Push vs Pull:
Keeping P2P Data Synchronized

CSC 466 | Project Proposal | University of Victoria

Rahil Wijeyesekera | Hamzah Banday

Prabhnoor Singh | Adithyakrishna Arunkumar

October 10, 2026

1. Problem and motivation

If several peers keep a copy of the same data, a change made on one peer needs to reach the others. For example, a shared task list should eventually show the same task status on every device. The question is how to spread those changes without sending more traffic than necessary.

We want to compare two straightforward approaches. With **push**, a peer sends an update when it learns about a change. With **pull**, peers check their neighbours at regular intervals and request newer data. Our main question is: **how does each approach trade update speed for communication overhead when the update rate and network delay change?**

2. Prior work

Demers et al. [1] studied how updates can spread between replicated databases through epidemic algorithms, including push, pull and push-pull exchanges. Their discussion also points out that pull requests can keep generating traffic when there are no new updates. We will use this work as a starting point for a small practical comparison. We are not proposing a new synchronization protocol; our contribution will be a reproducible testbed and measurements of these two implementations.

3. Proposed approach

We will build a small P2P key-value store in Python, with peers running as separate processes on one laptop and communicating over TCP sockets. Each peer will hold the same small set of records, such as task IDs and completion statuses. Each key will have one designated writer and an increasing version number. This keeps conflicting edits out of the experiment while allowing different peers to create updates to their own keys.

Both implementations will use the same connected ring: each peer communicates with its two neighbours. In push mode, a new version is sent to those neighbours and forwarded when first received. Version checks stop duplicate updates from being forwarded indefinitely. In pull mode, each peer asks its neighbours for their current version lists, then requests only records with newer versions. A record learned through pull can be served to another peer at its next check.

The main runs will use eight peers. We will also make smaller checks with four and sixteen peers to see how the results change with network size. A separate script will start runs, generate updates and collect logs; it will not relay synchronization traffic between peers.

4. Experiments and evaluation

We will first verify that both methods reach the same final state after updates stop. Then we will compare low and high update rates (one update every five seconds versus one per second), pull intervals of 0.5, 2 and 5 seconds, and no added delay versus 100 ms of added one-way delay per message. Delay will be applied in the messaging layer, so this is an application-level delay experiment rather than a full Internet emulator.

For each comparison, we will keep the topology, record sizes, starting state and generated update sequence the same. Each configuration will run at least five times with different seeds, replaying each seed for both methods. We will also include a fixed idle period to measure polling traffic when nothing changes.

Measure	How we will measure it
Update propagation delay	Time from creating a version until each peer holds that version or a newer version of the same key. Report the median and 95th percentile.
Communication overhead	Total serialized application bytes and messages sent during equal-length runs, including requests, replies and duplicate updates. TCP/IP overhead is excluded.
Final convergence time	Time from the last generated update until every peer holds the final version of every key. Record a timeout if convergence does not occur.
Idle traffic	Bytes and messages per second during the period with no new updates.

Because all processes run on the same host, we can use a shared monotonic clock for timing and avoid clock synchronization errors. We will show variation across runs alongside the averages. We expect push to spread changes sooner and shorter pull intervals to reduce waiting at the cost of more requests, but we will treat these as hypotheses. The conclusions will apply to our topology and implementation, rather than every P2P system.

5. Deliverables and scope

Our deliverables will be the two synchronization implementations, scripts and instructions for repeating the experiments, CSV logs, comparison plots, and a final report and presentation. The core project will focus on stable peers and small records. Peer discovery, concurrent edits to the same key, security and large file transfers are outside this scope. If time allows, we will add a push-plus-periodic-pull variant and test whether it helps a peer catch up after reconnecting.

6. Tentative schedule

Week 1	Review prior work; build peer connections, record storage and version checks.
Week 2	Implement push and pull; check final-state correctness on a small network.
Weeks 3-4	Add workload generation, delay controls and logging; run repeated experiments.
Week 5	Analyse the results, make plots and investigate unexpected behaviour.
Week 6	Finish the report, demonstration and presentation.

Reference

[1] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart and D. Terry. "Epidemic Algorithms for Replicated Database Maintenance." Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, 1987, pp. 1-12. doi:10.1145/41840.41841.