IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510157

Downfeather: A Machine-Learning-Driven Adaptive Multi-Connection Download Manager

Jatin Verma[1], Abhishek Jatla[2], Thanuja N BS[3], Dr. Beenarani Manoj[4]

Department of Computer Science and Engineering, SRM Institute of Science and Technology, Vadapalani Campus, Chennai, India[1-4]

Abstract: This paper presents Downfeather, a download manager that keeps learning while a download runs. A file is split into HTTP Range segments and fetched over several connections, and the manager tunes how much each connection reads per call using only the speeds it measures. Development took four iterations. Six recorded traces were clustered with K-Means and mined with Apriori to give a first speed governor; because that governor rested on little data and could only slow a connection down, an online gradient "brain" replaced it. Later versions added exploration noise and finally a saved memory of how speed changed after each of the brain's own moves. A real download of PixelMaster.exe (251.23 MB) over two connections finished in 43.1 s at 46.64 Mbps, with nearly equal shares (51.5% and 48.5%). A small online controller written with the Python standard library and NumPy can therefore cope with a noisy link without a pre-trained model.

Keywords: Adaptive Download Manager, Multi-Connection Download, Online Learning, Gradient Descent, K-Means, Association Rule Mining, Work Stealing, HTTP Range Requests

I. INTRODUCTION

People download large files every day: installers, disk images, datasets, software packages. A single HTTP connection often cannot use the full capacity of a modern link: servers may throttle each connection, one lost packet can stall a TCP stream, and the speed keeps changing from second to second. The download ends up slow and unsteady, and bandwidth goes unused.

A common remedy is segmented downloading, where the file is cut into byte ranges and fetched in parallel with HTTP Range requests [1]. Even so, most accelerators keep fixed settings: a set number of connections, a set read size and a static split. Once the link changes, those settings no longer fit, and the slowest connection decides when the download finishes.

This paper presents Downfeather, a download manager that borrows ideas from machine learning (ML) to tune itself during a download. K-Means was run on six recorded traces and Apriori mined the clusters for rules, which drove a first speed governor. Because the governor leaned on very little data, an online gradient "brain" replaced it and now adjusts the read size of each connection from the speed it measures. Two further iterations added exploration and a map of what the brain had seen, and then a memory of how speed changed after each increase or decrease, kept from run to run. The stages are described in order, since the reason behind each change explains the final design.

Downfeather is written in Python. The engine needs only the standard library and NumPy [2], and Matplotlib [3] draws the live plot. The program probes the server, pre-allocates a sparse file and gives byte-range segments to the connections; an idle connection takes half of the largest remaining segment (work stealing). The remaining sections cover the literature, architecture, methodology, source code, results and future work.

II. LITERATURE REVIEW

Download software has moved from fixed, single-stream transfers to adaptive ones that react to measurements. This section reviews the techniques the project draws on: parallel HTTP downloading, throughput modelling, clustering and rule mining, and online gradient-based adaptation.

A. Evolution of Download Acceleration

1) Single-Connection Transfers

The simplest download opens one connection and reads until the response ends. Its speed is capped by the slowest part of the path and by TCP behaviour such as slow start and loss recovery, and a single lost packet can stall the whole transfer.

2) Segmented and Parallel Downloading

HTTP lets a client request only a byte range of a file; a server that supports this replies with status 206 Partial Content and a Content-Range header [1], [4]. The file can then be fetched in pieces over several connections and written at the



International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026
DOI: 10.17148/IJARCCE.2026.1510157

right offsets. Lawrence [5] notes that this helps mainly when a host throttles each connection separately and on lossy networks, and lists the costs: extra handshakes, servers without Range support, connection limits and the disk effects of pre-allocating large sparse files. A manager should therefore decide for itself how to read and how to share work.

B. Machine Learning for Throughput Modelling

Schmid, Höss and Schuller [6] survey throughput prediction for wired and wireless networks, from simple smoothing filters to learned regressors. Their recurring message is that measurements are noisy and non-stationary, so an estimator must adapt quickly without overreacting. In the same spirit, Downfeather tracks each connection's speed with an exponentially weighted mean and variance, and uses the coefficient of variation (standard deviation divided by the mean) to show how unstable a connection is.

C. Clustering and Rule Mining for Control

K-Means splits observations into k groups by assigning each point to its nearest centroid and recomputing the centroids; it goes back to Lloyd [7], and k-means++ [8] gives better starting points. Association-rule mining finds rules of the form $\{A, B\} \rightarrow C$; the Apriori algorithm [9] builds frequent itemsets level by level and keeps rules whose support and confidence pass a threshold, while lift compares confidence with the base rate of the consequent. Rules ranked by confidence, then support and specificity, form classification-based-on-associations (CBA) lists. Together they turn raw traces into short, readable rules, which drove the first iteration of the controller.

D. Online Gradient-Based Adaptation

Stochastic approximation is the standard tool for tuning a parameter from noisy measurements of a system's own performance. Robbins and Monro [10] introduced updates with shrinking step sizes, and Spall [11] showed that a gradient can be estimated from performance under small random changes to the parameters. The least-mean-squares (LMS) rule of Widrow and Hoff [12] fits a linear model online by correcting it in proportion to its error. Later iterations of Downfeather combine these ideas: each connection nudges its read size with small noise, fits utility with an LMS update and takes the fitted slope as the gradient.

E. Challenges and Emerging Trends

Noisy, non-stationary links: speed changes every second, so a controller needs smoothing, a stability measure and small steps.

Server behaviour: some servers ignore Range headers, throttle or close connections early, so the manager must fall back safely.

Small training data: six traces reveal structure but give no statistical guarantee, the main reason for moving from fixed rules to online learning.

Disk and system effects: pre-allocating very large files can cause stalls unless the file is sparse [5].

Interpretability: controllers with a handful of parameters are easier to inspect and debug than large learned models.

F. Research Gap and Project Contribution

Parallel downloading, throughput prediction and stochastic optimisation are usually treated apart. This project brings them together in one small tool and records how each iteration fixed a problem left by the one before. Its contributions are:

1) From data to learner: K-Means and Apriori on six traces give a variance budget and rules for choosing the number of connections, and the paper explains why this controller was replaced.

2) A per-connection online gradient learner that tunes read size from speed, stability and overhead without a pre-trained model, refined with exploration noise and an LMS fit.

3) A gradient memory (the measured effect of earlier read-size increases and decreases) stored in a small binary file and reused in later runs.

4) Testing of the final iteration on a real download, with a live plot and text output showing speed, data used and gradient of each connection.

Every building block already exists on its own: byte-range downloading, throughput estimators, rule mining and stochastic approximation. What is missing is a compact tool that joins them and stays small enough to inspect, which Downfeather aims to provide.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510157

III. SYSTEM ARCHITECTURE AND DESIGN

The adaptive download manager has four parts: probing, segment scheduling, per-connection learning and live monitoring. This section examines the limits of fixed-setting downloaders, presents the proposed architecture and outlines its advantages.

A. Existing System Architecture

Conventional downloaders rely on static choices made before the transfer starts, which is why an adaptive design was chosen.

1) Fixed-Split Download Accelerators

A typical accelerator divides the file into equal parts, one per connection, and reads each with a constant buffer size. Three weaknesses follow.

Static per-connection split: a slow connection decides the finish time while the others sit idle.

Fixed read size: one size cannot suit both a steady and a noisy link; too small wastes calls and headers, too large reacts slowly.

Repeated handshakes: opening a new connection for every resume or request adds TLS and header overhead.

2) The Bottleneck of Instability

The biggest weakness is speed instability. With no feedback, the manager cannot tell whether a change in settings helped or hurt. The proposed architecture measures speed, spread and overhead on every connection and adapts continuously.

B. Proposed Model Architecture

The final system is a pipeline of a probe, a shared segment table, one worker per connection, a learning “Brain” and a live monitor (Fig. 1). The probe sets the mode: if the server accepts Range requests, workers draw from the shared segment table; if not, a single plain download is used. Each worker passes speed samples to the Brain, which returns the next chunk size, and the monitor reads the same shared state to draw the live plot. Because the parts are separate, the learning rule can be swapped without touching the download engine.

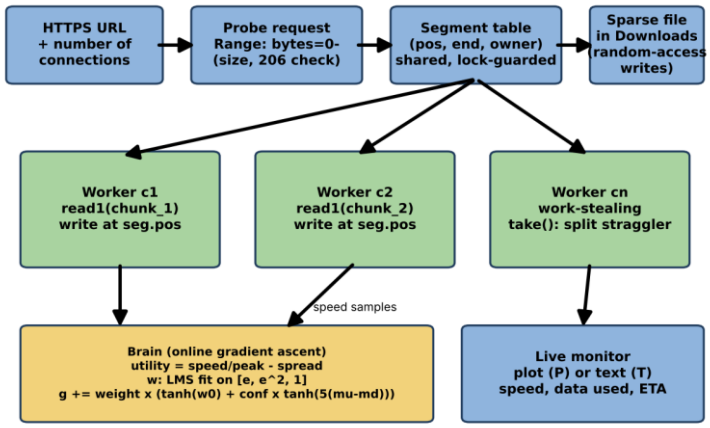


Fig. 1 Proposed system architecture of Downfeather.

1) Probe and Segment Module

The first request carries the header Range: bytes=0- and is itself the start of the download, so no request is wasted. If the server replies with status 206 and a known size, a sparse output file is created and the work is split into one segment per connection; otherwise the manager falls back to a plain download. Each segment records its position, end and owner. When a connection finishes, take() hands it an orphaned segment or half of the largest remaining one, so the fastest connection ends up doing more of the work.

2) The Learning Core (Brain)

Each connection reads chunks of size 65536 × exp(tanh(g + e)) bytes, where g is a learned value and e is small exploration noise; since tanh is bounded, the chunk stays between about 23.5 KiB and 174 KiB. Speed is measured over windows of at least 50 ms and smoothed. The brain computes a utility (speed relative to its peak, minus the relative spread), and an

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510157

LMS update fits utility ≈ w·[e, e², 1], whose slope w_0 is the estimated gradient. In the final iteration this slope is combined with the gradient memory, the average change in relative speed after earlier increases and decreases of g.

3) Persistent Memory and Output

At regular intervals the brain saves its state (weights, utility, counts, learned g per connection and the four gradient-memory numbers) in a 42-value NumPy file, brain.npy, written atomically so a crash cannot corrupt it. The next run starts from the learned values. The monitor offers a live plot of per-connection speed and data used, or a text line with progress, total speed, ETA and each connection's speed and gradient.

C. Advantages of the Proposed Architecture

1) Adaptive Speed and Stability

Each connection's read size is tuned from its own measurements, and the utility penalises unstable speed, so behaviour favours steady throughput over short-lived peaks.

2) Efficient Use of Connections

Work stealing removes idle connections at the end of a download, a keep-alive read loop with range resume avoids repeated handshakes, and failed segments return to the pool for another connection.

3) Lightweight and Interpretable

The controller is a few dozen numbers and short formulas. It needs no GPU, no training data at run time and no external service, and every decision can be printed and inspected.

4) Foundation for Future Enhancement

The learner state lives in a small file and the segment engine does not depend on the learning rule, so new controllers, more connections, resume support or a graphical interface can be added without redesign.

In sum, the architecture swaps fixed settings for a few cheap measured quantities, and work stealing keeps every connection busy until the last byte.

IV. METHODOLOGY

Downfeather was built in steps, each removing a limit of the one before. Fig. 2 shows the flow: recorded traces, knowledge extraction, controller design, a real download and its evaluation.

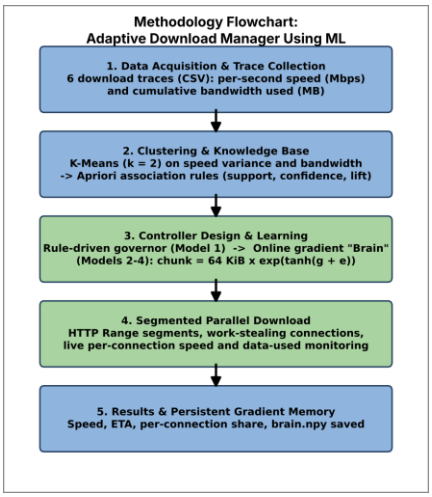


Fig. 2 Flowchart of the methodology.

A. Data Acquisition and Trace Analysis

Six download traces were recorded as CSV files, each holding the file size, a per-second speed series in Mbps and the cumulative bandwidth used; sizes range from 32.77 MB to 586.93 MB. For every trace the average speed, speed variance and final bandwidth were extracted. K-Means with k = 2 was run on (speed variance, bandwidth used), both divided by their range, starting from the first k points and stopping after 30 passes or when the centroids stop moving.

B. Knowledge Base and Rule Mining

Each trace became a transaction of items: SPD_HIGH/SPD_LOW (against the median speed, 32.69 Mbps), VAR_HIGH/VAR_LOW (against 96.40 (Mbps)², the midpoint of the centroid variances), BW_HIGH/BW_LOW

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510157

(against 294.17 MB, the midpoint of the centroid bandwidths) and the cluster label. Apriori mined itemsets of up to three items with minimum support 2/6 and minimum confidence 0.60, ranked by confidence, support and antecedent length; a rule engine returns P(VAR_LOW | context) from the best matching rule, or 0.5 if none matches.

Table I lists the main statements of this step.

TABLE I
KNOWLEDGE BASE AND RULE MINING.

Code element	What it does
OBSERVATIONS / CENTROIDS	The six traces with size, speed, variance, bandwidth, cluster and the two centroids, (150.50, 437.91) and (42.30, 150.42).
Thresholds	Cut-offs for high/low labels: variance 96.40, bandwidth 294.17 MB and speed 32.69 Mbps (median).
TRANSACTIONS	Turns each trace into a set of labels such as {SPD_HIGH, VAR_HIGH, BW_LOW, CLUSTER_1}.
mine_rules()	Runs Apriori on label sets of up to three items present in at least 2 of 6 traces; keeps rules with confidence ≥ 0.60 and computes lift.
RuleEngine	p_var_low() returns the chance that variance is low using the best matching rule, or 0.5 if none applies.

C. Mined Decision Rules

Apriori produced 40 rules from the six transactions. Table II shows those whose antecedent uses only speed and bandwidth items and whose consequent is the variance class, which the governor of iteration 1 could use before a transfer started. The rule {BW_HIGH} → VAR_HIGH (confidence 1.00) says large transfers were always unstable, so that governor gave them a tighter variance budget and more connections.

TABLE II
DECISION RULES MINED BY APRIORI.

Rule (if → then)	Support	Confidence	Lift	Meaning
{BW_LOW, SPD_LOW} → VAR_LOW	0.33	1.00	2.00	Small, slow transfers are steady
{BW_HIGH} → VAR_HIGH	0.33	1.00	2.00	Large transfers are unstable
{BW_LOW} → VAR_LOW	0.50	0.75	1.50	Small transfers are mostly steady
{SPD_HIGH} → VAR_HIGH	0.33	0.67	1.33	Fast transfers are often unstable
{SPD_LOW} → VAR_LOW	0.33	0.67	1.33	Slow transfers are often steady

With only six transactions the support values are coarse (0.33 is two traces, 0.50 three), so the rules are a readable summary of the traces rather than statistical evidence, which fits the decision to replace the governor with an online learner.

D. How the Design Evolved

The K-Means and Apriori work described the traces but was not yet a download manager. Four iterations followed, each keeping what worked and changing the one thing holding it back. Fig. 3 shows the path, and Table III compares the iterations feature by feature.

1) Iteration 1: A Governor Driven by Rules

The first downloader was built straight from the trace analysis. It starts uncapped and adjusts one shared speed limit every 0.5 s. The rules set a variance budget $B = V_2 + P(\text{VAR_LOW}) \cdot (96.40 - V_2)$, where $V_2 = 42.30$; if the measured variance exceeds it, the limit is cut by $1/\sqrt{(v/B)}$, never more than 15% per step, and if it sits well inside, the limit is raised slowly and eventually lifted. The rules also choose the number of connections (at most 3), and idle connections pull 8 MiB blocks from a shared queue. Every decision traced back to a readable rule, but six traces are too few to lean on, and a speed limit can only hold a connection back, never make it faster.

2) Iteration 2: Learning the Read Size

The second version dropped the rules and asked how much each connection should read per call. Every connection gets its own value g, and a small learner with weights [1, −1, −1] moves g to improve a utility of speed relative to peak minus instability (CV) and overhead. The file was cut into byte-range segments, idle connections took half of the largest

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510157

remaining one, and the learned values were saved in brain.json with a side file for resuming. This is where the project turned from rules to learning, but the brain only reacted and could not tell whether a speed change was its own doing.

3) Iteration 3: Exploration and a Map of What Was Seen

To separate its effect from the network's, the third version adds small random noise e to every read size and fits utility ≈ w·[e, e², 1] with an LMS update, making the slope a real gradient as in perturbation-based stochastic approximation. The first read after a new request is ignored because it contains server think time, connection set-up latency is learned so work stealing is skipped when it would not pay off, and g is remembered per connection number. A 24-bin landscape of the utility seen at each read size lets the brain move towards its remembered best; the saved memory grew to 102 numbers in brain.npy.

4) Iteration 4: Remembering Consequences (Final)

The landscape recorded where the utility had been, but not what followed the brain's own moves. The final version replaces it, and the run history, with the gradient memory: the average change in relative speed after an increase of g and after a decrease. Speed is measured over windows of at least 50 ms, the overhead term leaves the utility (utility = speed/peak − spread), and the saved state shrinks to 42 numbers. A live Matplotlib plot replaced the text graph, the text mode became one live line, and the resume side file and multi-line dashboard were dropped to keep the program small.

Table III shows that every iteration answered a specific weakness. The rule governor could only slow connections down, so the first learner used a read size that moves both ways; it could not separate its effect from noise, so exploration was added; and the landscape was finally replaced by a smaller memory of consequences.

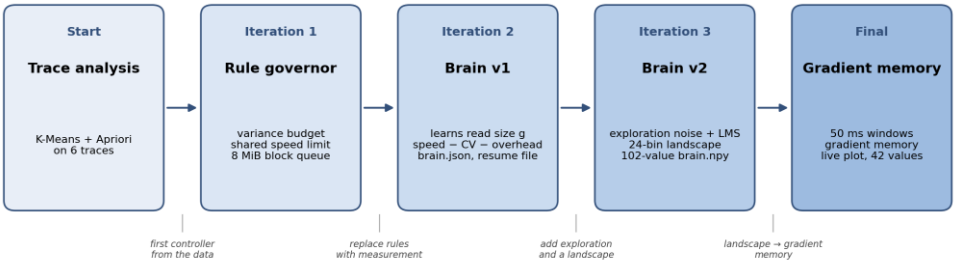


Fig. 3 Evolution of Downfeather from trace analysis to the final iteration.

TABLE III
FEATURE-BY-FEATURE TRACE OF THE FOUR ITERATIONS.

Aspect	Iteration 1	Iteration 2	Iteration 3	Iteration 4 (final)
Speed measured	Averaged over the last 1 s, sampled every 0.5 s	Taken from every single read	Every read, ignoring the first after each request	Windows of at least 50 ms, ignoring the first read
What is adjusted	One speed cap shared by all connections	The read size g of each connection	Read size plus random exploration change e	Read size, e and a step from the gradient memory
Learning signal	Variance budget derived from the Apriori rules	Utility: speed relative to peak, minus instability (CV), minus overhead	LMS slope of utility against e, plus best utility-map point	Same slope, plus speed gained after earlier changes
Memory kept	None; the rules were fixed from six traces	brain.json: weights, one shared g, counts	brain.npy: 102 values (g, 24-bin map, last 16 runs)	brain.npy: 42 values (g and gradient memory)
Resume	Partial file kept after an interruption	Side file holding the segment table	Side file marked "Incomplete (resumable)"	Not supported; listed as future work

E. Evaluation Protocol

The final iteration was tested on a real HTTPS download of PixelMaster.exe over two connections. Reported are completion time, average speed, the share of data per connection, per-connection speed and cumulative data over time (from the live plot), the gradient memory printed at the end, and a check of the saved file.

V. SOURCE CODE

This section follows the order in which the project grew: trace clustering, the rule governor of iteration 1, and the gradient-learning download manager with its engine. The rule-mining code is covered in Section IV-B, and iterations 2 and 3 are not listed separately because their ideas live on in the final code.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510157

A. Trace Clustering (K-Means)

The first segment reads the six CSV traces, computes the features and groups them with a hand-written K-Means (Table IV).

TABLE IV
TRACE CLUSTERING (K-MEANS).

Code element	What it does
parse_bw()	Converts bandwidth text such as "228.12 MB" or "1.2 GB" into megabytes (1 GB = 1024 MB).
manual_kmeans()	Scales speed variance and bandwidth by their range, starts from the first k points and assigns each trace to its nearest centroid.
Update loop	Recomputes each centroid as the mean of its cluster. Stops after 30 passes or when the centroids stop moving.

B. Rule-Driven Governor

Iteration 1 uses the rules to set a variance budget and adjusts a speed limit smoothly (Table V).

TABLE V
RULE-DRIVEN GOVERNOR.

Code element	What it does
p_low() / budget()	Estimates the chance of low variance for fast and slow transfers and converts it into the budget $B = V_2 + P\cdot(VAR_THR - V_2)$.
Decrease	If variance exceeds the budget ($\rho > 1$), the cap is multiplied by $\max(0.85, 1/\sqrt{\rho})$.
Increase / release	If $\rho < 0.7$ the cap grows 1–5% per tick, and is removed once it no longer limits speed.
choose_connections()	Opens all allowed connections (hard limit of 3) when the rules predict high variance, and one connection otherwise.

C. Gradient Brain (Final Model)

The final iteration learns each connection's read size from its own speed. The per-connection read size dates from iteration 2, exploration and the LMS fit from iteration 3, and time windows and gradient memory are new in iteration 4 (Table VI).

TABLE VI
GRADIENT BRAIN (FINAL MODEL).

Code element	What it does
Utility	util = mean/peak − spread, where spread is the standard deviation divided by the mean. A high, steady speed gives a high utility.
Gradient memory	Running averages mu and md of the change in relative speed after an increase and after a decrease of the read-size parameter g.
LMS update	Fits utility to the exploration change e with probe = [e, e^2, 1].
Gradient step	$dg = weight\cdot(\tanh(w_0) + conf\cdot\tanh(5(mu - md)))$. The parameter g is increased by dg.
Exploration	The next exploration change is e = (spread + weight)/(1 + spread + weight) × N(0,1), so noisier links are probed more widely.

D. Segmented Work-Stealing Download

The engine divides the file into segments and lets idle connections help the slowest one (Table VII).

TABLE VII
SEGMENTED WORK-STEALING DOWNLOAD.

Code element	What it does
take()	Gives an idle connection an unclaimed segment or half of the largest remaining one (above 256 KiB).
Range handling	Requests bytes pos–end; any reply other than 206 is an error, so a full body is never written at the wrong offset.
Retries	After more than 6 failures in a row, the segment is handed to another connection.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510157

Code element	What it does
No Range support	If the server cannot serve ranges, one connection downloads the whole file and restarts from byte 0 after an error.

E. Brain Initialisation and Persistent Memory

The learned state is loaded at the start of a run and saved during and after the download (Table VIII).

TABLE VIII
BRAIN INITIALISATION AND PERSISTENT MEMORY.

Code element	What it does
Load	Reads brain.npy at start-up; a missing or invalid file starts every value at zero.
Layout	The 42 values: weights [0:3], utility, run and sample counts, g per slot [6:38], gradient memory [38:42].
Atomic replace	os.replace() swaps the temporary file into place, so an interruption cannot leave a corrupted brain.npy.

F. Live Monitoring and Output

The last block handles what the user sees and produces the results used in the next section (Table IX).

TABLE IX
LIVE MONITORING AND OUTPUT.

Code element	What it does
render()	Smooths the speed of each connection over time and computes the total speed, the progress and the estimated time remaining (ETA).
Text mode	Prints progress, total speed in Mbps, ETA, and the speed and gradient of each connection on one live line.
Final summary	Prints the saved file path, elapsed time, average speed, the gradient memory, and the data and share of each connection.

VI. RESULTS AND DISCUSSION

This section reports the structure in the six traces, the mined rules and the behaviour of the final iteration on a real download. All numbers come from the program output and screenshots taken during the project.

A. Overall Model Performance

The final iteration downloaded PixelMaster.exe over two connections: 251.23 MB (129.36 MB on c1, 121.87 MB on c2) reached the Downloads folder in 43.1 s at 46.64 Mbps. The near-equal shares (51.5% and 48.5%) suggest that work stealing and per-connection adaptation kept both connections busy.

B. Trace Clustering Results

Table X lists the six traces and their clusters. Cluster 1 groups the three large files (228–587 MB) whose speed varies strongly, with centroid variance 150.50 $(\text{Mbps})^2$ and bandwidth 437.91 MB; Cluster 2 groups the smaller or steadier transfers, with centroid variance 42.30 $(\text{Mbps})^2$ and bandwidth 150.42 MB. Fig. 4 shows both groups and their centroids.

TABLE X
CLUSTERING RESULTS FOR THE SIX DOWNLOAD TRACES.

File	Size (MB)	Avg speed (Mbps)	Variance $(\text{Mbps})^2$	Bandwidth used (MB)	Cluster
arkime-6.7.0-1.el8.x86_64.rpm	228.12	36.21	136.57	228.12	1
prebuilt_linux_x64_v8.zip	498.68	29.96	117.02	498.68	1
QGIS-OSGeo4W-4.2.3-1.msi	586.93	37.22	197.92	586.93	1
download_metrics.csv	209.19	12.60	16.42	209.19	2
KeePassXC-2.7.12-arm64.dmg	32.77	30.25	43.78	32.77	2
Scrypted-0.147.0.Setup.exe	209.29	35.13	66.71	209.29	2

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510157

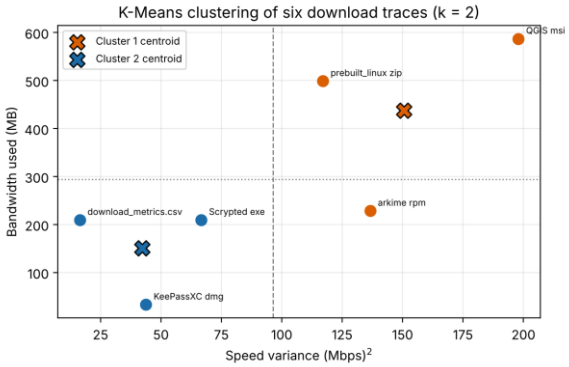


Fig. 4 K-Means clustering of the six download traces.

Cluster 1 files are the largest and least steady (variances 117 to 198 (Mbps)²), while every Cluster 2 file stays below 67 (Mbps)². Low speed alone therefore does not indicate an unstable link, whereas a large transfer did in these traces.

C. Final Model Download Performance

Table XI summarises the test download: file, connections, completion time, average speed, per-connection shares and the two gradient-memory values.

TABLE XI
PERFORMANCE OF THE FINAL MODEL ON THE TEST DOWNLOAD.

Measure	Value
File downloaded	PixelMaster.exe (HTTPS download, GitHub release)
Connections used	2
Total data	251.23 MB (c1: 129.36 MB, c2: 121.87 MB)
Completion time	43.1 s
Average speed	46.64 Mbps
Share per connection	c1 51.5%, c2 48.5%
Gradient memory after a read-size increase (mu)	−0.0001
Gradient memory after a read-size decrease (md)	−0.0927
Text-mode line at 13.8% progress	Total 38.28 Mbps, ETA 42 s; c1 22.1 Mbps (gradient +0.018), c2 16.2 Mbps (gradient +0.019)

Fig. 5 is the live plot captured during the download. Both connections climb to roughly 25–30 Mbps within about two seconds, then move between 22 and 37 Mbps with the two curves rising and falling together, while the cumulative data grows almost steadily.

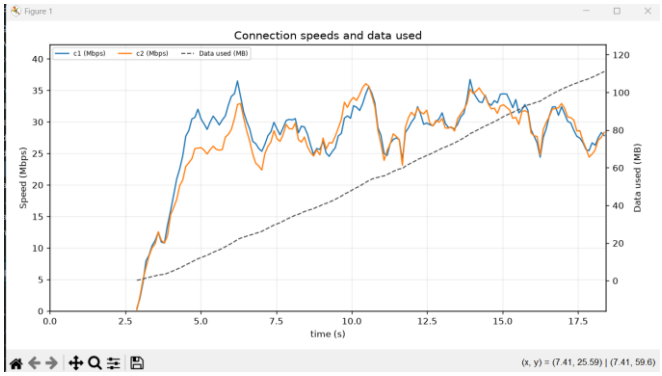


Fig. 5 Live plot of connection speeds and data used.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510157

Fig. 6 shows the text-mode output at 13.8% progress: both connections already show small positive gradients (+0.018 and +0.019), meaning the brain was increasing both read sizes.

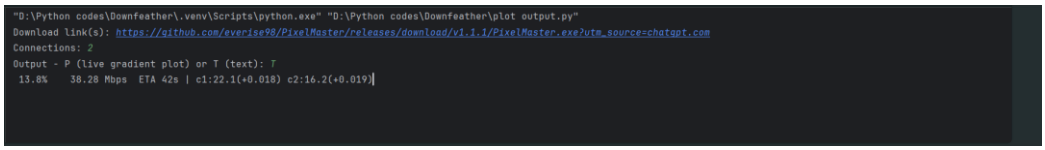


Fig. 6 Text output of the final model run.

D. Discussion of Results

1) Speed and Balance

The 46.64 Mbps average includes the start-up period; once running, each connection holds roughly 25–36 Mbps, and the 51.5% / 48.5% split suggests neither was left idle.

2) Gradient Memory

The memory after an increase is almost zero (−0.0001), so larger reads did not hurt relative speed, while the memory after a decrease is clearly negative (−0.0927), so smaller reads did. The gradient step therefore pushes g upward, and this is saved in brain.npy for the next run.

3) Limitations

The results come from one real download and six traces, so they show system behaviour but are not a statistical comparison; a comparison of all four iterations and a single-connection baseline are left as future work.

4) What Survived from Each Iteration

Penalising instability: from the variance budget of iteration 1, to the CV term of the utility in iteration 2, to the spread term of the final utility.

Sharing the work: the shared queue of 8 MiB blocks became byte-range segments that idle connections split in half, unchanged in the final engine.

Measuring speed: skipping the first read after a request and measuring over time windows both follow from the lesson that single reads say little about the link.

Memory: it shrank from 102 numbers to 42 when the landscape and run history were replaced by the gradient memory.

What was dropped: the rule base, the overhead term, the resume side file and the multi-line dashboard, to keep the program shorter.

Overall, the final iteration completed a real download with balanced connections and a gradient memory that favours larger reads; the evidence is a single download, so the findings are a demonstration, not a benchmark.

VII. CONCLUSION AND FUTURE WORK

This project shows that an adaptive download manager can be assembled from small, readable machine-learning parts, and that most of the effort goes into iteration. K-Means and Apriori turned six traces into a variance budget and a few rules, which gave the first governor; because it could only slow a connection down, an online gradient brain replaced it, tuning each connection's read size from measured speed, with exploration noise and an LMS fit first and a gradient memory of what followed each change last.

On a real download of PixelMaster.exe, the final iteration moved about 251.23 MB over two connections in 43.1 s at 46.64 Mbps with a 51.5% / 48.5% split. Its gradient memory showed that larger reads were safe and smaller reads lowered the relative speed.

Planned extensions are baseline comparisons against a single connection and a fixed read size on many files and servers, larger and more varied traces with k-means++ seeding, an adaptive number of connections, and resume, checksum verification and HTTP/2 or HTTP/3 support.

ABBREVIATIONS

ML: Machine Learning; ARM: Association Rule Mining; CBA: Classification Based on Associations; HTTP(S): Hypertext Transfer Protocol (Secure); TLS: Transport Layer Security; TCP: Transmission Control Protocol; CSV: Comma-Separated Values; LMS: Least Mean Squares; CV: Coefficient of Variation; Mbps: Megabits per second; MB: Megabyte (10^6 bytes); KiB / MiB: Kibibyte / Mebibyte; ETA: Estimated Time of Arrival; RFC: Request for Comments.



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510157

ACKNOWLEDGEMENT

We express our humble gratitude to our Honorable Chancellor, Pro Chancellors, Vice Chancellor, Registrar, Dean [FET] and Vice Principals of SRM Institute of Science and Technology for the facilities and support extended for this minor project work.

We thank the Head of the Department of Computer Science and Engineering, Vadapalani Campus, for valuable suggestions and encouragement, and our supervisor, Dr. Beenarani Manoj, Associate Professor, for her mentorship and for the freedom to explore topics of our interest. We also thank the faculty and students of the department, and our parents, family and friends, for their constant support.

REFERENCES

[1]. R. Fielding, M. Nottingham, and J. Reschke, "HTTP Semantics," RFC 9110, Internet Engineering Task Force, 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9110

[2]. C. R. Harris *et al.*, "Array Programming with NumPy," *Nature*, vol. 585, pp. 357–362, 2020.

[3]. J. D. Hunter, "Matplotlib: A 2D Graphics Environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[4]. Mozilla Developer Network, "206 Partial Content," *MDN Web Docs*. [Online]. Available: https://developer.mozilla.org/docs/Web/HTTP/Status/206

[5]. E. Lawrence, "Parallel Downloading," *text/plain* (blog), Nov. 22, 2024. [Online]. Available: https://textslashplain.com/2024/11/22/parallel-downloading/

[6]. J. Schmid, A. Höss, and B. W. Schuller, "A Survey on Client Throughput Prediction Algorithms in Wired and Wireless Networks," *ACM Computing Surveys*, vol. 54, no. 9, Art. no. 194, 2022.

[7]. S. P. Lloyd, "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[8]. D. Arthur and S. Vassilvitskii, "k-means++: The Advantages of Careful Seeding," in *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2007, pp. 1027–1035.

[9]. R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," in *Proc. 20th International Conference on Very Large Data Bases (VLDB)*, Santiago, Chile, 1994, pp. 487–499.

[10]. H. Robbins and S. Monro, "A Stochastic Approximation Method," *Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, 1951.

[11]. J. C. Spall, "Multivariate Stochastic Approximation Using a Simultaneous Perturbation Gradient Approximation," *IEEE Transactions on Automatic Control*, vol. 37, no. 3, pp. 332–341, 1992.

[12]. B. Widrow and M. E. Hoff, "Adaptive Switching Circuits," *IRE WESCON Convention Record*, Part 4, pp. 96–104, 1960.

 This work is licensed under a Creative Commons Attribution 4.0 International License