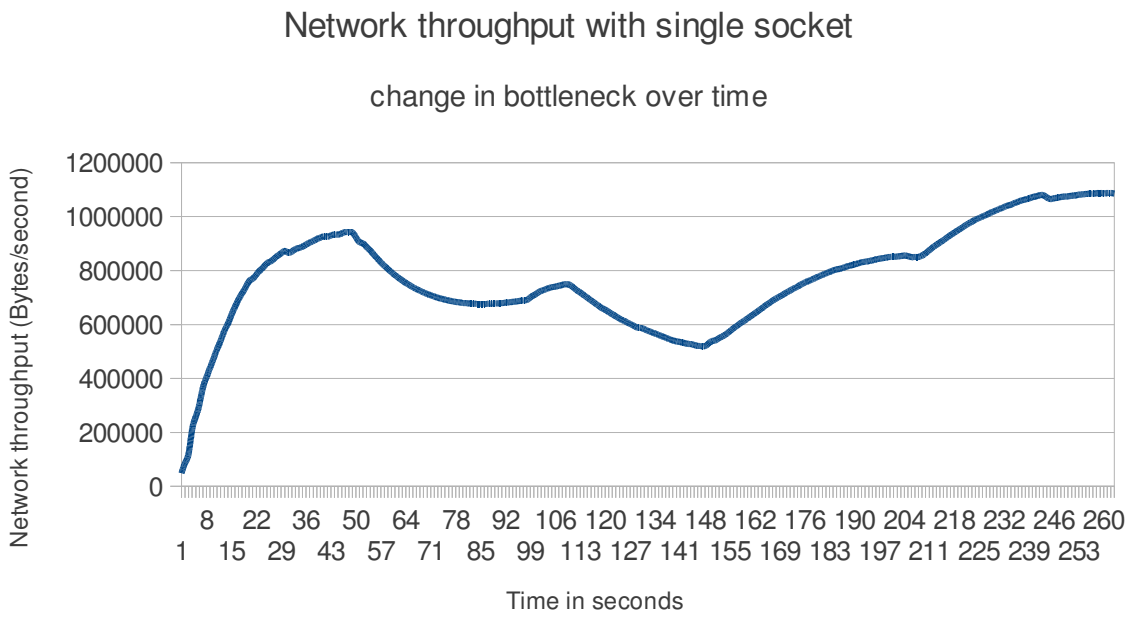
The learning agent implements an *active reinforcement learning*[11] mechanism using Q-learning. Initially, the agent has no policy and so, the Boltzmann GLIE policy picks up a random action until at a later stage with low temperature values where the agent takes more "informed" decisions. The exploration/exploitation policy is to select an action with probability:

$$P(a \mid s) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_{a' \in A} e^{\frac{Q(s,a')}{T}}}$$

Figure 13: Boltzmann explore/exploit policy

Where, T is the temperature. Larger T leads to greater exploitation. We start with value of 100 for T and reduce by 0.03% in each iteration until it is 1.

The environment is uncertain as the network bottleneck keeps on changing continuously. There is no one optimal concurrency index for all scenarios. The graph below shows the change in network bottleneck over time.



As the graph shows there are sharp changes in network bottleneck over time. The above graph is averaged and null-cycled over 50 seconds, which shows that even after every 50 seconds the network bottleneck either gets increased or decreased. An upward trend means that it got increased whereas a downward trend means that it got decreased over the last 50 seconds.

The agent learns an optimal Q function which is the expected value of taking one of the actions (add or remove segment and, no-action) in any state and then following an optimal policy thereafter. We perform TD updates after each action:

$$Q(s,a) = Q(s,a) + \alpha \times (R(s) + \beta \max_{a'} Q(s',a') - Q(s,a))$$

Where α is the learning rate and β is the discount factor. Note that for this project we particularly need a high learning rate, because, as shown above, the network bottleneck keeps on changing very frequently and so, in order to exploit the changes, the agent will have to respond quickly to them. For all experiments conducted, we've used a static discount factor which is 0.8.

Note that we do not require a transition function as we are learning the model directly, similar to updates in temporal difference learning mechanism. Also, note that there is not terminal state, so, the problem is basically infinite-horizon and the agent is sent an exclusive stop signal by the system that terminates its execution. This happens when concurrency index drops down to zero, meaning that there are no I/O threads alive, which in-turn means that there are no segments left to download.

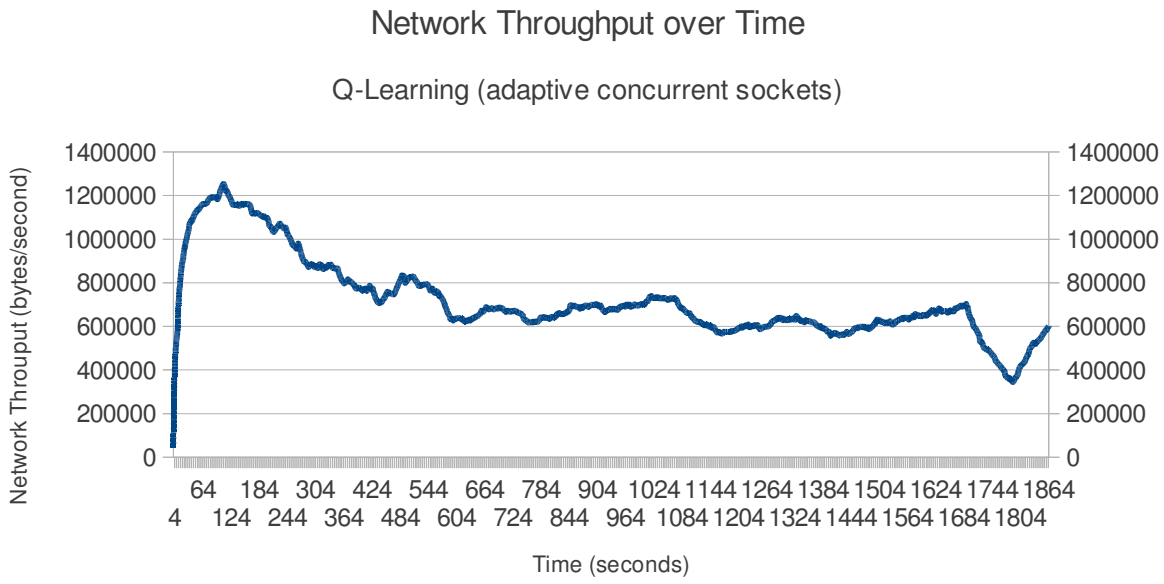
As this is not a goal based problem, there is no need to speed up or back-propagate the rewards. The environment is not dangerous as we are only manipulating the concurrency index and the actions never cross the I/O thread limits (min and max allowed concurrent I/O).

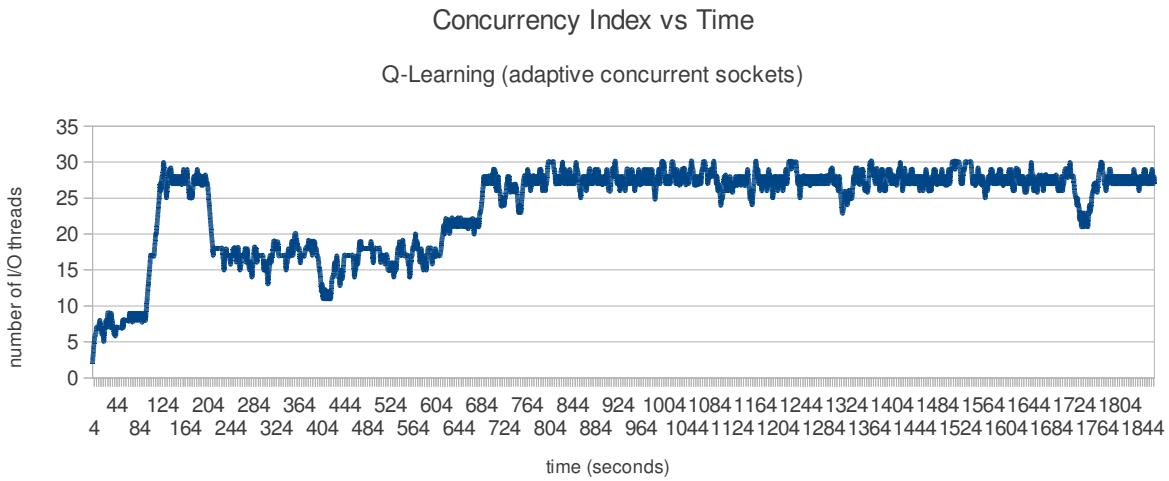
Memory Usage

Each state action pair has a unique optimal Q-value, so, for an environment with limits of "m" minimum threads, "x" maximum threads and "n" actions, we can have (x – m)n number of Q values. So, the space complexity is O(xn) assuming "m" to be always equal to 1.

Adaptive Sockets (Model Free Q-Learning) vs Single Socket performance:

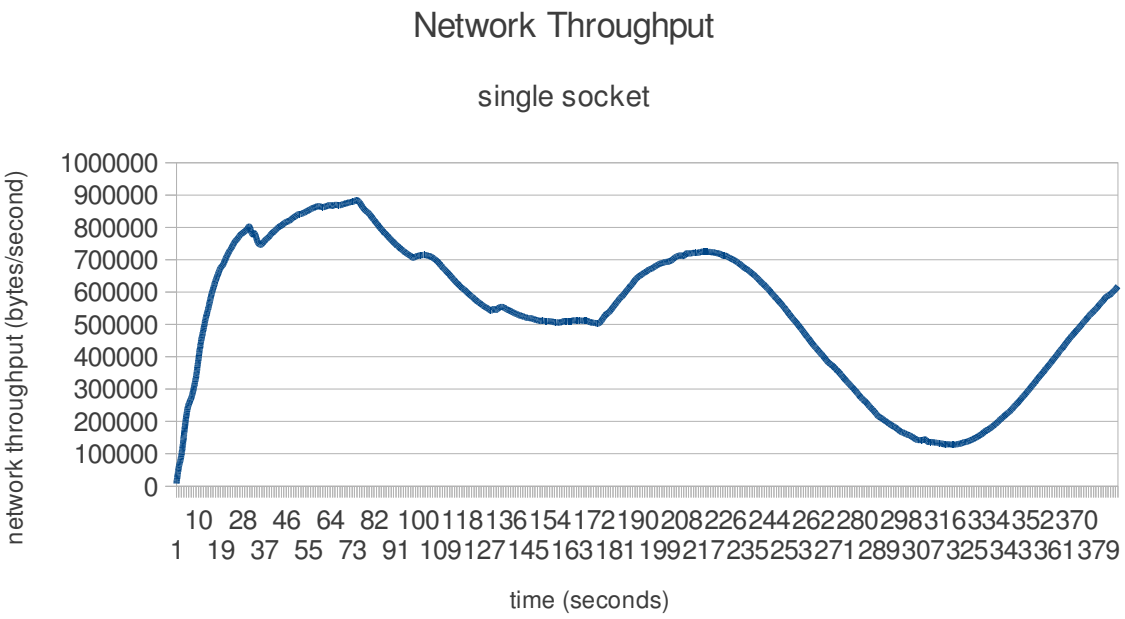
The graphs below show how the network throughput and concurrency index change over time. Notice that there is a sudden "burst" in throughput initially. This is a very common behavior shown by all Internet Service Providers (ISPs) today. An ISP tries to boost initial data transfer because that helps in caching over audio/video streaming. At the later stage the throughput stabilizes to its actual value.





The above graph shows change in concurrency index over time, which is basically the number of parallel I/O streams retrieving data at any instance. Note that the Q-learning agent responds to the initial burst in throughput given by the ISP with over 25 concurrent I/O threads and then reduces down to 17 threads where-after it picks up slowly and stabilizes to 27 threads. There is still some disturbance towards the end because of the high learning rate.

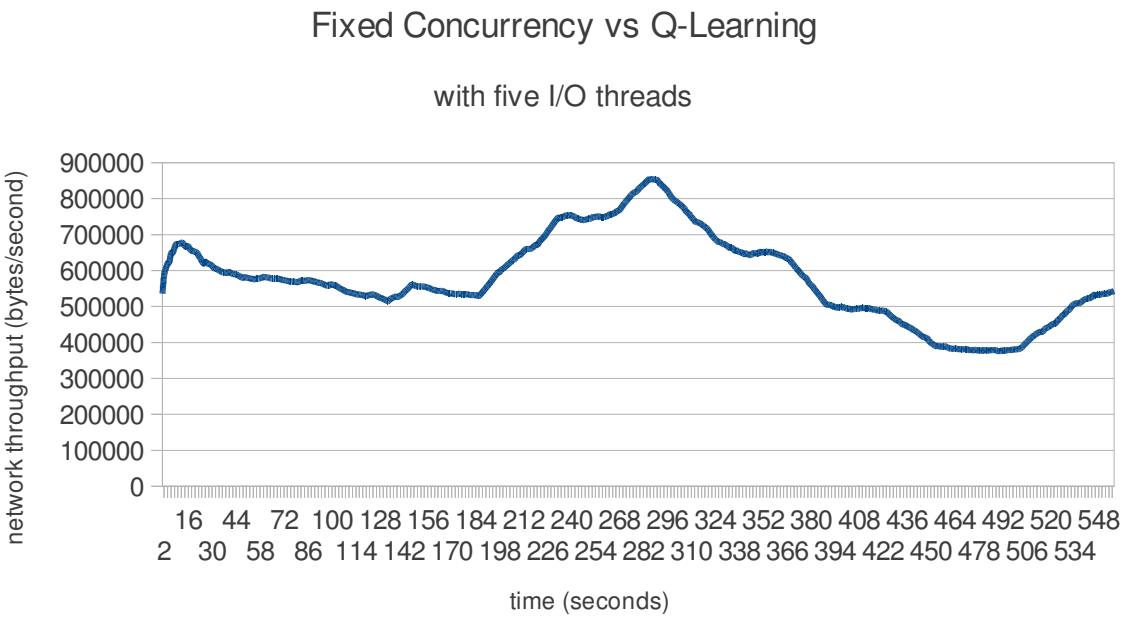
Comparing this with the single socket performance (concurrency index is always 1), we notice that the Q-Learning throughput performance is much smoother and reacts to changes in the bottleneck.



As can be seen from the above performance graph for single I/O thread data retrieval, it's overall performance gets hampered by the ISPs attempt to curb the transfer rate multiple times during the experiment. This can happen due to various other reasons as well, for example, sudden increase of data usage by other applications or increased number of clients over the network. The average throughput for single socket was 5.2 Mbps whereas it was 7.3 Mbps for **adaptive sockets delivering a sheer 42% increase in performance**.

Adaptive Sockets (Model Free Q-Learning) vs Fixed Parallel Sockets performance

Here, we compare the performance of Q-Learning with a static concurrency index value. In static concurrency scenario there is no scope for adding or removing a segment, so, we start with a fixed number of parallel I/O threads[8] for this experiment. Whereas, for the adaptive sockets, Q-Learning will decide the optimal number of sockets on the fly. The performance graphs for Q-Learning agent is shown if previous sections. The performance graphs for fixed parallel sockets is shown below:



As can be seen from the above graph, the parallel sockets do resolve the problem with single sockets and respond in a much better way to bottleneck changes around 464th second. Still, there can be a better concurrency index value for this experiment. According to Q-Learning 27 is the optimal concurrency index for this experiment. The overall throughput static concurrency with five I/O threads was slightly above than single socket, 5.7 Mbps compared to 7.3 Mbps with adaptive sockets, which makes it a **significant 28% increase in performance with adaptive sockets**.

Q-Table and Impacts of Learning Rate

The Q-Table is attached along with this report, and it shows the value for each state (concurrency index, improved) and action (add, remove, no-action) pair. The only noticeable impact with learning rate is that if we keep it too low (~0.1) then the agent starts behaving like a fixed-concurrency index agent as it responds (if it does) very slowly to the change in network bottleneck size. A too high value (>=0.8) reduces the performance drastically, even lower than the single-socket performance because of the added overhead of taking quick add/remove segment actions. Adding a segment initiates a new request and dropping a segment closes the request, so, both of these actions consume some network. This overhead causes a lot of bandwidth to get wasted away. For the experiments done in this project, learning rate was 0.5 which seemed optimal based upon empirical data not covered in this report.

Observations

1. While the Q-learning seemed to perform very good once it is trained for an appropriate number of training epochs, we found the most difficult part is to tune it to the right parameters (e.g. discount, learning rate, rewards) to get the learning algorithm to work.
2. I tried to increase the network size to 1000 peers for learning with feature approximation but the average reward collection was not as I had expected, this might be to various causes like the features selection was either too simple or not completely accurate which could also be anticipated from the feature weights of some of the features like Peer acceptance rate. Another reason for this might be the parameters need to be tuned more effectively.
3. While working upon the file selection part, I noticed the same behavior (as in 2), except that learning was for a window size of 500 files.
4. Up to 30-40% increase in performance compared to single socket data retrieval and in some cases even higher than that.

Data Retrieval

1. Up to 28% increase in performance compared to fixed-five parallel sockets data retrieval.
2. Too high learning rate degrades the performance even lower than single socket agent
3. While the Q-learning adaptive socket agent seemed to perform very good once it is trained for an appropriate number of training epochs (at high temperatures), it was apparent that the most difficult part is to tune it to the right parameters (e.g. discount, learning rate, rewards) to get the learning algorithm to work.
4. In an attempt to explore the corner cases the thread limits (concurrency index limits) were set to as low as 1 to 2 and also as high as 70 to 100, but the average reward collection was not as expected because optimal value picked up in 70-100 range was 75 whereas it should have be only 70 because the actual optimal value is just 27 when we set the range from 1 to 100. It might be because of the temperature of Boltzmann exploration function picking up too much randomness due to slow decrease in temperature. (0.03% every iteration). A better algorithm to control the temperature was needed.

References

1. A Five-Year Study of File-System Metadata, Nitin Agrawal, William J. Bolosky, John R. Douceur, Jacob R. Lorch, *Microsoft Research*
2. A Large-Scale Study of File-System Contents, John R. Douceur and William J. Bolosky, *Microsoft Research*
3. Sivakumar, H, S. Bailey, R. L. Grossman, "PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks", Proceedings of IEEE Supercomputing 2000, Nov., 2000. http://www.ncdm.uic.edu/html/psockets.html
4. SuperNet Network Testbed Projects: http://www.ngi-supernet.org/
5. Transmission Control Protocol (TCP), IETF RFC 793, September 1981
6. CommunityBackup, Reinforcement Learning for Usage based File Selection: https://sourceforge.net/projects/autofileselection/ [code section]
7. CommunityBackup, Reinforcement Learning for P2P Cloud Backup Networks: https://sourceforge.net/projects/autopeerselect/ [code section]
8. Watson, R., Coyne, R.,"The Parallel I/O Architecture of the High-Performance Storage System (HPSS)", IEEE MS Symposium, 1995
9. Class lectures of CS533 course.
10. Reinforcement Learning in BitTorrent Systems; Rafit Izhak-Ratzin, Hyunggon Park and Mihaela van der Schaar
11. [Mitchell, 1997] T. M. Mitchell (1997). Machine Learning. McGraw-Hill.
12. [Sutton and Barto, 1998] R. S. Sutton and A. G. Barto (1998). Reinforcement Learning: An Introduction. The MIT Press.
13. Sayit, Muge Fesci; Kaymak, Yagiz; Teket, Kemal Deniz; Cetinkaya, Cihat; Demirci, Sercan; Kardas, Geylani, "Parent Selection via Reinforcement Learning in Mesh-Based P2P Video Streaming," *Information Technology: New Generations (ITNG), 2013 Tenth International Conference on* , vol., no., pp.546,551, 15-17 April 2013
14. A Survey of P2P Backup Networks, Bill Studer, Department of Computer Science, University of Colorado Boulder
15. Making Backup Cheap and Easy, Landon P. Cox, Christopher D. Murray, and Brian D. Noble, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor
16. A Cooperative Internet Backup Scheme, Mark Lillibridge Sameh Elnikety Andrew Birrell Mike Burrows, Michael Isard, HP Systems Research Center, Palo Alto, CA
17. https://sourceforge.net/projects/communitybackup/ [code section]
18. https://sourceforge.net/projects/autofileselection/ [code section] , Mall & Ahmed
19. https://sourceforge.net/projects/autopeerselect/ [code section]
20. www.iblocklist.com
21. Class lectures of CS533 course

APPENDIX I

Code-work used in this project

Please note that several modules have been lifted from the existing implementations of CBK.

Common modules:

1. **Qfunctionapprx.m**: This file implements the function approximation based q learning agent.
2. **Greedy.m**: This file implements the greedy agent.
3. **Common.py**: contains common code work, shared as a library.
4. **Config.py**: contains necessary configurations.
5. **EnvironmentSimulator**: Contains the asynchronous simulation server.
6. **Setup.py**: Contains the compilation script.
7. **scan_db.sqlite**: database file. Can be viewed by SQLite manager addon of firefox.
8. **ScanDB.py**: Contains the database interface.

Peer Selector:

1. **auto_peer_selection.py**: Q-Learning implementation of the Intelligent Data Agent
2. **auto_peer_selection_greedy.py**: Greedy implementation of the Data Agent
3. **PeerSelectionAgent.py**: Contains an abstract representation of the required units.
4. **Network.py**: Contains the network simulator used in the project.
5. **Mdp.py**: Contains a generic MDP process which is used in this project.

File Selector: (required from a previous implementation)

1. **mainmodule.m**: The mainmodule.m file contains the construction parameters of MDP(S,A,R,T).
2. **modelfreeqleaaarning.m**: This file implements the model free q learning agent.
3. **Backup.m**: Implements the backup action.
4. **Skip.m**: Implements the skip action.
5. **Scanfiles.m**: This file communicates with the simulator.
6. **StatRecorder.py**: Contains the file system statistics recorder.
7. **Agents.py**: Contains Value-Iteration algorithm.
8. **AutoFileSelection.py**: Contains the python code that uses value iteration to select files.

Adaptive Data Retrieval:

1. **iget.py:** Model Free Q-Learning implementation of the Adaptive Data Retrieval Agent
2. **iget_fixed_parallel_sockets.py:** Fixed-concurrency implementation of the Data Retrieval Agent
3. **iget_single_socket.py:** Single I/O thread implementation of the Data Retrieval Agent
4. **mdp.py:** Contains a generic MDP process which is used in this project.

APPENDIX II

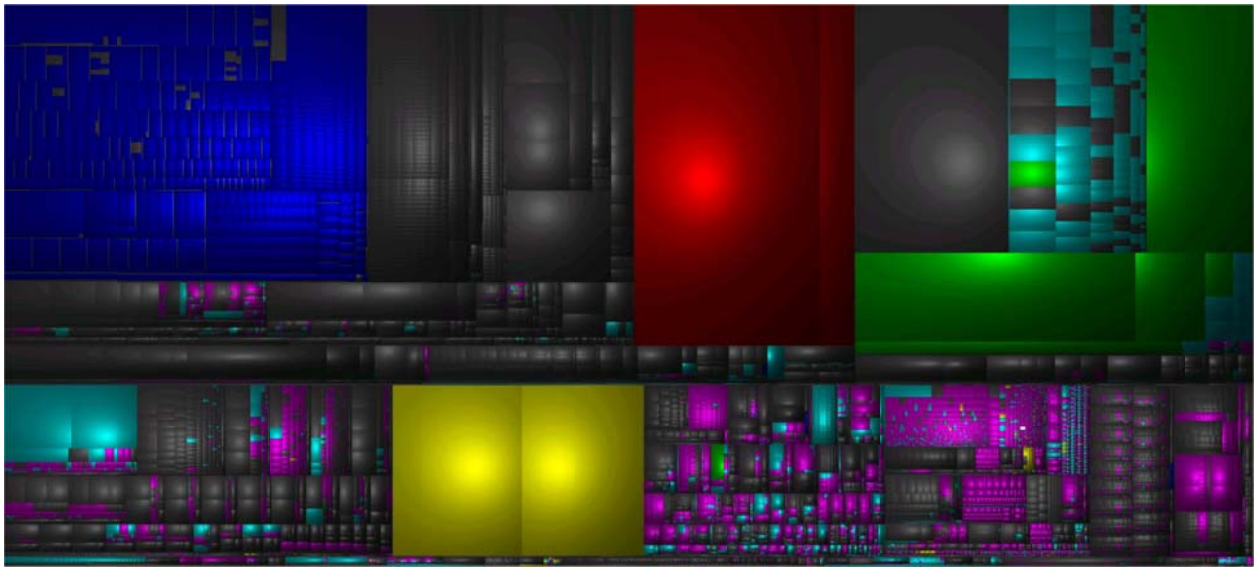
Feature 1:

File size / Average change in file size.

Assumptions:

- Zero length files usually are of no importance.
- Large files (> 100MB) usually are media, installation or database files and might not be of much importance.

Reasons for selecting this feature:



Above image is a disk map of an 80 GB hard-disk containing 200K files. Note that every 1mm rectangle in the map above is a large file occupying significant disk space. If we try to analyze and point out some rectangles which can be potential candidates for our important file selection, we obvious will never select the rectangles occupying the largest area. For example, red,yellow, green, turquoise and a few Grey ones are those files which we will simply neglect, as they are too large to be an important or user-generated file. This assumption might not be valid for several reasons but definitely is a good criteria for selecting files, and after-all, we have other features to compensate for those reasons. So, if we consider only file size, then we might be interested in pink, grey and blue files, or something else, which agent has to decide.

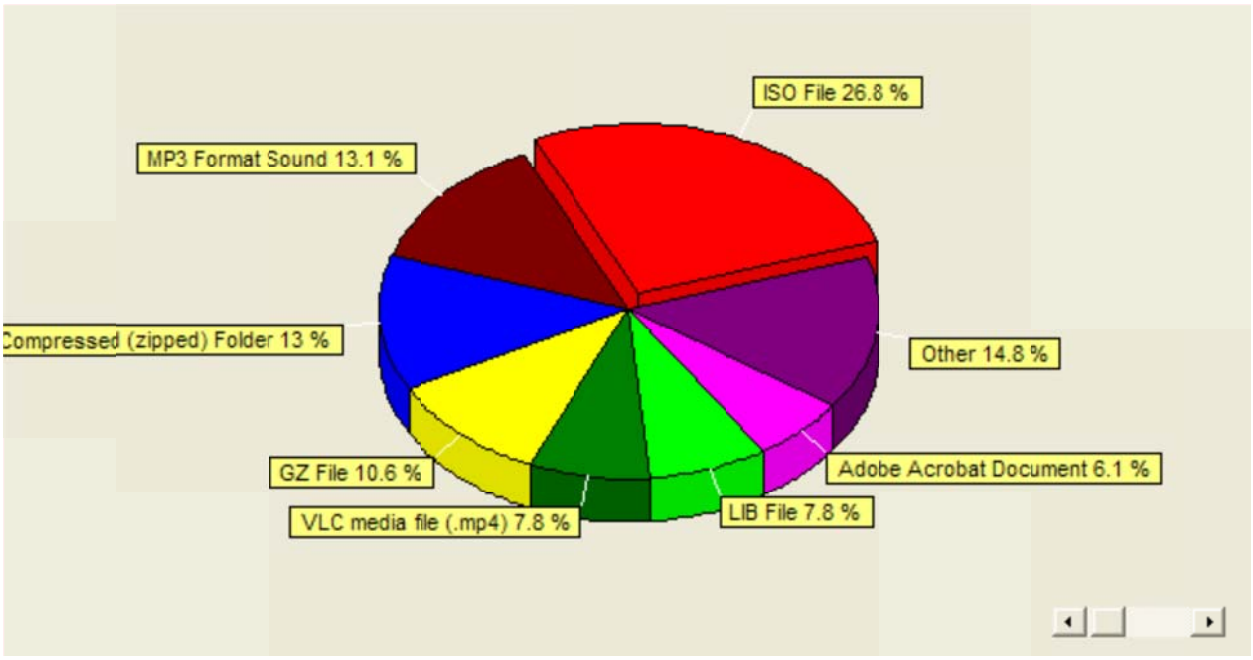
Feature 2:

File Type

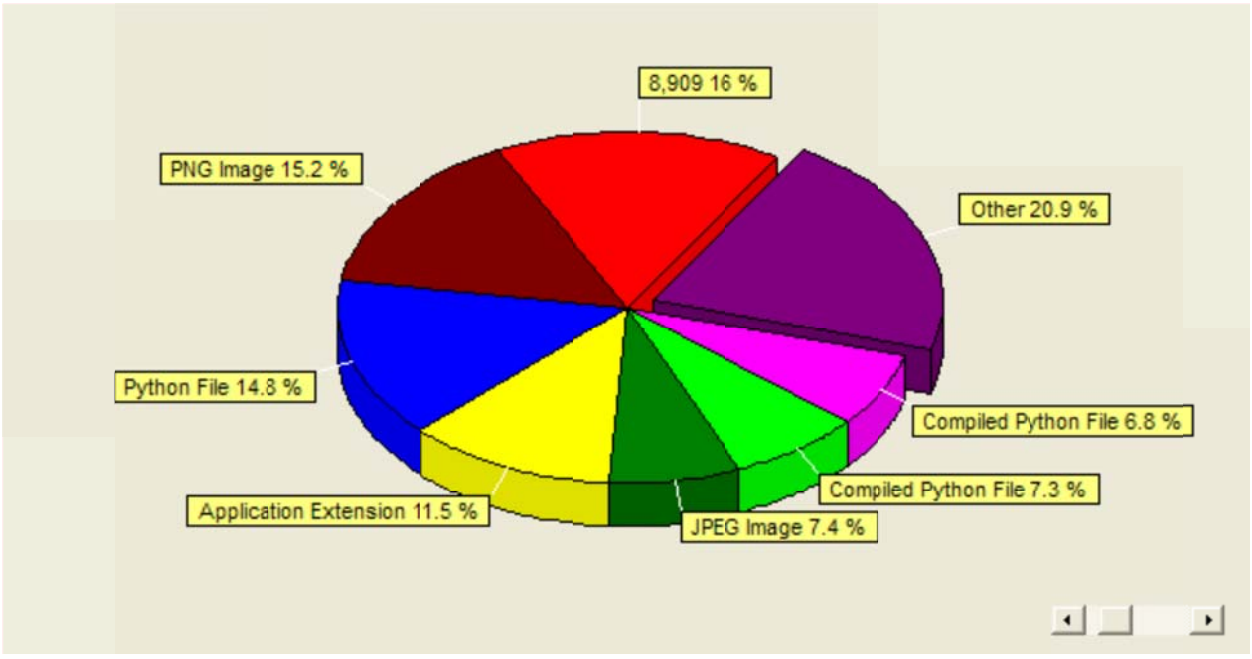
Assumptions:

- All system/executable file types score lower when compared to document, music, image or text types.

Reasons for selecting this feature:



The above image shows a file type to their total disk space consumption comparison. ISO files are disk images, so they have consumed 26.6% of the hard-disk. One might be interested in GZ files consuming 10.6% of the hard-disk, this has to be decided by the agent. Question is, what file size and file type combination gives better reward? Or, for that matter, any other feature combination.



This image above gives an idea of the density of each file type on the hard-disk. PNG images are present all over the disk with 15.2% popularity. This system contains a lot of python code-work too (14.8%). Obviously, we do not want all the PNG files, because, many will be icons and

cached images,but we do want most of the python files. This is where the file-type feature hallucinates between important and unwanted files.

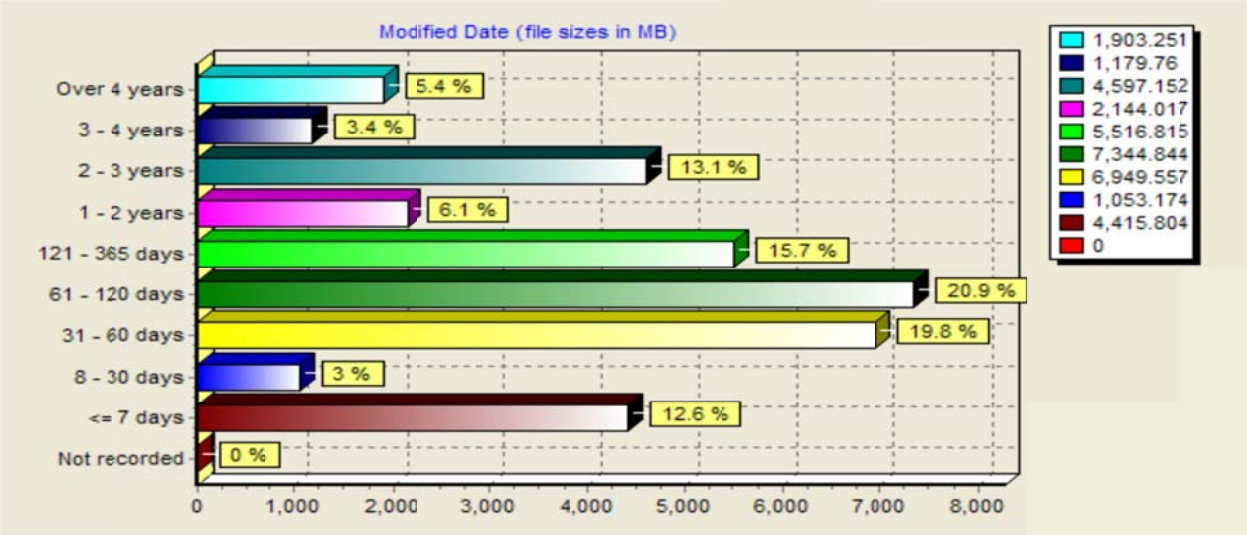
Feature 3:

Average modification interval:

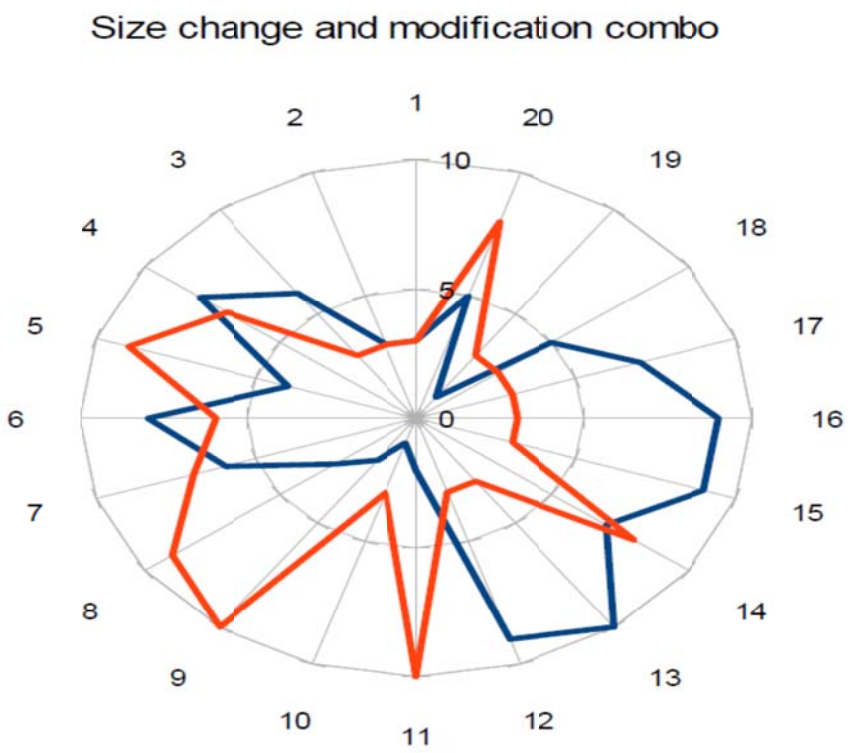
Assumptions:

- We assume that files which have been modified most number of times, recently, are the ones important to the user.

Reasons for selecting this feature:



This image above shows the modification time vs number of files comparison. Files which have been modified recently in past 7 days are the one which the agent should be concentrating more upon. 12.6% of files were modified in past 7 days, whereas 20.9% of files were modified in past 61-120 days. The agent has to decide what range of this feature value it should select to consider as an important file.



The net graph above compares the average modification interval with average size change of twenty files, over the scale of 0 to 10. This shows that the two features are entirely different. Some might complain that modification of a file brings change in the size, so they are similar features, but, it is true only for some files, but, not all of them.

Feature 4:

Average File Usage: Number of times, a file has been found loaded inside RAM.
Assumptions:

- We do not want to backup files which have never been opened by the user.

Feature 5:

Backup Times: Number of times, a file has been backed up.
Assumptions:

- We do not want to backup same files again and again.

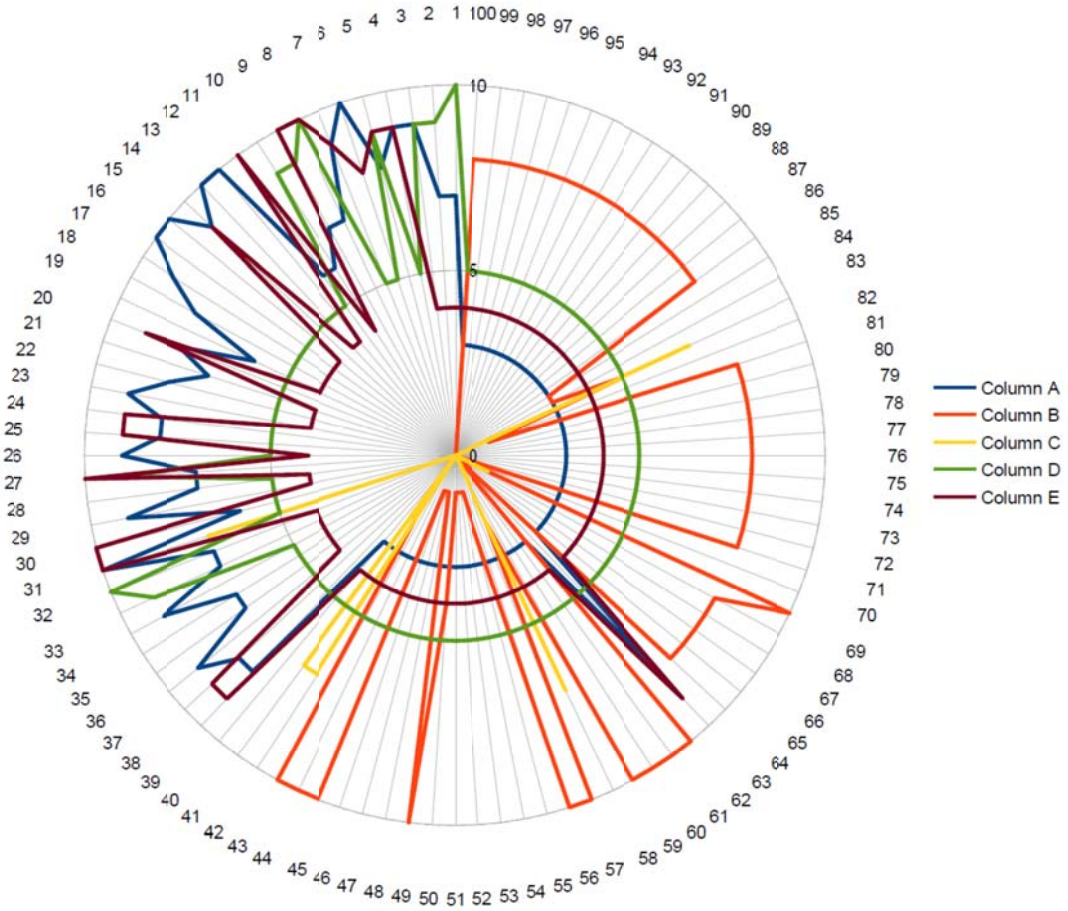


Figure: Comparison of all the features in a net graph for 100 files

Columns:

A: Average Size Change
B: Average File Usage
C: File Type Popularity Index
D: Number of backup copies
E: Average Modification Interval

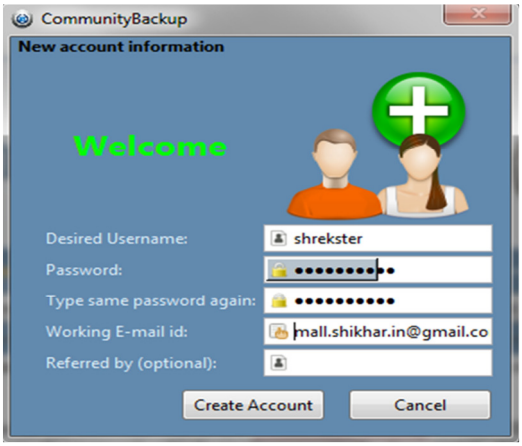
The net graph on previous page demonstrates the variety of inputs that we are able to provide to the agent. Average file size change (A) alters only for files ranging from 1-9 and 31-32, this feature lets outshine those files which have actually shown some data movement among the set of 100 files. As we can se that the modification interval too varies for these files, indicating the consistency of the statistics generated.

Another fact that becomes visible from the graph is that average file usage is either 0 or 10 for most of the files, this indicates that some (having a value of 10) files remained open for a very long interval hence ,should be selected as important files and must get backed up.

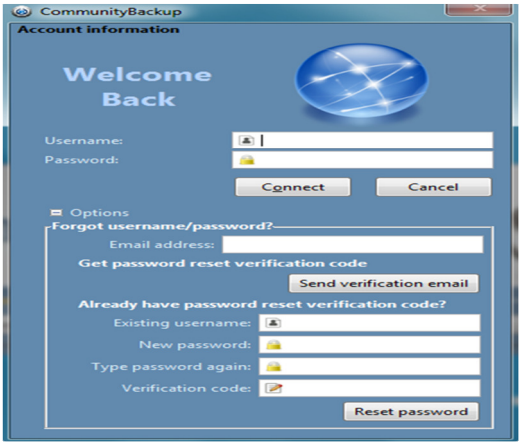
APPENDIX III
Live-project working demo

Window Screenshots

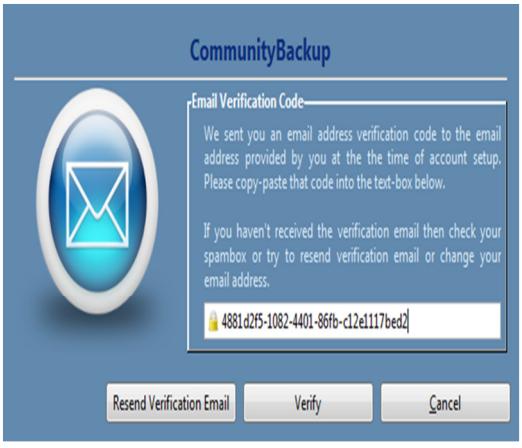
Startup flash screen. New/Existing account options.



New account creation

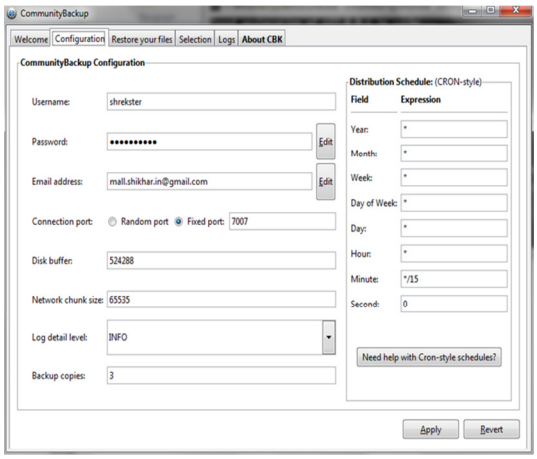


Existing account. Forgot password, recovery wizard

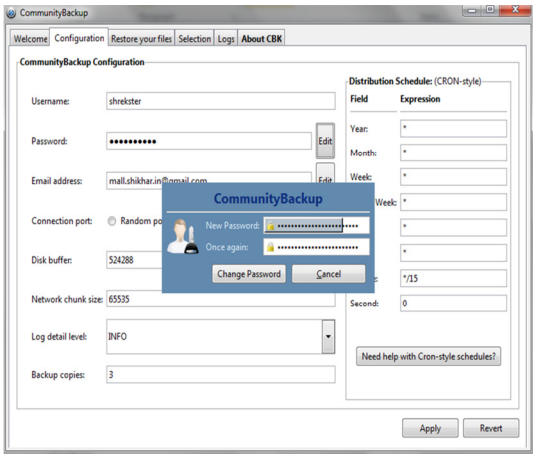


New account verification screen

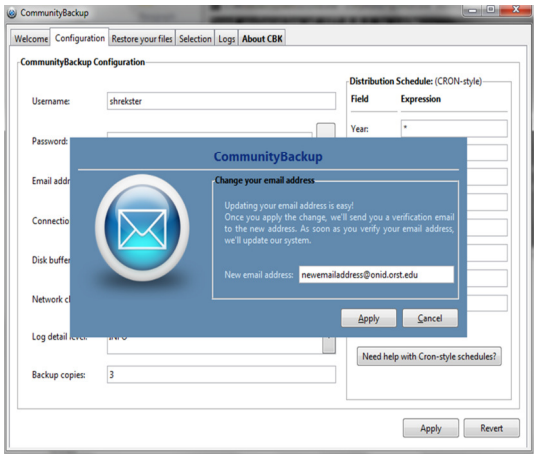
Front tab



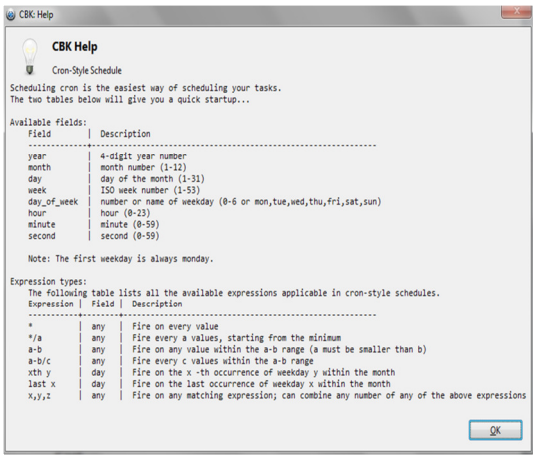
Backup configuration options. Server optons



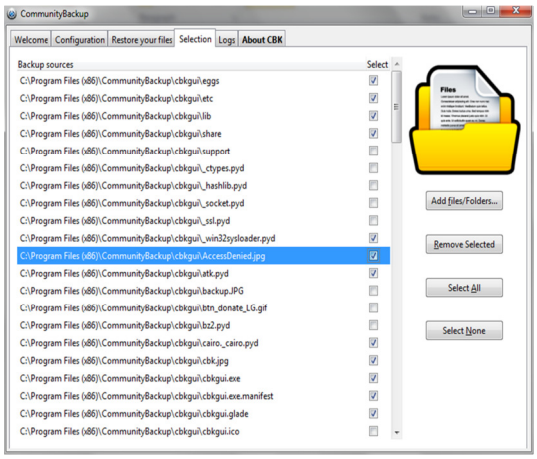
Password renewal screen



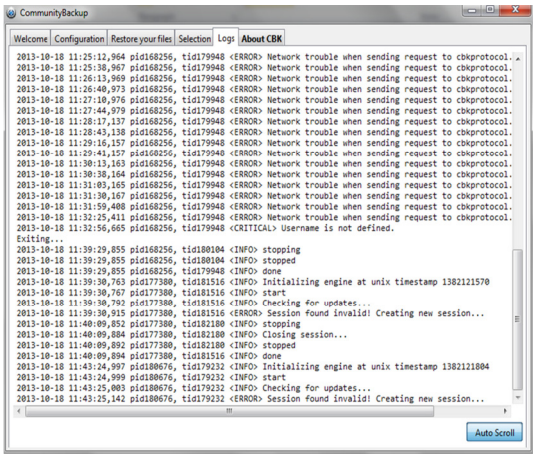
Email address renewal screen



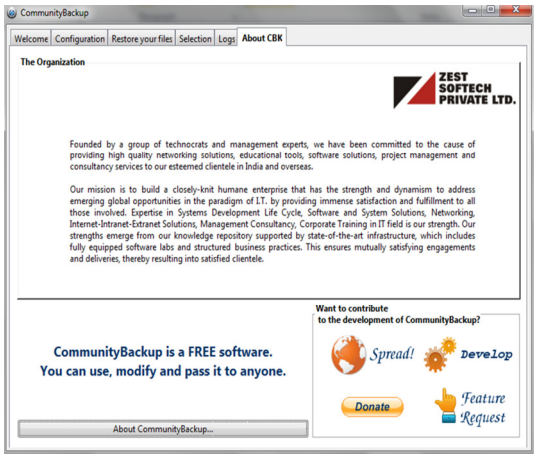
Scheduler help text



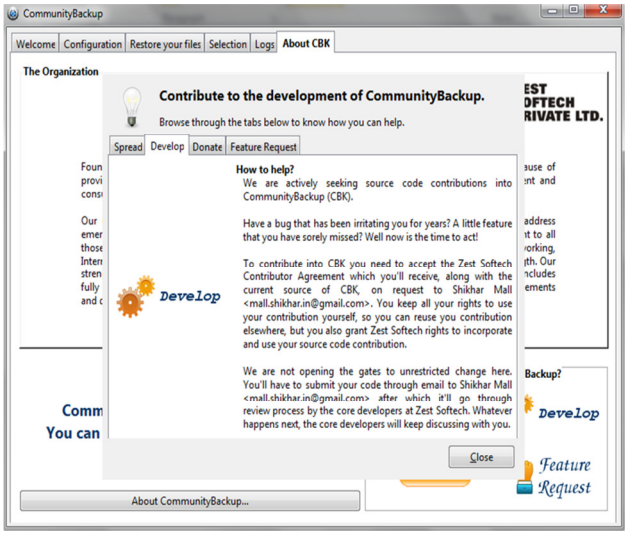
File selection engine



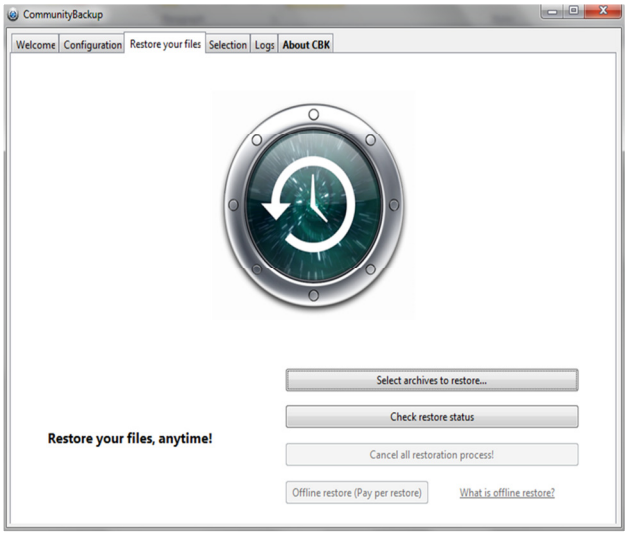
Backup engine log



About tab



References, credits and feedback window



Data recovery tab

APPENDIX IV

Usage instructions

Running Intelligent Peer Selector (first start the file selection engine)

Greedy:

Command: python auto_peer_selection_greedy.py <output file>

RL Agent:

Command: python auto_peer_selection.py <output file>

Running Intelligent File Selector

Starting the statistics collector:

Command: statRecorder.exe target location (ex: statRecorder C:\)

Starting the simulator:

Command: environmentSimulator.exe

Starting Value iteration:

Command: autoFileSelection.exe

Starting q-Learning:

Command: matlab -nojvm -nodesktop -r "mainmodule;quit;"

Using Adaptive Data Retrieval to download a file from internet:

RL Agent:

Command: python iget.py <source-url> <output file>

Single socket Agent:

Command: python iget_single_socket.py <source-url> <output file>

Fixed-Parallel Sockets Agent

Command: python get_fixed_parallel_sockets.py <source-url> <output file>