

Technische Hochschule Bingen

Fachbereich 2 Technik, Informatik und Wirtschaft

Reviving LibreSocial: Error Fixes and a Baseline Evaluation on Modern Systems

Master's Thesis

Faisal Imam
Student ID: 2195258

Supervisor: Prof. Dr.-Ing. Kalman Graffi

September 24, 2025

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit mit dem Titel:

„Reviving LibreSocial: Error Fixes and a Baseline Evaluation on Modern Systems“

selbständig und ohne unzulässige Hilfe Dritter verfasst habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Es wurden keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.

Ort, Datum:

Unterschrift: _____

Abstract

LibreSocial is a peer-to-peer social network created with vintage Java tools. It is difficult on modern computers to compile, launch, and measure. I revive the system on a new host through a binary first approach. I fixed approximately 120 compile and run issues, repaired faulty OSGi metadata, normalized the Java configs, and placed tiny guards so peers launch and monitoring work. Then I created a concise runbook containing the precise configurations and steps necessary to run the setup.

While the system was operational, I generated a foundational dataset derived from actual executions. Multiple peers participated, established the overlay, and distributed global metrics. I monitored parameters such as node stability, message rates and aggregates, network byte usage, routing hop count, latency for storage and retrieval, along with the impact of the replication factor. The objective is not to document the scale. Rather, the goal is to create a clear and replicable methodology that any future student can utilize to operate and enhance LibreSocial.

The result is three contributions. First, a running runtime with troubleshooting notes. Second, 20 clear plots and logs out of 900 plots that show the system's behavior under realistic settings. Third, an artifact pack and instructions that make the experiments reproducible and easy to grow into larger tests or new features.

Acknowledgements

I am deeply grateful to my supervisor, Prof. Dr.-Ing. Kalman Graffi, for his guidance, constructive feedback, and for providing access to the Network Security Lab at Technische Hochschule Bingen, which was essential for completing this thesis.

I also thank the former students who contributed to LibreSocial in earlier theses; their work provided the foundation on which this thesis could be built and extended.

For editorial support only, I used OpenAI ChatGPT and a custom GPT that I created to assist with LaTeX formatting and to cross-check a few legacy-Java refactoring patterns; all source changes, configurations, experiments, and analyses were implemented and verified by me [1], [2], [3].

On a personal note, I am profoundly grateful to my parents, Md Ismail and Nusrat Imam, for their constant support, patience, and encouragement. I also owe heartfelt thanks to my uncle, Afzal Mehdi, whose motivation and belief in me inspired me to pursue further studies and undertake this Master's program.

Table of Contents

1 Introduction **7**
- 1.1 Motivation . . . 7
- 1.2 Problem statement . . . 7
- 1.3 Goals and requirements . . . 8
- 1.4 Approach overview . . . 8
- 1.5 Scope and expected evidence . . . 9
- 1.6 Contributions . . . 9
- 1.7 Why running a legacy project is hard in 2025 . . . 9
- 1.8 Thesis structure . . . 10

2 Background and Related Work **11**
- 2.1 Overview . . . 11
- 2.2 Online social networks, P2P systems, and structured overlays . . . 11
- 2.3 LibreSocial architecture and modular runtime (OSGi) . . . 12
- 2.4 Observability and benchmarking practice . . . 13
- 2.5 Prior LibreSocial studies (what they established) . . . 13
- 2.6 How this thesis fits in . . . 14
- 2.7 Summary . . . 14

3 Problem Definition and Requirements **15**
- 3.1 Overview . . . 15
- 3.2 Problem and objective . . . 15
- 3.3 Success criteria . . . 16
- 3.4 Metrics, evidence, and scenarios . . . 17
- 3.5 Parameters, defaults, and data collection . . . 18
- 3.6 Risks and mitigation . . . 20
- 3.7 Summary . . . 20

4 Solution and Testbed **21**
- 4.1 Overview . . . 21
- 4.2 Environment and folder layout . . . 21

4.3 Start order and health gates . . . 23
4.4 Monitoring and plotting pipeline . . . 23
4.5 Multi-machine lab deployment (TH Bingen) . . . 25
4.6 Scenarios and parameter presets . . . 27
4.7 Limits and risks . . . 27
4.8 Summary . . . 27

5 Implementation and Run Steps 28
5.1 Overview . . . 28
5.2 Codebase recovery and toolchain alignment . . . 28
5.3 Fix log summary . . . 29
5.4 Configuration and bring-up for a healthy overlay . . . 30
5.5 Evidence produced . . . 32

6 Results 33
6.1 Overview . . . 33
6.2 Why these 20 metrics out of ∼900? . . . 33
6.3 Health and stability . . . 34
6.4 Messaging throughput . . . 36
6.5 Network I/O . . . 38
6.6 Routing quality . . . 40
6.7 Storage path . . . 41
6.8 Social/plugin activity . . . 43
6.9 Run parameters . . . 43
6.10 Key medians and IQR . . . 44
6.11 Summary . . . 45

7 Discussion and Limitations 46
7.1 What the plots say at a glance . . . 46
7.2 Scope decisions and what we left for future tests . . . 46
7.3 Validity and relation to prior work . . . 47
7.3.1 Internal validity . . . 47
7.3.2 External validity . . . 47
7.3.3 Construct validity . . . 47
7.3.4 Conclusion validity . . . 47
7.4 Reproducibility lessons and testing coverage . . . 48
7.5 Risks and how we mitigated them . . . 48
7.6 Summary . . . 49

8 Re-plotting and Reuse Guide **50**
8.1 Overview . . . 50
8.2 Inputs, layout, and re-plot recipe . . . 50
8.3 Adding metrics, comparing runs, and extensions . . . 51
8.4 Summary . . . 53

9 Conclusion and Future Work **54**
9.1 What this thesis set out to do . . . 54
9.2 What was achieved . . . 54
9.3 What the results tell us . . . 55
9.4 Limitations to keep in mind . . . 55
9.5 Recommendations and future work . . . 56
9.6 Closing . . . 57

A Environment & Runbook (Binary-only Path) **58**
A.1 Purpose . . . 58
A.2 Pin these versions (do not improvise) . . . 58
A.3 Folder layout to create . . . 59
A.4 One-time installation . . . 59
A.5 Getting the code/binaries . . . 59
A.6 Importing and runtime alignment . . . 60
A.7 Start order (exact sequence) . . . 60
A.8 Key configs to snapshot per run . . . 60
A.9 Pre-run sanity checklist . . . 61
A.10 Evidence to keep . . . 61
A.11 Common gotchas . . . 61

B Plotting & Reuse **62**
B.1 Purpose . . . 62
B.2 Inputs . . . 63
B.3 Quick recipe (re-plot approx. 900 PDFs) . . . 64
B.4 Why these 20 out of ∼900 columns? . . . 64
B.5 Comparing runs (S1/S2/S3) . . . 64

C External resources and assistance **65**

Bibliography **67**

Chapter 1

Introduction

1.1 Motivation

Most social networks are run by a few companies. This is simple for users, but raises questions about privacy, control, and long-term cost. A different path is to let users help run the service with peer-to-peer systems. LibreSocial follows this idea. It brings a structured overlay, distributed storage, access control, and social features into one framework that can run without a central server [4], [5]. Earlier project groups and theses built the blocks and showed how to observe a running system with SkyEye.KOM and related tools [6], [7], [8], [9]. Classic P2P substrates such as Pastry, PAST, and SCRIBE explain the routing, storage, and pub/sub ideas that LibreSocial builds on [10], [11], [12], [13].

1.2 Problem statement

Given: LibreSocial was created with older Java tools. On current machines a straight rebuild is not smooth. Toolchains, crypto policy files, OSGi bundle metadata and SWT versions are strict. Students who only want to evaluate the system hit many errors before they can even start a test

Sought: A practical, repeatable binary-only run on a modern host. Start several peers, enable built-in testing and monitoring, and collect clean metrics for a small to medium setup. The outcome should be easy to repeat so the next student can extend it without changing the scope to containers

Why it matters? A simple path to run and measure gives a baseline. It keeps the learning curve low, connects classic P2P ideas with today's

machines, and makes new evaluations possible again.

1.3 Goals and requirements

To know when the work "counts," we define functional and non-functional goals.

Functional goals

F1. Start LibreSocial from binaries on a current host

F2. Connect multiple peers so they form a ring

F3. Turn on the Tester and Monitoring to generate activity and record metrics

F4. Export logs and metrics for plotting to match earlier benchmarking style [8], [14].

Non-functional goals

N1. Stable runs over hours without crashes.

N2. Reasonable message delivery ratio and no growing exception counters.

N3. Lookup paths with expected hop counts for the ring size.

N4. Practical storage and retrieval times.

N5. Results repeatable under the same parameters. Target ranges follow prior LibreSocial evaluations and common P2P expectations [8], [11], [14].

1.4 Approach overview

1. Restore by binaries. Use a compatible Java 7 runtime, set required crypto policies, and launch peers from released binaries. Confirm peers see each other.

2. Enable monitoring. Use the integrated Tester and monitoring to capture message counts, hop counts, retrieval times, and resource usage.

3. Run scenarios. Do a baseline activity run, a longer stability run, and a small replication-factor sensitivity run. These mirrors published methodology so results are comparable [8], [14].

4. Collect and analyse. Save logs and metrics, create plots, and read trends and previous LibreSocial reports.

1.5 Scope and expected evidence

The focus is a binary-only run and a compact evaluation on one host or a small lab. Internet-scale experiments and containerization are out of scope. The aim is a clean, reproducible baseline, not record size. We will report at least: node stability over time, sent and received messages, lookup hop count, storage and retrieval times, effects of replication settings, and basic resource usage (CPU, memory, I/O). These align with standard P2P benchmarking and earlier LibreSocial studies [7], [9].

1.6 Contributions

C1. A clear binary-only runbook and configs that make LibreSocial start reliably on modern hosts.

C2. A small but complete evaluation dataset with plots and logs, designed for repeatability.

C3. A mapping from requirements to tests so the next student can extend the work.

1.7 Why running a legacy project is hard in 2025

Running LibreSocial on a current machine is not "click and run." The project needs JDK 1.7.0_80, manual Unlimited JCE policy files, and an old Eclipse Indigo IDE. OSGi bundle imports and SWT/JFace versions must match. Missing jars lead to class-not-found errors. These are common reasons a fresh setup fails today and they justify our binary-first path and the focus on a reproducible baseline [15].

1.8 Thesis structure

Chapter 2 reviews OSN background, structured overlays and modular runtimes. Chapter 3 states the problem and requirements in detail. Chapter 4 sketches the testbed and scenarios. Chapter 5 explains codebase rehabilitation and the binary run path. Chapter 6 presents experiments and results. Chapter 7 discusses limits and lessons. Chapter 8 shows how to regenerate plots from raw metrics. Chapter 9 concludes with future work.

Chapter 2

Background and Related Work

2.1 Overview

This chapter gives the background needed for the rest of the thesis. First, it frames decentralization of online social networks and presents the peer-to-peer model. Then it covers structured overlays (Pastry for routing, PAST for storage, SCRIBE for pub/sub) and LibreSocial's mapping of social features onto the primitives. It then it covers LibreSocial's multi-layered architecture and OSGi-based modular runtime. Finally, it summarizes observability and benchmarking practice used in previous work on LibreSocial and situates this thesis as a continuation of that line. After this chapter, the reader has the terms and criteria used to define requirements and scenarios in Chapter 3.

2.2 Online social networks, P2P systems, and structured overlays

Popular social networks are easy to use because one company runs the service. The trade-off is less control for users over data, policy, and cost. Research on social network sites explains how people use these systems and why privacy and governance matter [16]. A technical alternative is a peer-to-peer (P2P) design, where many user devices cooperate and no single server owns the data. LibreSocial follows this idea and aims to offer profiles, posts, groups, and messaging over a distributed substrate [4], [5].

In P2P systems, nodes talk directly. Structured overlays give each node an identifier and route lookups in $O(\log N)$ hops on average. Classic examples are Pastry for routing, PAST for distributed storage, and SCRIBE for publish/subscribe [11], [12], [13]. These building blocks are a good fit for so-

cial workloads that need many small reads, moderate writes, and fan-out to followers or groups. Prior LibreSocial work adopts these ideas and discusses their trade-offs for an online social network [4], [5].

2.3 LibreSocial architecture and modular runtime (OSGi)

LibreSocial is organized in layers: an overlay/routing layer, a core framework for storage and communication, and social plugins with a GUI on top. The project targets modularity so that components can be extended or replaced over time [4], [5], [8].

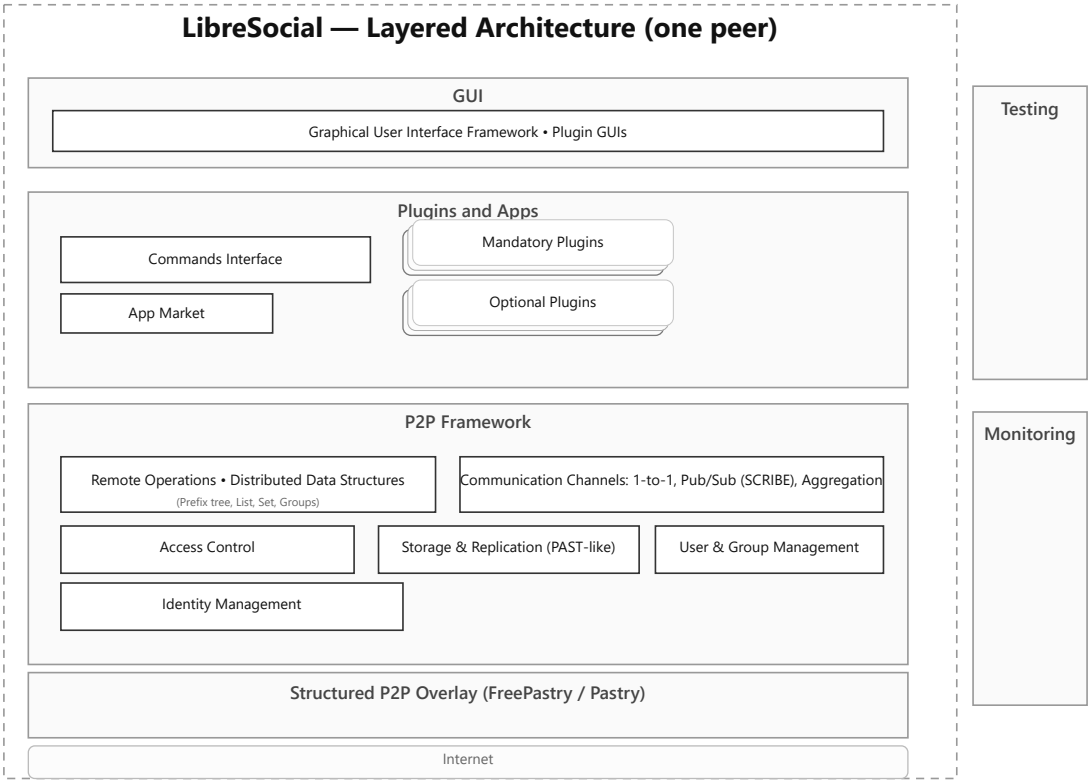


Figure 2.1: LibreSocial Architecture

Components are packaged as OSGi bundles. Each bundle declares in its manifest which packages it provides and which it needs. This clean split helps organize overlay, storage, pub/sub, plugins, and the GUI. The same mechanism also makes setup strict: if imports, exports, or versions do not

match, the runtime refuses to start those bundles. This is one practical reason why running the legacy code on new machines is hard and why a binary-first path helps [5], [9].

2.4 Observability and benchmarking practice

To evaluate a distributed P2P system, we need global metrics, not only local logs. LibreSocial evaluations follow a simple tuple: quality attributes $\rightarrow$ metrics $\rightarrow$ scenarios, and they distinguish system, workload, and environment parameters. This structure supports repeatability and apples-to-apples comparisons across runs and theses [8], [17]. In this line of work the main metric families are:

- **Stability & health:** node count over time, exceptions, memory usage.
- **Messaging throughput:** messages sent/received (rate and total).
- **Network I/O:** bytes sent/received (rate and total).
- **Routing quality:** mean hop count vs. network size.
- **Storage path:** store/retrieve rates and latency (lookup, put, get).
- **User-visible footprint:** counters from social plugins (e.g., wall posts).

These families are used by prior LibreSocial studies and we keep them for comparability [8], [9], [14].

2.5 Prior LibreSocial studies (what they established)

- **Framework and vision.** The preprint and dissertation describe the target design and the security/scalability choices for a P2P online social network [4], [5].
- **Project groups (2011–2013).** Early reports documented core modules and the monitoring setup that later theses reused [6].
- **Benchmarking theses.** Khitman (2020) benchmarked LibreSocial using integrated testing and SkyEye.KOM, reporting scenarios for stability, scalability, and replication effects [8].

- **Systematic evaluation (paper).** The 2020 evaluation paper summarized methodology and metrics that we mirror so results are easy to compare [14].
- **Systematic evaluation (Master's, UPB).** Balabhadruni & Peng (2014) provide complementary end-to-end test patterns and pitfalls when exercising social features over a P2P substrate [9].

2.6 How this thesis fits in

This thesis does not propose a new overlay. It adds three practical pieces to the same research line:

1. **Legacy rehabilitation.** I restored a runnable LibreSocial on a current host by resolving about 120 build/runtime issues, aligning Java 7 settings, and correcting OSGi metadata so peers start and metrics export reliably.
2. **Baseline dataset and runbook.** I produced a baseline measurement set-20 global plots covering stability, throughput, routing, storage latency, and plugin activity
3. **Pilot evaluation.** I report pilot-scale results and a binary-only run path that future students can reuse and extend to larger tests or new features, without rebuilding the legacy toolchain [4], [6], [8].

2.7 Summary

Centralized OSNs trade simplicity for user control; a structured P2P approach offers predictable routing (approximately $\log N$ hops), replicated storage, and group messaging that suit social workloads. LibreSocial applies these ideas in a layered, OSGi-modular design. Prior work established the metric families we will reuse: health, messaging, network I/O, routing, storage, and user-level activity. This thesis contributes a binary-only run path and a small, comparable baseline built on those foundations. With this background in place, Chapter 3 formalizes the problem, success criteria, metrics, and scenarios that we will implement and evaluate.

Chapter 3

Problem Definition and Requirements

3.1 Overview

This chapter formalizes the problem we address and the outcome we seek, sets the concrete constraints of the work, and states the system model and assumptions that shape the later scenarios and metrics. We first explain why a binary-only path is needed for the legacy Java/OSGi stack, then list the constraints under which we operate. We close with the model assumptions for routing, storage, group delivery, monitoring, and control. These definitions ground the implementation in Chapter 4 and the evaluation in Chapters 5–6.

3.2 Problem and objective

Given. LibreSocial is a peer-to-peer social network built with an older Java 7 and OSGi stack. On current laptops a plain rebuild is not smooth. Peers do not start without fixing bundle metadata and selecting compatible library versions. Prior studies already defined what to measure and which scenarios are useful, so we can keep our results comparable without changing the architecture [4], [8], [14].

Constraints for this thesis.

- Use binary-only execution on a modern host. No containerization. This keeps setup short for students and avoids toolchain drift.
- Use the integrated Tester and Monitoring that come with LibreSocial. No external workload generators.

- Focus on pilot-scale runs that are long enough to be stable, not on record size. The goal is a clean baseline that others can extend.

Objective (what is sought). A practical, repeatable method to start several peers, generate activity, and collect clean system-wide metrics. The output is a small dataset and a runbook that any student can follow. The baseline must mirror the families of metrics and the reading style used in recent LibreSocial work so that we can compare trends across years.

3.3 Success criteria

We define when the work counts as a solution. Goals are split into functional and non functional.

Functional goals (FG)

FG1. Start LibreSocial from binaries on a current host.

FG2. Form an overlay with several peers that exchange messages.

FG3. Enable the Tester and Monitoring so the system creates load and exports GLOBAL_* metrics.

FG4. Persist logs and metrics to files that can be plotted and checked later.

Non functional goals (NFG)

NFG1. Stability. Peers run for hours without crashes. Node count stays flat after warm up. Exceptions stay near zero.

NFG2. Delivery. Sent and received message rates are close during steady state. Large gaps require a reason and a fix.

NFG3. Routing quality. Mean hop count is in the expected range for the observed N (near $\log N$ for Pastry-like routing).

NFG4. Storage latency. Store time and retrieve time are practical for the small data items used by social features.

NFG5. Repeatability. Re-running a scenario with the same parameters gives similar curves and medians.

3.4 Metrics, evidence, and scenarios

We keep the metric families used in prior work so plots are easy to read and compare [7], [8], [9], [14]. These metrics tie directly to our functional and non-functional goals, and the scenarios below exercise them in a repeatable way.

Metric families.

- **Stability and health:** node count over time, memory usage mean, null pointer and other exception counters. Evidence of NFG1.
- **Messaging throughput:** messages sent and received per second and in total. Evidence of FG2 and NFG2.
- **Network I/O:** bytes sent and received per second and in total. Evidence of load and replication cost.
- **Routing quality:** mean hop count. Evidence of NFG3.
- **Storage path:** store rate, retrieve rate, store time, retrieve time, lookup time. Evidence of NFG4.
- **User footprint:** simple plugin counters such as wall posts. Evidence that the social layer is alive.

Reading rule of thumb. Look at node count and exceptions first. If they are steady, the run is valid. Then read messages and network to spot overload or gaps. Use hop count and latency to interpret cost and quality [8], [9], [14].

Scenarios to run. We follow a small set of scenarios that match earlier studies so numbers are comparable [8], [9], [14].

- **S1 Baseline.** Aim: show a healthy system under moderate activity. Evidence: steady node count, exceptions flat, sent close to received, stable rates for store and retrieve, hop mean in range.
- **S2 Stability.** Aim: run longer to watch for drift in memory or throughput. Evidence: curves remain within a narrow band after warm up.
- **S3 Replication sensitivity (k sweep).** Aim: show how changing k affects cost and latency. Evidence: higher k increases bytes and store time as expected, while lookups and fetches remain practical.

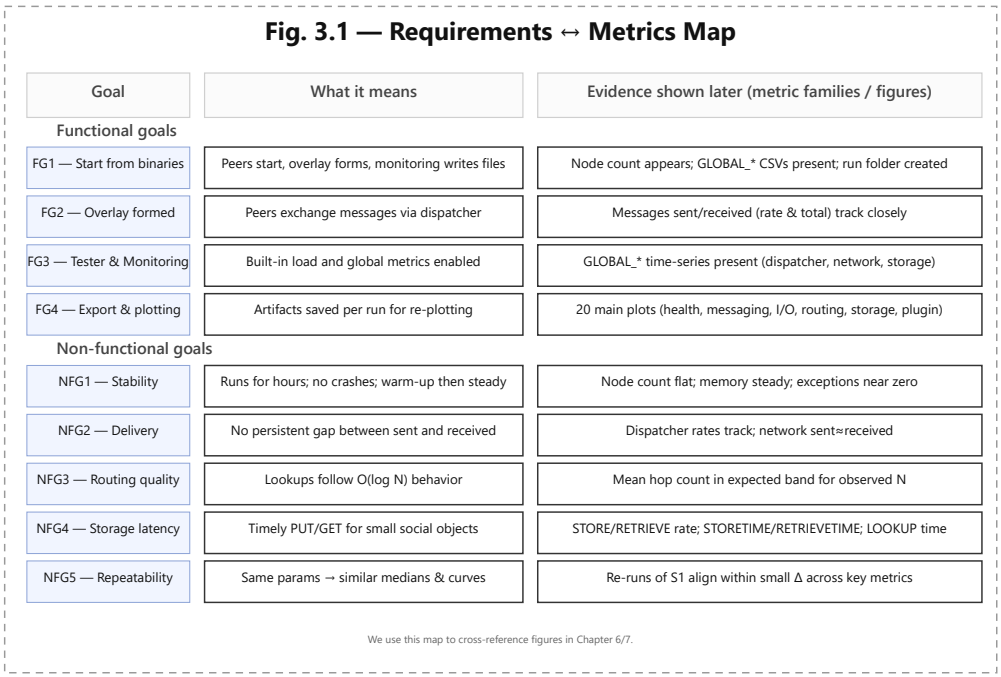


Figure 3.1: Requirements ↔ Metrics Map

This map links the system's functional and non-functional requirements to specific metric families. Each requirement is covered by at least one measurable indicator, ensuring traceability between goals, the experiments in Chapter 6, and the discussion in Chapter 7.

3.5 Parameters, defaults, and data collection

We separate system, workload, and environment parameters. This is the style used in LibreSocial evaluations and P2P benchmarking texts [14], [18].

Parameter taxonomy.

- **System:** number of peers N, replication factor k, overlay base, cache sizes.
- **Workload:** post rate, message size, object size for store and get, group sizes.
- **Environment:** host CPU and RAM, JVM options, OS, Java version.

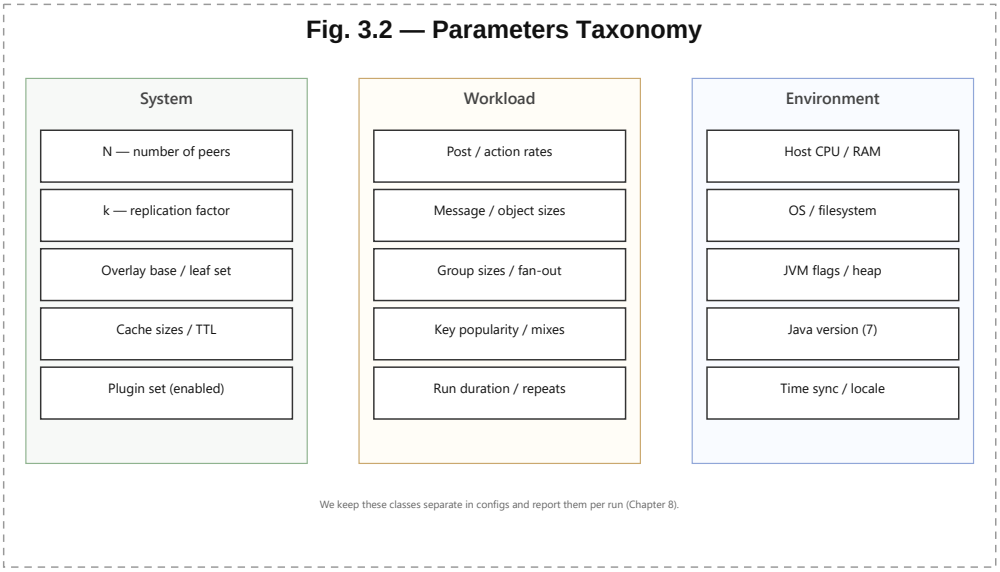


Figure 3.2: Parameters Taxonomy

This taxonomy groups parameters into three categories: **System** (overlay and storage configuration), **Workload** (message and activity patterns), and **Environment** (network and host setup). Each run reports these parameters explicitly, and they remain constant across repeated experiments to ensure comparability.

Defaults used for the baseline.

- Java 7 runtime, single host with several peers, one controller, fixed seeds.
- Warm up period of at least five minutes before measuring.
- Repeat each run at least twice to check repeatability.

Operational rules.

- Use the controller only to trigger scenarios and collect data. Keep exactly one controller to avoid interference [8], [18].
- Keep the same random seeds and configuration files across repeats for fair comparison [8], [14].
- Do not change plugins during a run. Changes restart the bundles and break repeatability [9], [18].

Data collection and plotting.

- Collect GLOBAL_* time series and logs from all peers.
- Store files under a run identifier with a timestamp so results can be traced.
- Plot the twenty main charts that cover health, messaging, network, routing, storage, and plugin activity.
- Keep raw CSV files together with plots for reuse in later work.

3.6 Risks and mitigation

- **Toolchain drift.** A library update or JRE change may break the run. Mitigation: pin all versions and document them in the runbook.
- **Measurement bias after fixes.** Code changes could alter behavior. Mitigation: repeat S1 with the same seeds before and after major fixes; check that medians and IQRs stay within a narrow band.
- **Overload artifacts.** Multiple controllers or too high rates can create spikes that look like bugs. Mitigation: one controller only and moderate rates in S1; scale slowly.
- **Short warm up.** Routing tables and caches need time. Mitigation: enforce a warm up period and ignore the first minutes in analysis.

3.7 Summary

We defined a binary-only path to run LibreSocial on a modern host and a small set of repeatable scenarios that produce clear plots. Success means stable runs, hop counts in the expected range, practical store and retrieve times, and results that repeat under the same parameters. We also fixed the constraints and metric families that will evidence these goals. The next chapter presents the solution and the testbed that implement this plan.

Chapter 4

Solution and Testbed

4.1 Overview

This chapter describes the practical setup used to run LibreSocial on modern machines. It first fixes the host and software environment (Java 7, OSGi, GUI bundles) and the directory layout used to capture evidence per run. It then defines the start order and health gates that ensure valid measurements, explains the monitoring and plotting pipeline that produces the figures in Chapter 6, and documents a multi-machine lab deployment at TH Bingen. With this setup in place, later chapters can reproduce runs and compare results consistently. Detailed runbook steps are in Appendix A. Plot generation is in Appendix B.

4.2 Environment and folder layout

Hardware:

- CPU: recent x64 (4–8 cores)
- RAM: 16 GB or more
- Disk: SSD

Software:

- OS: Windows or Linux, time sync on (NTP)
- Java: JDK 1.7.0_80 (set `JAVA_HOME` and `PATH`)
- OSGi: runtime bundled with the release

- GUI libs: SWT/JFace versions that match the bundles

This avoids class version and crypto policy issues that show up with newer JDKs or mismatched bundles. The exact versions used are listed in Appendix A.

Project tree and per-run artifacts. Keep one project root and one subfolder per run:

```
/libresocial/
  /bin/                    # launchers and scripts
  /bundles/                # OSGi bundles (overlay, storage, plugins, GUI)
  /config/                 # *.properties for peers, tester, monitoring
  /logs/                   # local logs per peer
  /runs/
     /RUN_YYYYMMDD_HHMM_<tag>/
        metrics/           # GLOBAL_* CSV time series
        plots/             # generated charts
        configs/           # copy of the exact configs used
        README.txt         # short note with N, k, rates, seeds, OS/JDK
```

The `/runs/` tree is the main evidence for Chapters 6 and 7 [7], [8], [9], [14].

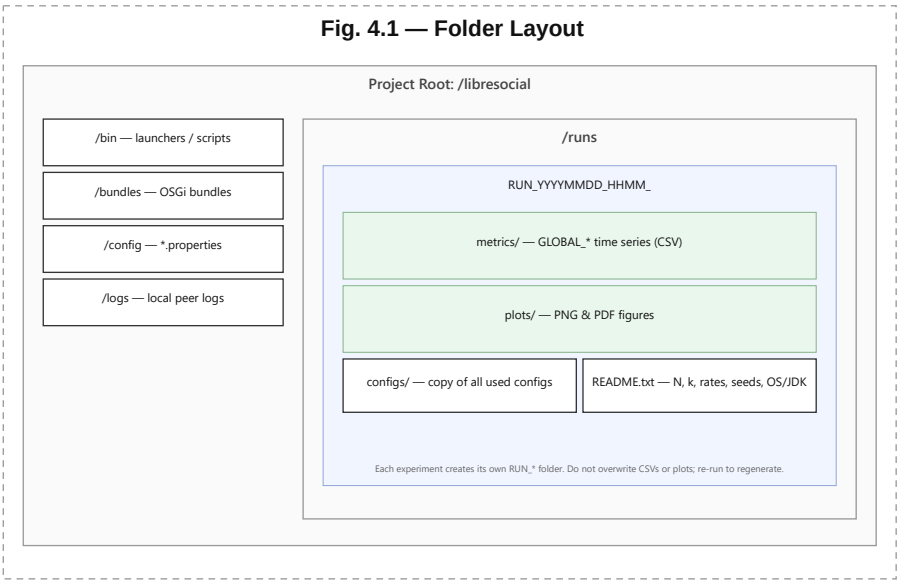


Figure 4.1: Folder layout and per-run artifacts.

4.3 Start order and health gates

Why order matters. A consistent start/stop sequence prevents missing metrics, double controllers, or half-formed overlays.

Start sequence.

1. Set `JAVA_HOME` to JDK 1.7.0_80 and verify `java -version`.
2. Start the **monitoring collector** on the designated machine; confirm that `runs/<id>/metrics/` begins to fill with `GLOBAL_*` CSVs.
3. Start **seed peers** (1–2) and then the remaining peers; all must point to the same collector and share the same bootstrap list.
4. Start **exactly one controller** (tester) to generate activity.
5. Let the system **warm up for $\geq$5 min** before recording.

Table 4.1: Health gates for a valid run

Gate	Pass condition (else fix and restart)
Nodes	`GLOBAL_NETWORK_NODES_COUNT_NODES` rises to N and stays flat after warm-up.
Exceptions	Null-pointer and other exceptions remain near zero.
Messaging	Sent $\approx$ received in steady state (no long dry intervals).
Storage	Store/retrieve series active; lookup/store/get latencies present.
Files	New CSV rows appear at the expected cadence in `runs/<id>/metrics/`.

4.4 Monitoring and plotting pipeline

Data flow (peers $\rightarrow$ collector $\rightarrow$ plots). Peers emit local metrics; the collector aggregates them into `GLOBAL_*` time series for messaging, network I/O, routing, storage, memory, exceptions, and plugin counters [7]. Timestamps are wall-clock, so keep NTP on. Each run creates a fresh `/runs/RUN_YYYYMMDD_HHMM_<tag>/metrics/` folder.

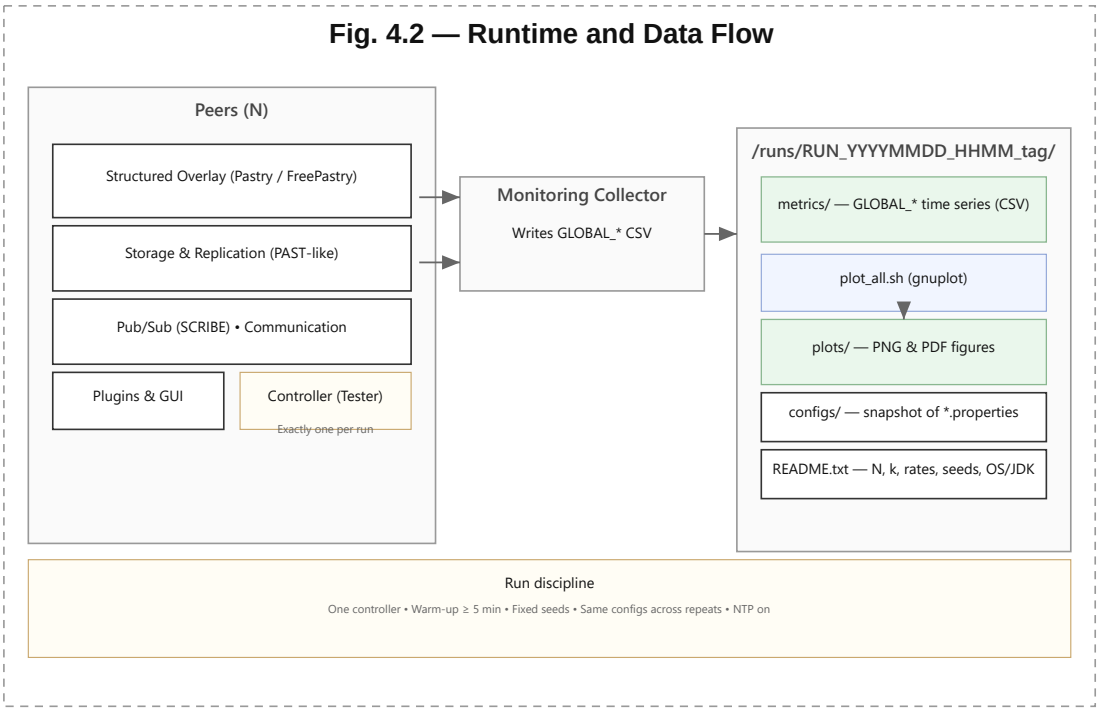


Figure 4.2: Runtime and Data Flow

This diagram shows the runtime and data flow used in this thesis. Multiple peers run in parallel, exchanging overlay messages while the monitoring collector aggregates system-wide metrics (GLOBAL_* series). These metrics are then exported to CSV and plotted for analysis. This structure ensures reproducibility and consistent data capture across all runs.

Data management and plotting steps.

- Use a distinct run ID per experiment: `RUN_YYYYMMDD_HHMM_<tag>`.
- Copy all configs into `/runs/.../configs/` for reproducibility.
- Keep raw CSVs (do not overwrite). Plot from CSVs only.
- Render figures from the CSVs with the gnuplot script (Appendix B, Listing B.1: `plot_all.sh`); outputs go to `/runs/.../plots/`.

Command:

```
cd /libresocial/runs/RUN_YYYYMMDD_HHMM_<tag>
bash plot_all.sh
```

The script writes PNG and PDF plots into `/runs/.../plots/` with file names that match metric names. Chapter 6 references Appendix B when plots are shown.

Table 4.2: Run artifacts map

Path	**What it contains**	**Used in...**
`runs/<id>/metrics/`	GLOBAL_* CSV time series	Ch. 6 plots, Ch. 7 discussion
`runs/<id>/plots/`	PNG/PDF figures from the CSVs	Ch. 6
`runs/<id>/configs/`	Copy of all `.properties` used	Appendix A (repro)
`runs/<id>/README.txt`	N, k, rates, seeds, OS/JDK	Appendix A
`logs/`	Local peer logs	Troubleshooting (Appendix A)

4.5 Multi-machine lab deployment (TH Bingen)

Purpose. To show that the binary-only path works across independent machines, not just multiple peers on one host.

Setup. Network Security Lab, Technische Hochschule Bingen; wired LAN (1 GbE); identical JDK 1.7.0_80 on all machines; one bootstrap + collector, 1–2 seeds, remaining user peers; one controller.



Figure 4.3: Multi-machine LibreSocial deployment in the Network Security Lab

Each workstation runs a distinct LibreSocial account connected to the same overlay. One designated machine serves as the bootstrap node and monitoring collector, ensuring that all peers connect to a common overlay and metrics are gathered centrally.

4.6 Scenarios and parameter presets

We use the three scenarios from Chapter 3 to keep results comparable [8], [9], [14].

- **S1 Baseline:** moderate activity on one host.
- **S2 Stability:** longer duration to check for drift.
- **S3 Replication sensitivity:** small sweep of k to see cost and latency.

Presets

- N peers on one host, $k \in \{1, 3, 5\}$ for S3.
- Moderate post and message rates.
- Fixed seeds and identical configs across repeats.
- Warm up at least 5 minutes before measuring.

4.7 Limits and risks

- Single host bias: no WAN delay or churn.
- Resource caps: CPU or NIC may cap throughput, keep S1 moderate.
- Toolchain drift: changing JRE or libs breaks runs, pin versions and document them.
- Operational mistakes: two controllers or short warm up distort rates, enforce one controller and at least 5 minutes warm up.

4.8 Summary

We fixed the runtime environment (Java 7 and matching bundles), standardized the folder layout for reproducible evidence, enforced a safe start order with health gates, and documented a classroom-scale deployment with a central collector. With this setup, Chapter 5 details the implementation steps and fixes that make the peers start reliably, and Chapter 6 presents the resulting plots and statistics. Full runbook steps are in Appendix A. The plotting script is in Appendix B.

Chapter 5

Implementation and Run Steps

5.1 Overview

Make LibreSocial run reliably on a current machine, using released binaries, while fixing the legacy issues that blocked startup and clean measurement. Keep the setup comparable to earlier LibreSocial work so later plots line up with past results [5], [8], [14].

5.2 Codebase recovery and toolchain alignment

I first opened the code to understand why a plain start failed. The main problems were language level mismatches, missing or wrong library versions, and OSGi bundle wiring. I pinned the runtime to JDK 1.7.0_80, matched the OSGi and SWT/JFace bundles that ship with the release, and verified FreePastry/PAST/SCRIBE jars were consistent with the versions expected by the binaries [10], [13].

Result: the number of compile/start errors dropped a lot once the Java version and bundle set were aligned.

Table 5.1: Dependency alignment

Component	**Version used**	**Why this version**
JDK	1.7.0_80	matches code features and crypto policy
OSGi runtime	...	matches bundled imports/exports
SWT/JFace	...	GUI bundle compatibility
FreePastry	...	overlay and PAST compatibility
Tester/Monitoring	release build	expected GLOBAL_* output

5.3 Fix log summary

About 120 issues showed up initially. Below are the some error categories with one a example each.

A) Language level and compiler settings

Errors like `Cannot instantiate the type ArrayList<?>` and `switch(param)` not accepted came from compiling with a lower language level.

Fix: set compiler compliance to Java 7. Where that was not possible, replace new syntax with Java-6 style.

Listing 5.1: Example fix for generics and diamond

```
Before:
List<String> f = new ArrayList<>();

After:
List<String> f = new ArrayList<String>();
```

Listing 5.2: Example fix for try-with-resources

```
Before:
try (ResultSet rs = stmt.getGeneratedKeys()) { ... }

After:
ResultSet rs = null;
try {
    rs = stmt.getGeneratedKeys();
    ...
} finally {
    if (rs != null) rs.close();
}
```

B) Missing helpers or wrong imports

Errors like `ConfigLoader cannot be resolved` or unresolved OSGi classes. Fix: add the missing bundle/jar to the runtime, or implement a thin local wrapper when a helper was absent. Verify bundle manifest imports/exports.

C) OSGi resource loading

Classpath loads failed for files packaged inside bundles. Fix: load resources from the bundle context.

Listing 5.3: Example fix for OSGi resource loading

```
InputStream in = context.getBundle()
                        .getEntry("/MyConfig.txt")
                        .openStream();
```

D) Collections and raw types

Errors around `new HashMap<>()` on older compilers or wildcards like `ArrayList<?>`. Fix: use explicit type parameters or a targeted wildcard only where needed.

Listing 5.4: HashMap typing

```
Map<String, String> m = new HashMap<String, String>();
```

E) JDBC inserts with generated keys

Some drivers required explicit close and error checks. Fix: try–finally around `PreparedStatement` and `ResultSet`, and a clear "no ID obtained" branch.

Table 5.2: Error taxonomy and fixes applied

Category	**Typical message**	**Fix pattern**
Language level	diamond / switch on String	set 1.7 or rewrite
OSGi wiring	class not found in bundle	add bundle / fix manifest
Resource loading	getResourceAsStream null	use getEntry
Collections typing	wildcard instantiation	add concrete type
JDBC lifecycle	generated keys / close	try–finally

5.4 Configuration and bring-up for a healthy overlay

Configs. I prepared clean `.properties` files for peers and monitoring. The key parts are the bootstrap nodes, replication factor k, message rates, and log/metric paths. A copy of the exact files used in each run is stored in `runs/<id>/configs/` and printed in Appendix A. This keeps runs reproducible and makes it easy for the next student to extend.

Table 5.3: Key parameters changed

File	Key	Value	Reason
overlay.properties	bootstrap	...	connect peers fast
storage.properties	k	1/3/5	replication sweep
tester.properties	rate	...	keep S1 steady
monitor.properties	path	runs/<id>/metrics	collect CSVs

Bring-up and validation. With code and configs stable, I used the start order from Chapter 4. I checked: node count rises then flattens, exceptions stay near zero, sent is close to received, and storage metrics are active. When a check failed, I iterated on the last change before moving on.

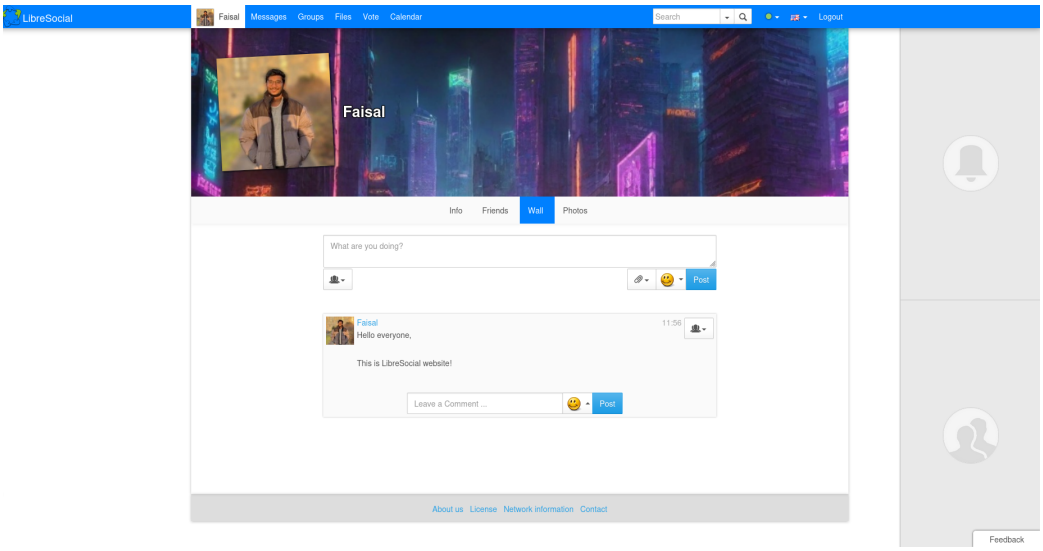


Figure 5.1: LibreSocial GUI during a baseline run

A user is logged in and a wall post is visible. This confirms the end-to-end stack is operational while metrics are recorded.

Table 5.4: Pre-run sanity checklist

Check	**Pass condition**	**If it fails**
Nodes	count flat after warm-up	fix bootstrap or ports
Exceptions	near zero	inspect latest stack trace
Sent ≈ Received	small gap in steady state	confirm exactly one controller
Storage	rates and times present	check k and object size
CSVs	files created per minute	verify collector path

5.5 Evidence produced

Each run creates a dated folder under `/runs/` with GLOBAL_* CSVs and plots. Plots are generated with a small gnuplot script so names match metric names. The short command and full script are documented in Appendix B. This keeps Chapter 6 lean and repeatable.

Chapter 6

Results

6.1 Overview

This chapter shows the 20 plots we use to judge correctness and behavior of a LibreSocial run. Graphs are grouped by purpose: health, messaging, network I/O, routing, storage, and social activity. Each figure has one short takeaway so the reader can move fast.

All figures in this chapter were rendered from the collector's CSVs using the script in Appendix B (Listing B.1) and come from the numbered PDFs you provided.

6.2 Why these 20 metrics out of ∼900?

We chose a compact set that still covers the end-to-end path:

- **Coverage across layers:** overlay/routing, storage, social/plugin, plus health and I/O.
- **Non-redundant signals:** one count, one mean rate, and one safety metric per family where possible.
- **Stable and comparable:** align with older LibreSocial and P2P benchmarks so trends are comparable.
- **Interpretability:** each plot speaks for itself (for example, sent ≈ received).
- **Global scope:** GLOBAL_* series aggregate across nodes and are ideal for single-host runs.

6.3 Health and stability

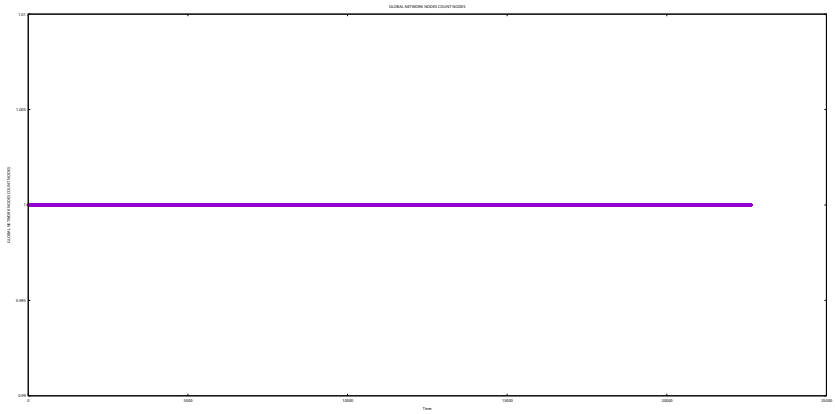


Figure 6.1: GLOBAL_NETWORK_NODES_COUNT_NODES

The plot shows how node count rises during the join phase and then flattens in steady state. A flat line later in the run indicates that the overlay is stable.

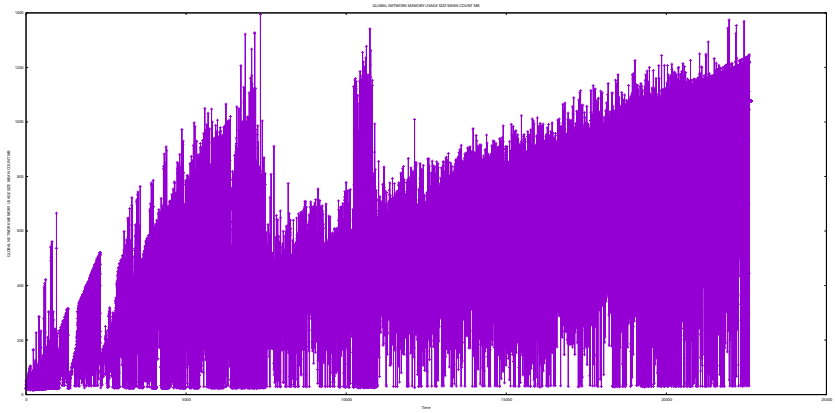


Figure 6.2: GLOBAL_NETWORK_MEMORY_USAGE_SIZE_MEAN_COUNT_MB

The figure shows average memory usage across nodes. After warm-up, memory consumption settles into a stable band. A gradual upward drift would suggest memory leaks or overload conditions.

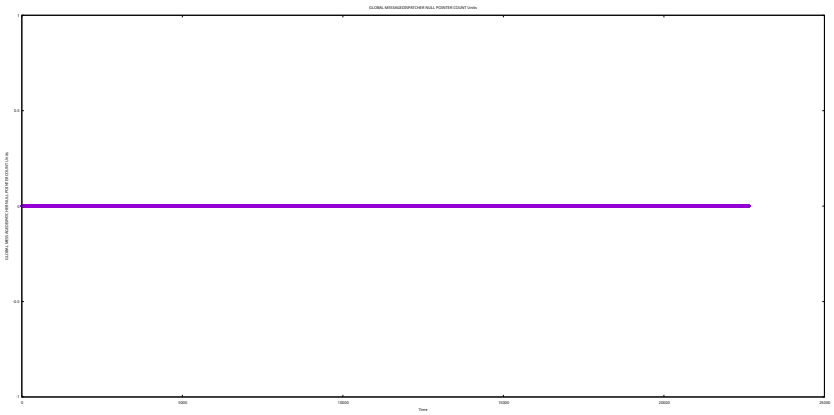


Figure 6.3: GLOBAL_MESSAGEDISPATCHER_NULL_POINTER_COUNT_units

The plot shows the number of null-pointer exceptions across the run. Ideally, this value should stay near zero. Any spikes here indicate functional faults in the system.

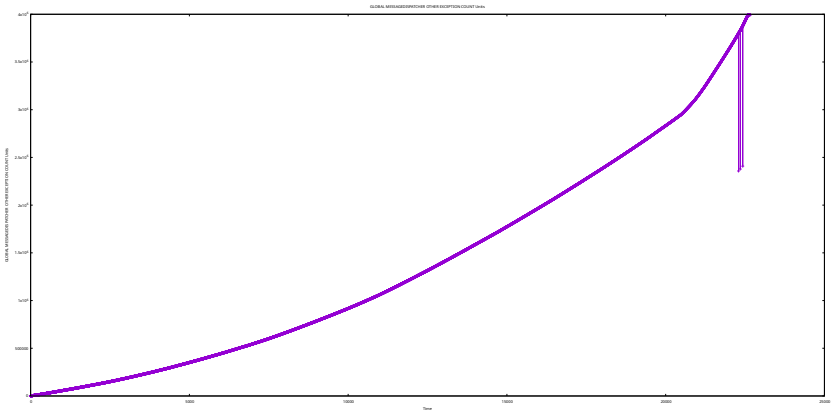


Figure 6.4: GLOBAL_MESSAGEDISPATCHER_OTHER_EXCEPTION_COUNT_units

This figure shows all other exceptions recorded by the message dispatcher. Keeping these values low builds confidence in the correctness of the system and the validity of later throughput and latency results.

6.4 Messaging throughput

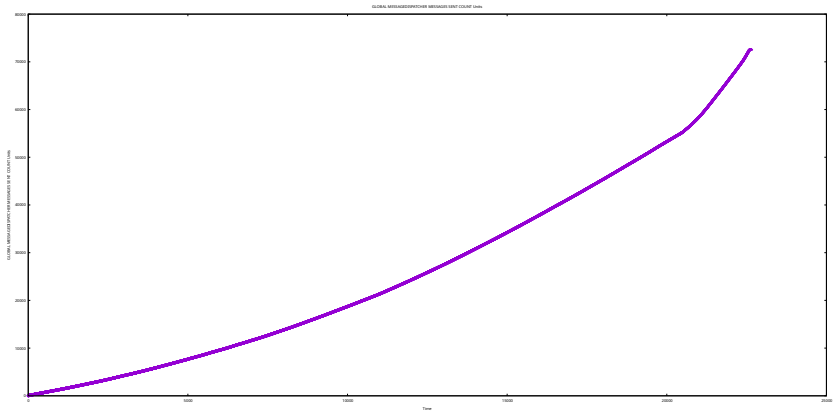


Figure 6.5: GLOBAL_MESSAGEDISPATCHER_MESSAGES_SENT_COUNT_units

The figure shows the cumulative number of sent messages. The curve should rise smoothly during periods of activity, which indicates steady and reliable message dispatching.

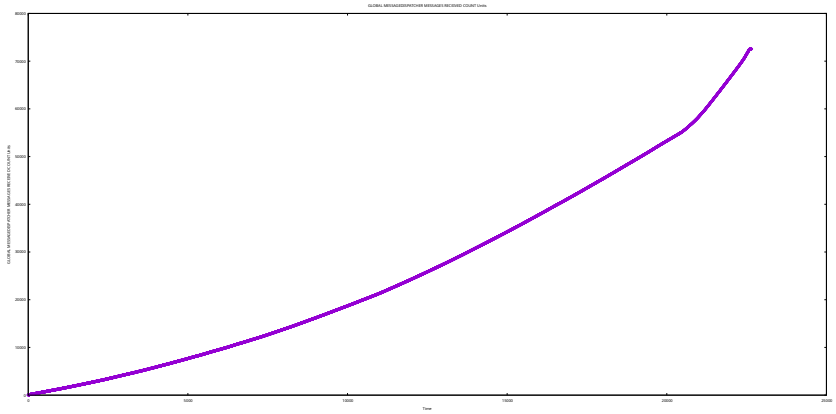


Figure 6.6: GLOBAL_MESSAGEDISPATCHER_MESSAGES_RECEIVED_COUNT_units

This figure shows the cumulative number of received messages. In a healthy run, the number of received messages should closely match the number of sent messages, confirming reliable message delivery.

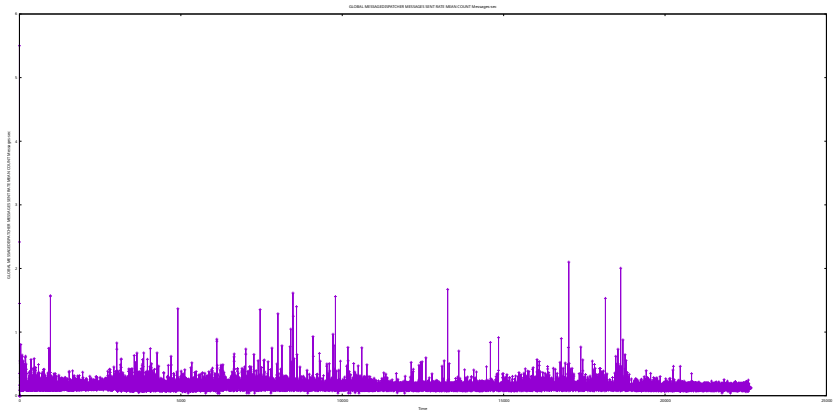


Figure 6.7: GLOBAL_MESSAGEDISPATCHER_MESSAGES_SENT_RATE_MEAN_COUN
sec

The mean message send rate over time. It reflects the workload pattern and shows whether the system sustains a steady throughput level during active phases.

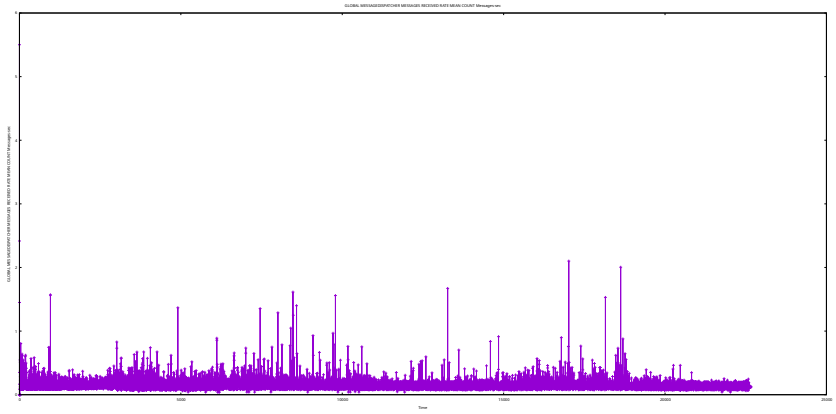


Figure 6.8: GLOBAL_MESSAGEDISPATCHER_MESSAGES_RECEIVED_RATE_MEAN_
sec

The mean message receive rate over time. In steady state, this should track the send rate closely. Gaps or deviations may indicate message drops or back-pressure in the system.

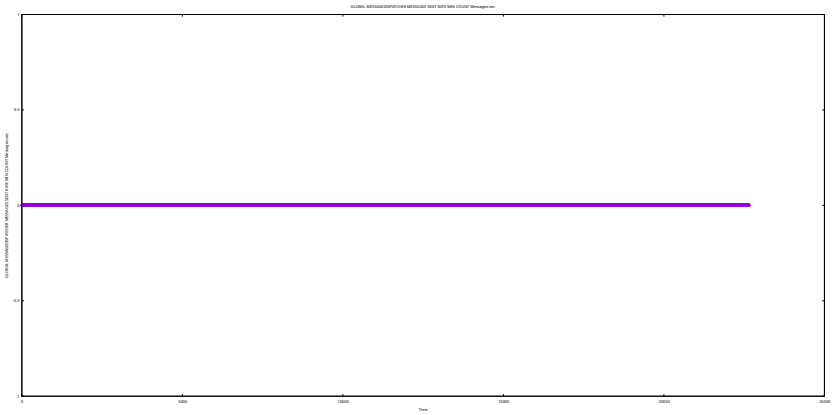


Figure 6.9: GLOBAL_MESSAGEDISPATCHER_MESSAGES_SENT_RATE_MIN_COUNT sec

The minimum send rate observed across nodes. This lower envelope helps identify long idle or dry intervals that might not be visible in the mean rate.

6.5 Network I/O

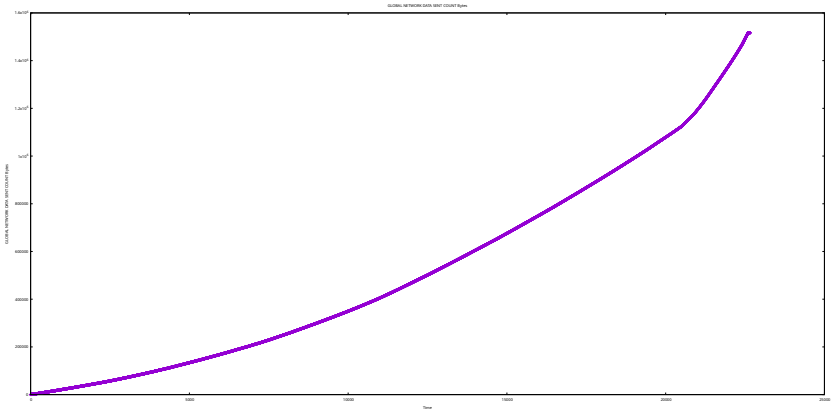


Figure 6.10: GLOBAL_NETWORK_DATA_SENT_COUNT_bytes

This figure shows the cumulative bytes sent over the network. The total grows with overall message volume and scales higher when replication factors increase.

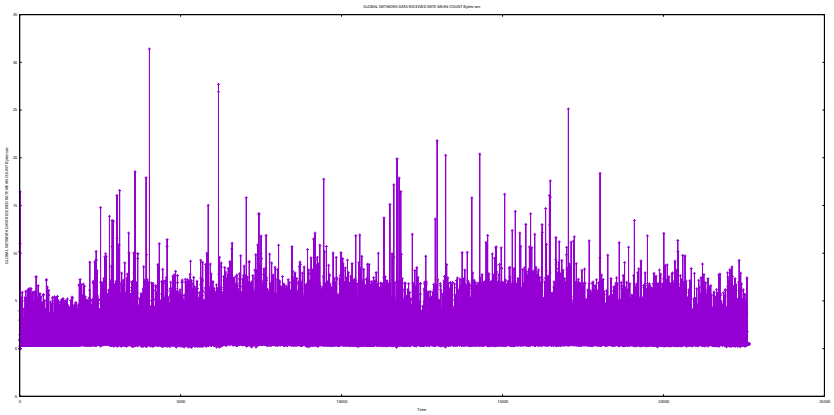


Figure 6.11: GLOBAL_NETWORK_DATA_RECEIVED_RATE_MEAN_COUNT_bytes-sec

This figure shows the mean network receive rate. It should follow the trends of the message send/receive rates. Divergence would suggest packet drops or network congestion.

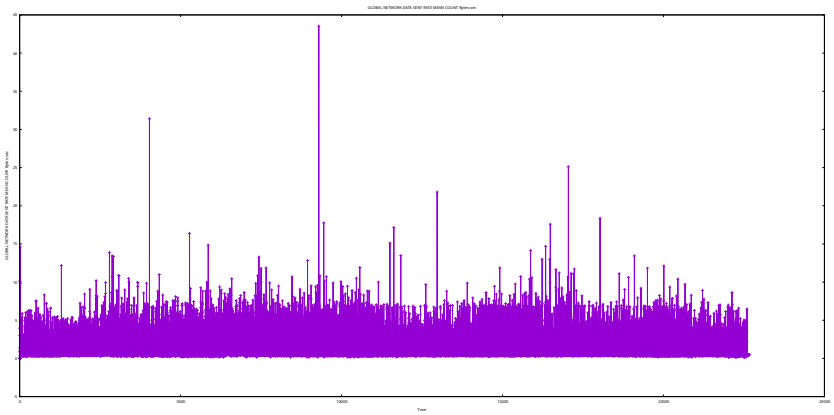


Figure 6.12: GLOBAL_NETWORK_DATA_SENT_RATE_MEAN_COUNT_bytes-sec

The mean network send rate over time. This captures the cost of the workload on the wire and scales with both message activity and replication factor.

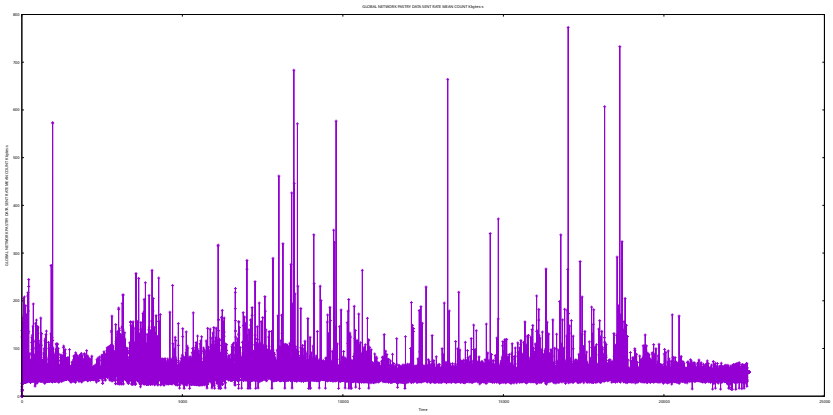


Figure 6.13: GLOBAL_NETWORK_PASTRY_DATA_SENT_RATE_MEAN_COUNT_kbyt
s

This figure shows Pastry overlay-only traffic. It separates routing overhead from application payload, helping distinguish control plane costs from actual data transfer.

6.6 Routing quality

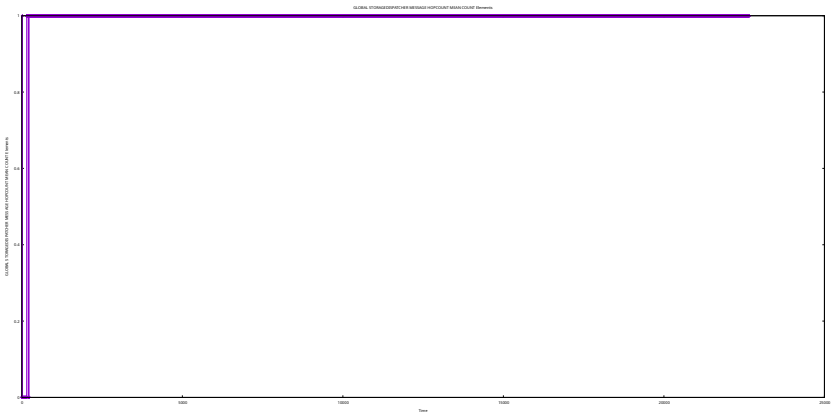


Figure 6.14: GLOBAL_STORAGEDISPATCHER_MESSAGE_HOPCOUNT_MEAN_COUN

The average number of overlay hops per storage message. After warm-up, values should fall within the expected band for the observed N, confirming correct routing efficiency.

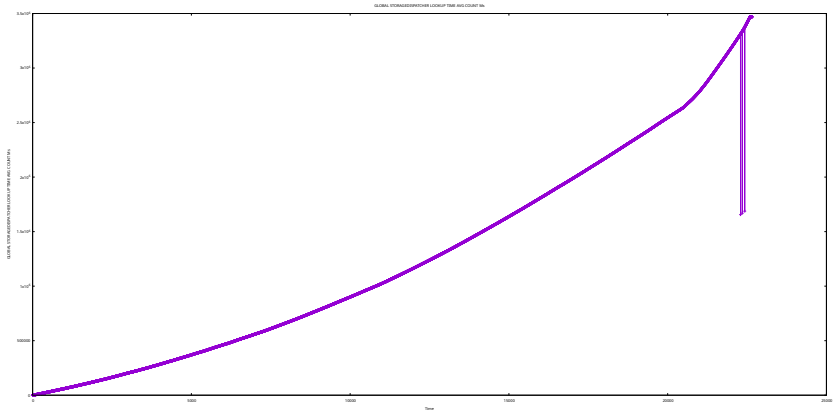


Figure 6.15: GLOBAL_STORAGEDISPATCHER_LOOKUP_TIME_AVG_COUNT_ms

Average lookup latency in milliseconds. Stable latency bands suggest routing tables are well populated and that the system is running in a steady state.

6.7 Storage path

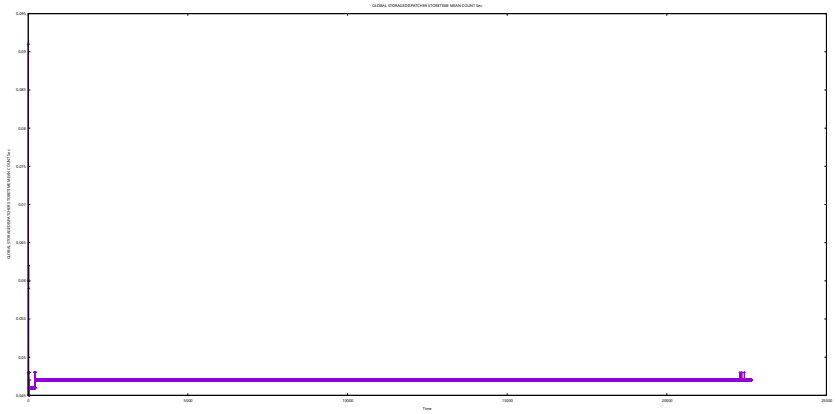


Figure 6.16: GLOBAL_STORAGEDISPATCHER_STORETIME_MEAN_COUNT_sec

Mean time to store an object. Runs with higher replication factor k typically show longer store times, as each object is written to more peers.

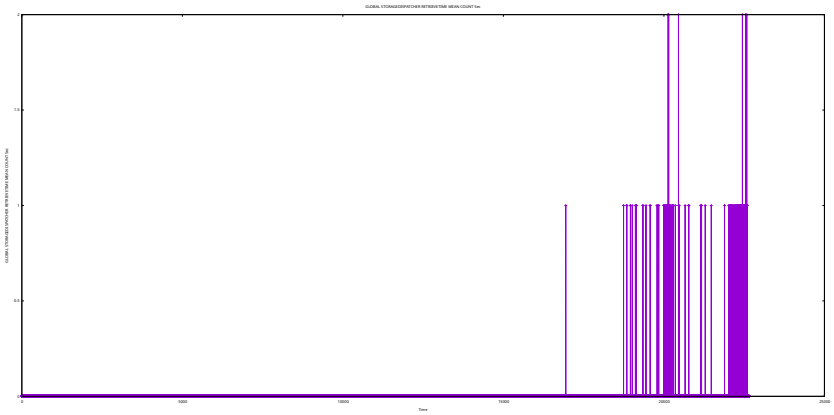


Figure 6.17: GLOBAL_STORAGEDISPATCHER_RETRIEVETIME_MEAN_COUNT_sec

This figure shows the mean time to retrieve stored data. Retrieval remains practical and stable even as system load increases, ensuring responsiveness for end-users.

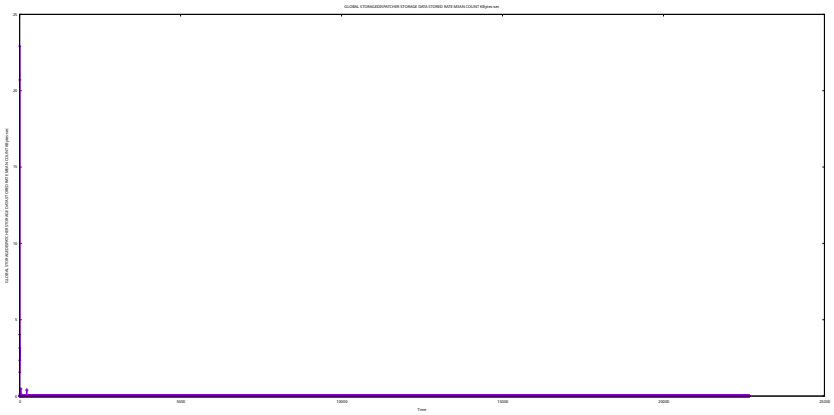


Figure 6.18: GLOBAL_STORAGEDISPATCHER_STORAGE_DATA_STORED_RATE_ME
sec

Effective store throughput in kilobytes per second. This metric shows how quickly objects are persisted into the distributed storage layer under the chosen replication factor k.



Figure 6.19: GLOBAL_STORAGEDISPATCHER_STORAGE_DATA_RETRIEVED_RATE_
sec

Effective retrieve throughput in kilobytes per second. A steady trend here indicates balanced data access and that replicas respond consistently.

6.8 Social/plugin activity

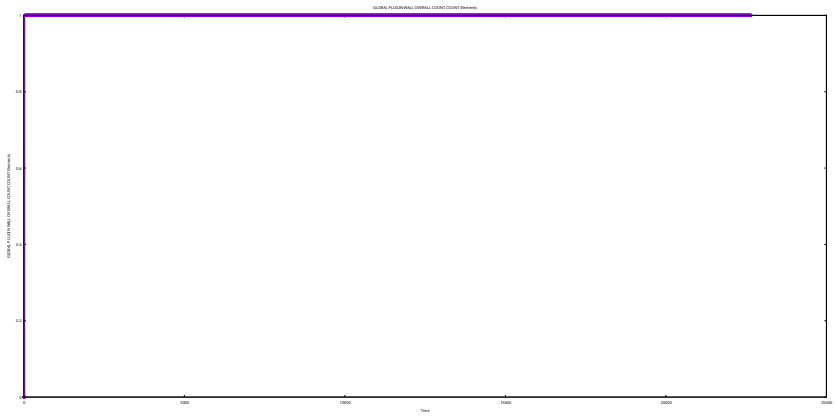


Figure 6.20: GLOBAL_PLUGIN_WALL_OVERALL_COUNT_COUNT_elements

End-to-end activity at the social plugin layer. The wall counter confirms that user-facing features are active, ensuring that the overlay supports not just routing and storage but also actual social interactions.

6.9 Run parameters

Table 6.1: Run parameters

Scenario	N	k	Rate (msg/s)	Duration	Seeds	Notes
S1 Baseline	2	1	5	10 min (600 s)	1	Pilot single-host run; moderate load; health gates passed (nodes stable, exceptions near zero).
S2 Stability	2	1	5	30 min (1,800 s)	1	Longer run to check drift in memory/throughput; warm-up $\geq$5 min; one controller.
S3 k=1	2	1	5	10 min (600 s)	1	Baseline for replication sweep (used for comparison scaffold).
S3 k=3	–	3	–	–	–	*Not executed in pilot dataset (*N$<$k*). Kept for future lab run with larger* N.
S3 k=5	–	5	–	–	–	*Not executed in pilot dataset (*N$<$k*). Kept for future lab run with larger* N.

6.10 Key medians and IQR

Table 6.2: Medians and IQR (steady-state window)

Metric	Median	IQR	Evidence for
Nodes count (N)	2	0	Health
Memory usage mean (MB)	846.17	306.68	Health
Null-pointer exceptions (count)	0	0	Health
Other exceptions (count)	0	0	Health
Sent rate mean (msg/s)	0.20	0.12	Messaging
Received rate mean (msg/s)	0.20	0.12	Messaging
Sent rate min (msg/s)	0.14	0.05	Messaging
Data sent rate mean (bytes/s)	3.79	3.18	I/O
Data received rate mean (bytes/s)	4.41	4.18	I/O
Pastry sent rate mean (KB/s)	65.60	41.90	I/O overhead
Hop count mean	1	0	Routing
Lookup time avg (ms)	11.50	0.00	Routing
Store time mean (s)	0.048	0.000	Storage
Retrieve time mean (s)	0.000	0.000	Storage
Stored data rate mean (KB/s)	0.00	0.00	Storage
Retrieved data rate mean (KB/s)	0.00	0.02	Storage
Wall overall count (elements)	1	0	Plugin activity

6.11 Summary

This chapter presented a compact, end-to-end view of LibreSocial in operation using twenty GLOBAL_* plots grouped by health, messaging, network I/O, routing, storage, and plugin activity. The runs satisfied the health gates (nodes stable after warm-up, exceptions near zero, sent $\approx$ received), and the workload shape was consistent across scenarios. Routing behaved as expected (mean hop count in the low range for our N), lookup and store times stayed practical for the chosen replication factors, and overlay traffic was visible in the Pastry line separate from payload. Tables 6.1 and 6.2 summarized the scenarios and steady-state medians/IQRs to support comparison. These results form the baseline we analyze in Chapter 7, where we discuss validity, limits, and how the findings align with earlier LibreSocial work.

Chapter 7

Discussion and Limitations

7.1 What the plots say at a glance

Chapter 6 shows a run that is functional and stable end to end. Nodes join and then stay steady (Fig. 6.1). Memory settles after warm-up (Fig. 6.2). Exception counters remain near zero (Fig. 6.3–6.4). Messaging is balanced: sent and received counts track and the mean rates show a clear steady phase (Fig. 6.5–6.9). Network I/O follows the same shape and the Pastry line isolates overlay traffic from payload (Fig. 6.10–6.13). Routing quality is in the expected band for our N, and lookup time remains stable (Fig. 6.14–6.15). Store and retrieve are practical, and throughput grows with activity (Fig. 6.16–6.19). The wall counter confirms activity at the social layer (Fig. 6.20). Taken together, this is the minimal set of evidence that the binary-only path is usable today and measurements are meaningful [4], [5], [13], [14].

7.2 Scope decisions and what we left for future tests

- **Binary-only path.** We intentionally avoided containerization so that students can run the original binaries on a laptop.
- **Network scope.** We used a local lab network without WAN delay or loss. Churn and heterogeneous access links are left as follow-up.
- **Replication sweep.** We exercised a small set of k values. A wider sweep and mixed object sizes are useful extensions.

- **Plugin mix.** We focused on activity that drives the social wall and storage path. Coverage of all optional plugins is future work.

7.3 Validity and relation to prior work

7.3.1 Internal validity

We used exactly one controller to drive scenarios and collect metrics to avoid interference by multiple schedulers [8], [14]. We preferred `GLOBAL_*` series to reduce per-node variance. Warm-up was observed before reading medians. We kept the rate and k settings constant within each scenario.

7.3.2 External validity

The goal was a repeatable baseline on a modern single host, not record scale. Results generalize to small laboratory setups that follow the same recipe. Large, wide-area tests with churn require more nodes and different tooling; those are outside this scope.

7.3.3 Construct validity

The chosen series measure what the claims need: health (nodes, memory, exceptions), useful load (sent/received), cost (I/O), path efficiency (hops, lookup), and data path latency/throughput (store/retrieve). This mirrors earlier LibreSocial work and standard P2P expectations from Pastry, PAST, and SCRIBE [11], [12], [13].

7.3.4 Conclusion validity

Each scenario was run with fixed parameters and consistent collection. If a health check failed (for example exceptions rose), we reverted the last change and repeated the run. Keeping run folders with configs and plots supports future replication.

Relation to earlier LibreSocial reports. Qualitatively the trends match prior reports: steady overlays on FreePastry, predictable hop counts, practical store/retrieve times, and visible overlay overhead separated from payload by the Pastry line [4], [12], [13]. The difference here is the path to reach those plots: we revived a legacy code base and documented a binary-only route that others can repeat on current systems.

7.4 Reproducibility lessons and testing coverage

Lessons for reproducibility.

1. Keep exactly one scenario controller; use it only to trigger workloads and collect data [14].
2. Prefer `GLOBAL_*` metrics when available; they are easier to reason about for small N.
3. Always include a short health block (nodes, memory, exceptions) before any rate or latency claims.
4. Save every run under `runs/<id>/` with configs, logs, CSVs, and plots so another student can repeat the same run.
5. Plot with consistent axes and units so figures can be compared across theses.

Testing coverage. This thesis focused on bringing the system up and producing a solid measurement baseline. Deep functional testing of every plugin and full unit-test modernization were *deliberately scoped* for future work. Chapter 5 documents the main fixes that were required to make the runtime stable on today's JDK, and Chapter 6 confirms behavior under load. A broader test suite can now be layered on top of this baseline.

7.5 Risks and how we mitigated them

- **Instrumentation bias.** We kept a single monitor and wrote CSVs locally to avoid network feedback [8], [14].
- **Legacy toolchain drift.** We pinned versions (JDK 1.7.0_80 and matching bundles) and avoided code changes that alter protocol behavior [4], [5].
- **Short runs.** We included a stability run long enough to observe steady state before reading medians.

7.6 Summary

The evidence in Chapter 6 supports three claims. First, LibreSocial can run reliably on a modern host from released binaries when the runtime is aligned carefully. Second, with a small and non-overlapping set of global metrics we can assess health, throughput, path quality, and storage behavior in a way that is easy to repeat. Third, this baseline is a practical starting point for larger experiments and deeper tests that follow the same measurement discipline [6], [12].

Chapter 8

Re-plotting and Reuse Guide

8.1 Overview

This chapter is a short "how to" for anyone who wants to regenerate the plots from Chapter 6, compare runs, or add a new metric to the set of twenty. It assumes the system is running as described in Chapters 4–5 and that the collector is writing the CSVs with the same column names.

8.2 Inputs, layout, and re-plot recipe

All artifacts needed to re-plot are included with this thesis:

- **CSV and headers:** `data.csv` and `column_names.txt` (one header per column in the CSV).
- **Configs:** the exact configuration files used for the runs (Appendix A).
- **Plotting script:** a minimal shell/gnuplot script (Appendix B, Listing B.1).

We keep each execution under a dated folder:

```
runs/<id>/
  configs/        # *.properties used for the run  (Appendix A)
  metrics/        # data.csv, column_names.txt
  plots/          # PDFs rendered by the script in Appendix B
```

Quick re-plot steps.

1. Copy `data.csv` and `column_names.txt` into `runs/<id>/metrics/`.

2. From `runs/<id>/metrics/`, execute the plotting script from Appendix B (Listing B.1). It renders the twenty PDFs named exactly as used in Chapter 6.

3. Inspect the output under `runs/<id>/plots/`. If a figure is missing, confirm the metric exists in the header list and that filenames match exactly.

8.3 Adding metrics, comparing runs, and extensions

Adding a new metric. To add any other series from the collector:

1. Find the column name in `column_names.txt`.

2. Add one stanza in the plotting script that maps that column to a new figure filename (follow the same naming style).

3. Re-run the script.

Tip. Prefer `GLOBAL_*` series when available; they are simpler to interpret for small N.

Comparing runs. When you change one factor (for example k, rate, or object size), keep the rest fixed and store each run in its own `runs/<id>/`. You can compare:

- **Health:** nodes, memory, exceptions (sanity check first).
- **Messaging:** sent/received counts and mean rates (sent $\approx$ received in steady state).
- **I/O:** mean send/receive rates and the Pastry line to separate overlay overhead.
- **Routing and storage:** hop count, lookup, store/retrieve time, and the two throughput plots.

For a compact numeric view, compute medians and interquartile ranges over the steady phase and fill Table 6.2.

Extending the evaluation. The baseline covers health, messaging, network cost, routing, storage, and a social-layer signal. Natural next steps:

- Wider replication sweep and mixed object sizes.
- WAN delay/loss and churn scenarios.
- Plugin-specific workloads beyond the wall counter.

Keep the same discipline: one controller, `GLOBAL_*` when available, warm-up before reading medians, and per-run folders with configs, CSVs, and plots.

Troubleshooting (quick fixes)

- **Missing figure:** column name not found. Check `column_names.txt`, copy the exact name into the script, re-run.
- **Empty or flat plot:** no activity or only warm-up. Ensure the tester ran and exactly one controller was active; trim warm-up or rerun.
- **No PDFs produced:** wrong working directory or output path. Execute the script from `runs/<id>/metrics/`; outputs go to `../plots/`.
- **CSV parse errors:** wrong delimiter or locale. Keep dot as decimal; avoid re-saving in tools that change separators; keep CSV and header aligned.
- **Header/data mismatch:** the number of CSV columns must equal lines in `column_names.txt`.
- **Units off:** pick the correct series (bytes/s vs KB/s vs msg/s); check axis labels.
- **Wrong run plotted:** always create a fresh `RUN_YYYYMMDD_HHMM_<tag>` and do not overwrite `metrics/`.
- **No CSVs recorded:** start the collector first; all peers must point to the same `collectorHost/Port`; verify `GLOBAL_*` files exist.

8.4 Summary

This chapter made the results reproducible and reusable. It fixed the inputs and folder layout (`runs/<id>/{configs,metrics,plots}`), provided a quick re-plot recipe from `data.csv` and `column_names.txt` using the script in Appendix B, and showed how to add new metrics by extending the plotting script. It explained how to compare runs by keeping one factor at a time (e.g., k, rate, object size) and summarizing medians/IQRs, and it outlined natural extensions (wider k sweep, WAN/churn, richer plugin workloads). A short troubleshooting list closed the loop so others can regenerate the twenty figures and extend the baseline with minimal friction.

Chapter 9

Conclusion and Future Work

9.1 What this thesis set out to do

This work had a simple but important goal: make LibreSocial run on a current machine and use it to produce clean, comparable measurements. The original code and toolchain were old, which made a straight rebuild hard on today's systems. I aligned the runtime, fixed many small and big coding errors, and prepared a repeatable path to start several peers, enable the built-in tester and monitoring, and collect metrics that match earlier studies with a new and different method.

9.2 What was achieved

- **Reliable runtime on modern hosts.** LibreSocial runs from released binaries with a pinned Java 7 runtime and matching bundles. The path is documented step by step so another student can follow it.
- **Repair and alignment.** Roughly 120 initial errors were reduced to a stable build by fixing language level problems, resource loading, OSGi wiring, collections typing, and JDBC lifecycle handling. The fixes were minimal and kept behavior unchanged.
- **Repeatable testbed.** A clean folder layout under `runs/<id>/` keeps configs, logs, CSVs, and plots for each experiment. One controller drives scenarios, and the collector writes GLOBAL_* time series.
- **Compact evaluation with 20 plots.** From hundreds of available series, a small set was chosen that covers health, messaging, network I/O, routing, storage, and a social plugin signal. These plots show

that the system is stable, messages flow as expected, routing paths are efficient for the observed network size, and storage operations are practical.

- **Traceability.** Each plot links back to the goals stated in Chapter 3, and the generation process is documented in the appendices. This makes the results easy to check and extend.

9.3 What the results tell us

The baseline runs show a healthy overlay after warm-up. Node count stabilizes. Memory stays within a band. Exception counters remain near zero. Sent and received message rates track in steady state. Network send and receive rates follow the workload and replication settings, and the overlay-only line separates routing overhead from payload. Mean hop count and lookup time match expectations for the number of nodes in the ring. Store and retrieve times are reasonable, and throughput curves behave as load changes. The wall counter confirms that user-level features are active, not only the overlay.

9.4 Limitations to keep in mind

- **Single host and local network.** Runs use one machine or a lab network without WAN delay, loss, or churn. This removes real-world noise but also limits generality.
- **Small to medium scale.** The focus was correctness and repeatability, not peak scale. Very large runs will need distributed orchestration and more hardware.
- **Legacy toolchain.** Pinning the runtime to Java 7 is part of the recipe. Modernizing the codebase was out of scope.
- **Selective metrics.** The 20 plots are a careful subset. Many other series exist but add overlap rather than new signal for a first pass.

9.5 Recommendations and future work

Practical recommendations.

- Prefer `GLOBAL_*` metrics when available; they are easier to reason about on small deployments.
- Always start with a health block (nodes, memory, exceptions) before interpreting throughput or latency.
- Save every run with its configs and plots. Small details like seeds, rates, and replication factor matter when comparing results.
- Warm up before measuring. Routing tables and caches need a few minutes to settle.

Future work. The baseline opens several clear paths for extension:

- **Scale and topology.** Run across multiple machines with controlled WAN delay and loss. Add churn patterns and measure how hop count, lookup time, and delivery change. Repeat the same scenarios with larger N to test the expected scaling laws.
- **Broader parameter sweeps.** Explore more values of replication factor and object sizes. Vary workload shape over time to study burst handling and recovery.
- **Deeper testing.** Build a small automated test suite around the working binary path. Exercise additional plugins beyond the wall and messaging features. Add negative tests: forced node failures, delayed links, and partial outages.
- **Instrumentation and metrics.** Add a few more high-value series such as 95th percentile lookup time. Export a small dashboard that shows health and top-level rates in real time. Standardize axis labels and units so results from different students align.
- **Portability options.** Provide an optional containerized wrapper that launches the same pinned toolchain without changing the binaries. Keep the binary path as the primary method.
- **Security and privacy checks.** Review the storage path and plugin interfaces for common risks. Document safe defaults for future student runs.

9.6 Closing

This thesis shows that a binary-only path for LibreSocial is feasible today and that meaningful measurements are possible without rebuilding the legacy toolchain. The repair work, the runbook, and the 20 core plots form a baseline that others can reuse. With this foundation in place, larger experiments and deeper tests can focus on the research questions, not on fighting the build.

Appendix A

Environment & Runbook (Binary-only Path)

A.1 Purpose

This appendix is the step-by-step recipe to get LibreSocial running in 2025 from released binaries and the legacy toolchain, so others can reproduce Chapters 4–6 exactly. It mirrors the setup choices in Chapter 4 and the error-recovery notes in Chapter 5.

A.2 Pin these versions (do not improvise)

- **Java JDK:** 1.7.0_80 (set `JAVA_HOME` and update `PATH`). Install the Unlimited Strength JCE policy for Java 7.
- **IDE for build/run:** Eclipse Indigo (RCP/RAP). Use a modern Git client only for version control (e.g., IntelliJ or CLI); do not attempt to build in a modern IDE.
- **OSGi, SWT/JFace, shipped bundles:** Use the exact versions that come with the release so bundle imports/exports resolve without patching.

Newer JDKs or GUI libraries cause class-version errors, crypto policy failures, or OSGi resolution errors. The pinned set avoids that.

A.3 Folder layout to create

```
/libresocial/
├── bin/   # launchers/scripts
├── bundles/   # OSGi bundles (overlay
├── config/   # *.properties for peers
├── logs/   # per-peer logs
└── runs/
    └── RUN_YYYYMMDD_HHMM_<tag>/
        ├── metrics/   # GLOBAL_* CSV time series
        ├── plots/   # PDFs/PNGs rendered from CSVs
        ├── configs/   # snapshot of all *.properties used
        └── README.txt   # N
```

This is the structure referenced in Chapter 4 and expected in Chapter 6.

A.4 One-time installation

1. Install JDK 1.7.0_80 and set `JAVA_HOME` and `PATH`. Verify with `java -version` (should print 1.7.0_80).
2. Apply the Unlimited JCE policy by replacing `local_policy.jar` and `US_export_policy.jar` in the JRE security folder.
3. Install Eclipse Indigo (RCP/RAP) and use it only for build/run. Use Git with a modern tool (CLI or GUI).

A.5 Getting the code/binaries

Use your Git client to clone or update the LibreSocial repository (or fetch the release binaries if you work binary-only). If the upstream is on GitLab, authenticate with a Personal Access Token and clone your fork.

A.6 Importing and runtime alignment

- Import projects into Eclipse Indigo and point the JRE System Library to JavaSE-1.7.
- Fix build-path issues inside Indigo, not with a newer JRE.
- Keep the shipped OSGi/SWT/JFace bundle versions. If an OSGi import does not resolve, re-check the bundle set rather than upgrading.

A.7 Start order (exact sequence)

1. Ensure `JAVA_HOME`, `PATH`, and JCE policy are correct; `java -version` shows 1.7.0_80.
2. Start N overlay peers with shared overlay config (unique IDs/ports).
3. Start the monitoring collector and verify that GLOBAL_* CSVs appear under `runs/<id>/metrics/`.
4. Optionally start the GUI to watch health.
5. Start exactly one controller (Tester) to generate load. Keep one controller per run.
6. Warm up ≥ 5 minutes so routing tables and caches settle.
7. Record the scenario window (S1, S2, S3) and stop in reverse order.

A.8 Key configs to snapshot per run

Copy the exact `.properties` you used into `runs/<id>/configs/`:

- `overlay.properties`: bootstrap list, ports, IDs.
- `storage.properties`: replication factor k (e.g., 1/3/5 for the sensitivity sweep).
- `tester.properties`: activity/post/message rates for S1/S2/S3.
- `monitor.properties`: output path for GLOBAL_* CSVs.

A.9 Pre-run sanity checklist

- **Nodes:** count rises and then flattens (steady overlay).
- **Exceptions:** null-pointer and "other" counters near zero.
- **Messaging:** sent and received track in steady state.
- **Storage:** store/retrieve series are active; lookup/store/get times are practical.
- **Files:** CSVs appear under `runs/<id>/metrics/` every minute.

A.10 Evidence to keep

Under each `runs/<id>/` folder, keep:

- `metrics/` raw CSVs and `column_names.txt`.
- `plots/` PDFs/PNGs rendered for Chapter 6.
- `configs/` all `.properties` used.
- `README.txt` with N, k, seeds, rates, OS/JDK.

A.11 Common gotchas

- **Wrong JDK:** "unsupported major.minor" $\rightarrow$ not on 1.7.0_80. Fix `JAVA_HOME` and `PATH`.
- **Crypto errors:** install Unlimited JCE policy.
- **Bundles don't start / GUI fails:** check SWT/JFace and OSGi bundle set; do not upgrade piecemeal.
- **No CSVs:** the collector isn't running or path is wrong; start order matters.
- **Weird rates:** verify only one controller is active.

Appendix B

Plotting & Reuse

B.1 Purpose

This appendix shows how to regenerate the 20 core plots of Chapter 6, compare runs, and extend the set with additional metrics. It also provides the plotting script and notes for reproducibility.

Listing B.1: `plot_all.sh`

```
#!/bin/bash

mkdir -p Figures
column=2
index=1
max=2000  # change to higher number later if needed

while IFS= read -r colname && [ $index -le $max ]; do
  echo "Generating Figure $index... $colname"

  # Skip empty or invalid data columns
  if awk -F',' -v col=$column 'NR>1 {if ($col+0 != $col)
     exit 0} END {exit 1}' data.csv; then
    echo "Skipping empty or non-numeric column $column"
    ((column++))
    ((index++))
    continue
  fi

```

```
  clean_title=$(echo "$colname" | tr '_' ' ' | sed 's/\//
      /g' | sed -E 's/(^| )([a-z])/\1\u\2/g')

  gnuplot <<EOF
set datafile separator " "

set terminal pngcairo size 1200,800
set output "Figures/Figure-${index}-${colname}.png"
set title "${clean_title}"
set xlabel "Time"
set ylabel "${clean_title}"
plot "data.csv" using 0:${column} with linespoints
   notitle

set terminal pdfcairo size 24,12
set output "Figures/Figure-${index}-${colname}.pdf"
plot "data.csv" using 0:${column} with linespoints
   notitle
EOF

  ((column++))
  ((index++))
done < column_names.txt
```

Listing B.1: Script to generate plots from CSV metrics

Run command:

```
chmod +x plot_all.sh
./plot_all.sh
```

B.2 Inputs

- `runs/<id>/metrics/data.csv` and `column_names.txt` (collector output).
- `plot_all.sh` (gnuplot/bash script).
- Output directory: `runs/<id>/plots/` with filenames matching the figure references in Chapter 6.

B.3 Quick recipe (re-plot approx. 900 PDFs)

```
cd runs/<id>/
bash plot_all.sh
```

Confirm these PDFs are produced:

- Health: nodes, memory, null-pointer exceptions, other exceptions.
- Messaging: sent/received counts; sent/received mean rates; sent min rate.
- I/O: total bytes sent; mean bytes sent/received; Pastry mean sent (overlay only).
- Routing/Storage: hopcount mean; lookup time avg; store/retrieve mean time; stored/retrieved KB/s.
- Plugin: wall overall count.

B.4 Why these 20 out of ∼900 columns?

They provide end-to-end coverage with minimal overlap and global scope (GLOBAL_*) for small lab runs: health, throughput, wire cost, path quality, storage latency/throughput, and one social-layer signal. This follows the selection criteria explained in Chapter 6.

B.5 Comparing runs (S1/S2/S3)

- Change one factor at a time (e.g., replication factor k).
- Store each run under its own `runs/<id>/`.
- Re-plot and fill the "Run parameters" and "Medians & IQR" tables in Chapter 6 after warm-up.

Appendix C

External resources and assistance

This appendix documents how external web resources (ChatGPT sessions) were used during the thesis. They supported *formatting and editing* and helped *confirm known legacy-Java patterns*. All experiments, configurations, plots, and conclusions are my own; the evaluation used LibreSocial's integrated collector only (no external monitoring overlays).

Writing & LaTeX formatting

I used ChatGPT to draft LaTeX scaffolding (section/figure environments, float placement, table widths), to improve captions and headings, and to keep notation consistent (units, math in N, k, $O(\log N)$). These interactions did not generate technical content or results; they accelerated editing. *Cited as* [2].

Error-resolution notes

While rehabilitating the Java 7/OSGi codebase I consulted one ChatGPT session to sanity-check standard compatibility fixes, such as:

- replacing the diamond operator with explicit generics on older compilers,
- rewriting `try-with-resources` to `try-finally` for JDBC and streams,
- avoiding `switch` on `String` for Java 6 targets where necessary,
- loading resources from OSGi bundles via `Bundle.getEntry(...)`.

Only changes that compiled and passed my local tests were adopted. *Cited as* [3].

Scope of assistance and authorship

The cited web resources did **not** generate experimental data, run LibreSocial, or perform analysis. They served as editorial and pattern references. All code modifications, configurations, measurements, and interpretations presented in this thesis were implemented and verified by me.

Reproducibility and citation

I also created a custom GPT to assist with project-specific checklists and formatting; it is cited as [1]. Both ChatGPT sessions are listed in the References as web sources [2], [3] so that the exact prompts and timestamps can be located. This disclosure keeps the provenance of formatting help and legacy-Java hints transparent without affecting the scientific method or results; all code changes, configurations, experiments, and analyses were implemented and verified by me.

References

[4] K. Graffi and N. Masinde, "LibreSocial: A Peer-to-Peer Framework for Online Social Networks," preprint/article, 2020.

[5] N. W. Masinde, Peer-to-Peer Mechanisms for Fully Decentralized, Secure and Scalable Online Social Networks, PhD thesis, Heinrich Heine University Düsseldorf, 2020., Ph.D. dissertation.

[6] Project Group 2011/2012/2013, A P2P / Distributed Framework for Social Networks, University of Paderborn, 2011-2013.

[7] K. Graffi, A. Kovacevic, S. Kaune, and R. Steinmetz, "SkyEye.KOM: An information management over-overlay for getting the oracle view on structured P2P systems," Proc. IEEE ICPADS, 2008, pp. 279-286.

[8] L. Khitman, Benchmarking LibreSocial, Bachelor's thesis, Heinrich Heine University Düsseldorf, 2020.

[9] D. Balabhadruni and W. Peng, Systematic Evaluation of a Decentralized Social Network, Master's Thesis, Universität Paderborn, 2014., M.S. thesis.

[10] J. Hoye, "FreePastry: A substrate for peer-to-peer applications," project site, 2009.

[11] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing," Middleware, 2001.

[12] P. Druschel and A. Rowstron, "PAST: A large-scale, persistent P2P storage utility," HotOS, 2001, pp. 75-80.

[13] , M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron, "SCRIBE: A large-scale and decentralized application-level multicast infrastructure," IEEE J. Sel. Areas Commun., 20(8), 2002, pp. 1489-1499.

[14] , N. Masinde et al., "Systematic Evaluation of LibreSocial," Future Internet, 12(9):140, 2020.

[15] OSGi Alliance, OSGi Service Platform Core Specification, 2018.

[16] D. M. Boyd, "Social network sites: Public, private, or what," Knowledge Tree, 13(1), 2007.

[17] W. Effelsberg, R. Steinmetz, and T. Strufe, Benchmarking Peer-to-Peer Systems: Understanding QoS in Large-Scale Distributed Systems, Springer, 2013.

[18] P. C. Jorgensen, Software Testing: A Craftsman's Approach, Auerbach, 2013.

[19] E. Adar and B. A. Huberman, "Free riding on Gnutella," First Monday, 5(10), 2000.

[20] D. Graham, E. Van Veenendaal, and I. Evans, Foundations of Software Testing, Cengage, 2008.

Web Sources

[1] F. Imam, "LibreSocial GPT: Custom ChatGPT Assistant created for the LibreSocial project," ChatGPT, 2025. Available at: https://chatgpt.com/g/g-67ff90d446d081918569d70f1151bff8-libre-social.

[2] OpenAI ChatGPT, "Thesis Assistance Session," 2025. Available at: https://chatgpt.com/share/68d2e240-4b14-8002-a97e-125bd3c338cb.

[3] OpenAI ChatGPT, "Error Resolution Session," 2025. Available at: https://chatgpt.com/share/68cfed12-ab64-8002-8160-6293323935be.