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 11, November 2026

NetResilience: Evaluation of Adaptive Recovery Strategies Under Simulated Network Failures

Miss Kajal P. Badgujar[1], Manoj V. Nikum[2], Mrs. S. N. Raul[3]

Research Scholar,MCA, Shri Jaykumar Rawal Institute of Technology, Dondaicha, India[1]

HOD MCA & Assistant Professor, Shri Jaykumar Rawal Institute of Technology, Dondaicha, India[2]

Assistant Professor, Shri Jaykumar Rawal Institute of Technology, Dondaicha, India[3]

Abstract: Network-dependent applications may experience delays, packet loss, temporary disconnections, and unavailable services. Fixed retry strategies may perform poorly under changing network conditions and can increase response time or create duplicate requests. This paper presents the design of NetResilience, a Python-based simulation system for evaluating recovery strategies under controlled network-failure conditions. The proposed system models network delay, request failure, packet loss, and temporary service unavailability. It compares fixed-delay and exponential-backoff retry strategies with an adaptive retry method that considers observed failures and response-time information. The evaluation framework records request-delivery rate, recovery time, end-to-end latency, duplicate-request count, and retry overhead under different simulated network conditions and workload levels. A web dashboard is proposed to configure experiments and display comparative results. The work is designed as a reproducible software simulation and does not require additional network hardware. Since implementation results have not been supplied for this draft, the paper defines the experimental procedure and evaluation measures without presenting fabricated numerical findings. The completed experiments will determine the conditions under which adaptive retry behaviour offers measurable benefits over baseline strategies.

Keywords: Network resilience, adaptive retry, exponential backoff, fault simulation, Python, latency, packet loss, recovery strategy.

I. INTRODUCTION

Modern software applications depend on networks and remote services to exchange data. A temporary network disturbance can cause a request to time out, fail, or take longer than expected. Examples include unstable Wi-Fi, congested links, overloaded servers, and short periods of service unavailability. Even when a failure is temporary, the application may appear slow or unavailable to its users. A common recovery technique is to retry a failed request. Retrying can help when a failure is temporary, but an unsuitable retry policy can make the situation worse. If many clients retry immediately or at the same fixed interval, they may send additional traffic to an already overloaded service. This can increase latency and delay recovery. Repeated operations may also create duplicate effects when the application does not handle repeated requests safely. NetResilience is proposed as a Python-based experimental simulator for comparing retry and recovery strategies. Instead of testing against a live production network, it creates controlled failure conditions so that the same workload can be evaluated repeatedly. The project focuses on three policies: fixed-delay retry, exponential-backoff retry, and an adaptive retry policy based on recent failure and response-time observations. The main contribution is an accessible, reproducible framework that allows a student or developer to observe how retry behaviour changes under different simulated conditions. The project is intended for experimentation and learning; it is not a replacement for production network monitoring or a claim that one retry policy is universally best.

II. LITERATURE REVIEW

Retransmission timing is an established concern in network protocol design. RFC 6298 describes how TCP computes its retransmission timeout using smoothed round-trip time and variation, and specifies exponential backoff when a retransmission timer expires [1]. This demonstrates why retry timing should respond to network behaviour rather than repeatedly transmitting at an aggressive fixed interval. The TCP specification in RFC 9293 explains that retransmission timeout behaviour must account for variability across networks and refers to the algorithm in RFC 6298 [2]. Although NetResilience operates at the application-request simulation level rather than implementing TCP, these standards provide important background for the use of timeouts and backoff.

A fixed-delay policy is straightforward to implement and interpret: after a failed attempt, the system waits for the same configured interval before trying again. Its simplicity is useful as a baseline, but it does not increase its waiting period when failures continue. Exponential backoff increases the delay after successive failures, commonly subject to a

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

maximum cap. This can reduce repeated pressure during a sustained outage, although it may increase the time before a later attempt is made. Adaptive recovery introduces an additional decision step. The retry interval or decision to retry can be adjusted using recent observations such as consecutive failures, measured response time, and the recent success rate. The specific adaptive rule must be defined, bounded, and tested; otherwise, the policy may become difficult to reproduce or compare. The proposed study addresses a practical educational gap: a compact application in which multiple retry policies are evaluated under identical, configurable failure scenarios and compared using common metrics. The project does not claim to introduce a new transport protocol. Its focus is the design and experimental comparison of application-level recovery strategies

III. PROBLEM STATEMENT AND OBJECTIVES

A. Problem Statement Applications often use retry logic to recover from temporary request failures, but a single fixed retry interval may not perform consistently under changing network conditions. Repeated retries can add unnecessary traffic, increase latency, and create duplicate requests. There is a need for a controlled environment in which fixed-delay, exponential-backoff, and adaptive retry strategies can be compared using consistent workloads and measurable outcomes. B. Objectives • Design a Python-based simulator that generates configurable request workloads. • Simulate network delay, packet loss, request failure, and temporary service unavailability. • Implement fixed-delay and exponential-backoff retry strategies as baseline methods. • Implement a bounded adaptive retry method using recent failure and response-time observations. • Measure request-delivery rate, recovery time, end-to-end latency, duplicate-request count, and retry overhead. • Provide a web dashboard for configuring experiments and reviewing comparative results. • Keep experiment settings and random seeds so that runs can be repeated and compared. C. Scope and Assumptions The project simulates request and service behaviour within a local software environment. It does not emulate every detail of a real router, wireless channel, TCP stack, or Internet route. Failure probabilities, delays, outage windows, request volume, retry limits, and timeout values are configurable model inputs. Results therefore describe the simulator and its assumptions; conclusions should not be generalized to all production networks without further validation.

IV. PROPOSED METHODOLOGY

The proposed methodology consists of workload generation, fault injection, retry-policy execution, metric collection, and result visualization. Each experiment begins with a defined workload and network profile. The same profile is then used for each retry strategy so that differences in outcomes can be attributed to policy behaviour rather than different test conditions.

A. WORKLOAD GENERATION

The workload generator creates a sequence of uniquely identified requests. Each request contains a request identifier, creation time, simulated service operation, and an idempotency or duplicate-detection key. The experiment can vary the number of requests, request arrival pattern, and concurrency level. Unique identifiers allow the simulator to distinguish a retry of the same logical request from a new request.

B. NETWORK FAILURE SIMULATION

A simulated network layer decides how each attempt is handled. It can add a configurable delay, randomly drop an attempt, return a failure response, or mark the service unavailable during a selected time interval. The simulator records the cause of each failed attempt. A seeded pseudo-random generator should be used for probabilistic failures so that an experiment can be reproduced.

C. Retry Strategies

1) Fixed-delay retry: after a failed attempt, wait for a constant interval d before the next attempt. The interval remains unchanged until the maximum attempt count is reached.

2) Exponential-backoff retry: after the nth failure, wait for a delay that increases exponentially, for example $d_n = \min(d_0 \times 2^{(n-1)}, d_{max})$. Here d_0 is the initial delay and d_{max} is the configured maximum delay. The cap prevents the wait from growing without bound. This is an application-level comparison inspired by the general backoff principle; it is not a claim to implement TCP's retransmission algorithm.

3) Adaptive retry: calculate a bounded delay from recent observations. A simple initial design can maintain a moving average of recent response times and a consecutive-failure counter. When failures rise or response times become high, the policy increases its wait; after stable successful responses, it gradually reduces the wait toward a configured minimum. The exact update equation, window size, minimum, maximum, and retry limit must be fixed before running the final experiment and documented in the implementation.

Request Safety and Duplicate Detection

A retry may cause the same logical operation to be submitted more than once, especially when a response is lost after the server has processed the request. NetResilience should therefore assign each logical request a stable identifier and

record how many times it reaches the simulated service. The duplicate-request metric counts repeated deliveries of the same logical operation. In a real application, idempotency keys or deduplication at the service boundary are important safeguards; the simulator makes this risk visible rather than assuming that every retry is harmless.

Experiment Execution

For each network profile, the simulator executes all three strategies using the same request count, workload pattern, failure configuration, retry limit, and random seed. Multiple independent seeds should be run to reduce the influence of one random sequence. The experiment log should include configuration values, timestamps, request identifiers, attempt number, outcome, selected wait interval, and measured response time. Results should be summarized per run and across repeated runs.

V. SYSTEM ARCHITECTURE AND MODULES

NetResilience is designed as a modular Python application. A suitable implementation can use Flask for the web interface, SQLite for local experiment records, and Matplotlib or Plotly for charts. These are proposed implementation choices and can be adjusted to the available development environment.

- User Interface / Dashboard: accepts experiment settings, starts simulations, and displays status, tables, and charts.
- Experiment Controller: validates settings, creates experiment runs, and coordinates the modules.
- Workload Generator: creates request events and controls request volume and concurrency.
- Network Simulator: injects delay, packet loss, request failures, and service-outage windows.
- Retry Engine: executes fixed-delay, exponential-backoff, and adaptive policies under a common interface.
- Metrics Collector: calculates delivery rate, recovery time, latency, duplicate deliveries, and retry overhead.
- Data Storage: saves experiment configurations, run summaries, and detailed event logs in SQLite.
- Report / Visualization Module: compares strategies across scenarios and exports results for analysis.

A typical flow is: the user configures a scenario in the dashboard; the controller validates and stores the configuration; the workload generator creates requests; the network simulator produces success or failure outcomes; the retry engine decides whether and when to attempt again; the metrics collector records outcomes; and the dashboard presents the comparison.

VI. EXPERIMENTAL SETUP AND EVALUATION METRICS

A. Experimental Variables

Independent variables include the retry strategy, request volume, simulated base delay, packet-loss probability, failure probability, outage duration, maximum attempts, and workload concurrency. Dependent variables are the performance metrics defined below. Controlled variables include the request sequence, service processing assumptions, timeout configuration, and random seed for a paired comparison.

B. Evaluation Metrics

- Request-delivery rate: successfully completed logical requests divided by total logical requests, expressed as a percentage.
- Recovery time: elapsed simulated time from the start of a temporary outage until the service or workload returns to the defined recovery condition.
- End-to-end latency: elapsed time from the first attempt of a logical request until success or final failure; report successful and failed requests separately where appropriate.
- Duplicate-request count: number of repeated deliveries of the same logical request identifier beyond its first delivery.
- Retry overhead: total retry attempts beyond initial attempts, and optionally the ratio of retry attempts to total attempts.

C. Proposed Test Scenarios

- Stable network: low delay and negligible failure, used to observe baseline overhead.
- Intermittent packet loss: configurable packet-loss probability with no long outage.
- High-latency network: elevated and variable response delay.
- Temporary service outage: service unavailable for a defined interval and then restored.
- Combined degraded conditions: delay, loss, and outage periods together.
- Load variation: repeat selected scenarios at low, medium, and high request volumes.

Each scenario should be repeated with multiple random seeds. The final report should include the mean and standard deviation, or median and interquartile range when distributions are skewed. Graphs should compare all strategies under the same scenario. The experiment should also record the simulator version and configuration so another person can reproduce the run.

VII. RESULTS AND DISCUSSION

NetResilience proposes a Python-based simulation framework for comparing application-level recovery strategies under controlled network failures. It brings workload generation, configurable fault injection, retry-policy execution, metric collection, and dashboard-based comparison into one reproducible environment. The design includes fixed-delay retry and exponential backoff as baselines and a bounded adaptive policy that uses observed failures and response-time information.

The project's value will come from a transparent experimental comparison rather than an assumption that adaptive retry is always better. By measuring request delivery, recovery time, latency, duplicate requests, and retry overhead under identical scenarios, the system can show the conditions in which each policy is useful. The conclusions must be finalized only after the simulator is implemented and the experiments are run.

Future work may include randomized jitter to reduce synchronized retries, configurable retry budgets, circuit-breaker behaviour, multiple simulated services, richer workload distributions, and validation against a controlled test network. Additional work could also compare the simulator's assumptions with traces collected from a non-production test environment.

ACKNOWLEDGMENT

The author would like to thank the project guide and the Department of Master of Computer Applications, Sri Jaykumar Rawal Institute of Technology, for their guidance and academic support. The project guide's name has been left blank for completion.

REFERENCES

[1]. V. Paxson, M. Allman, J. Chu, and M. Sargent, "Computing TCP's Retransmission Timer," RFC 6298, Internet Engineering Task Force, June 2011. https://doi.org/10.17487/RFC6298.

[2]. W. Eddy, Ed., "Transmission Control Protocol (TCP)," RFC 9293, Internet Engineering Task Force, August 2022. https://doi.org/10.17487/RFC9293.

[3]. J. Postel, "Transmission Control Protocol," RFC 793, Internet Engineering Task Force, September 1981. (Historical background; superseded by later TCP specifications.)

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