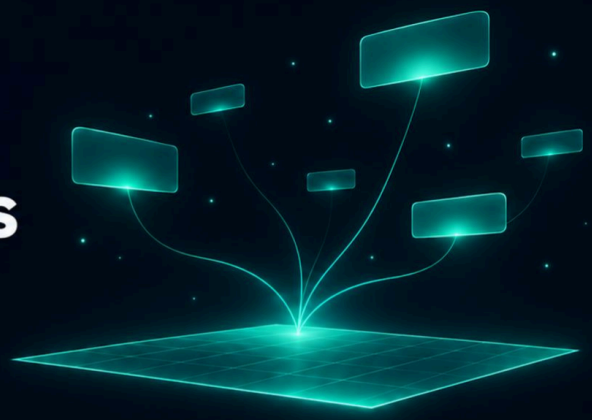
CroakOS

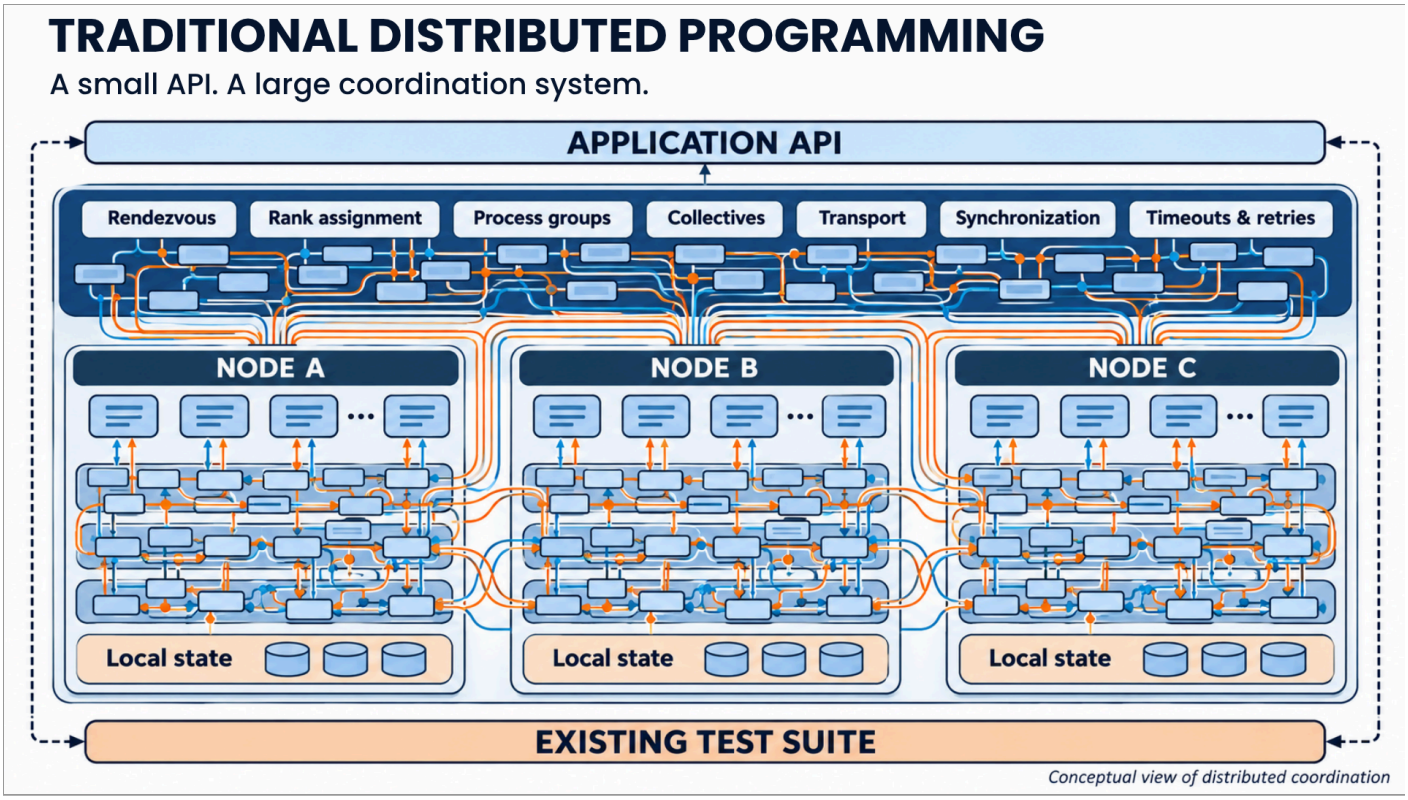
Engineering Impacts of a Distributed Operating System



John W. Fawcett
Fawcett Innovations, LLC
john@fawcettinnovations.com

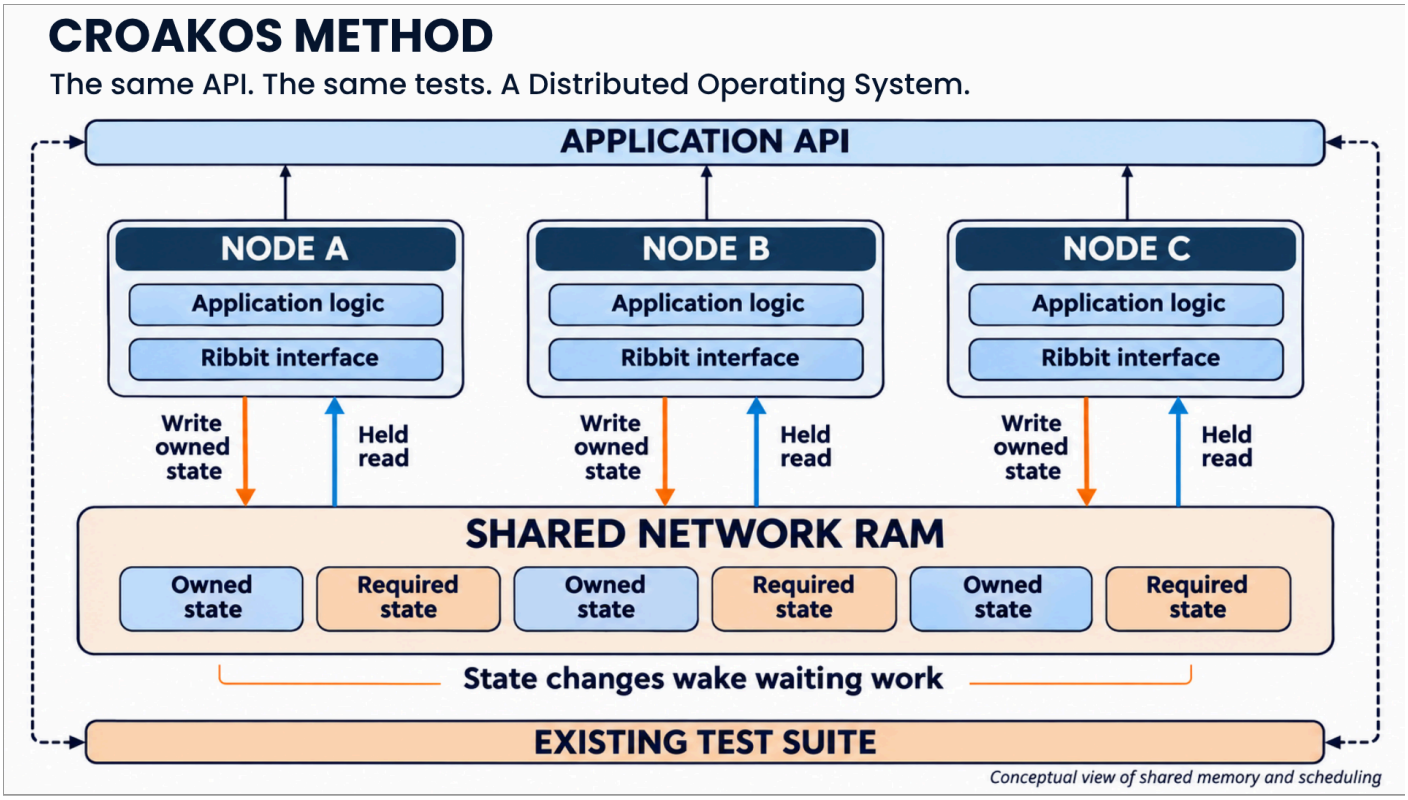
IETF 127 · HotRFC
15 November 2026

Display before the talk. Begin speaking on slide 2.



Anyone who has done distributed programming will recognize this admittedly hypothetical service topology, and you'll probably cringe at the thought of having to maintain it.

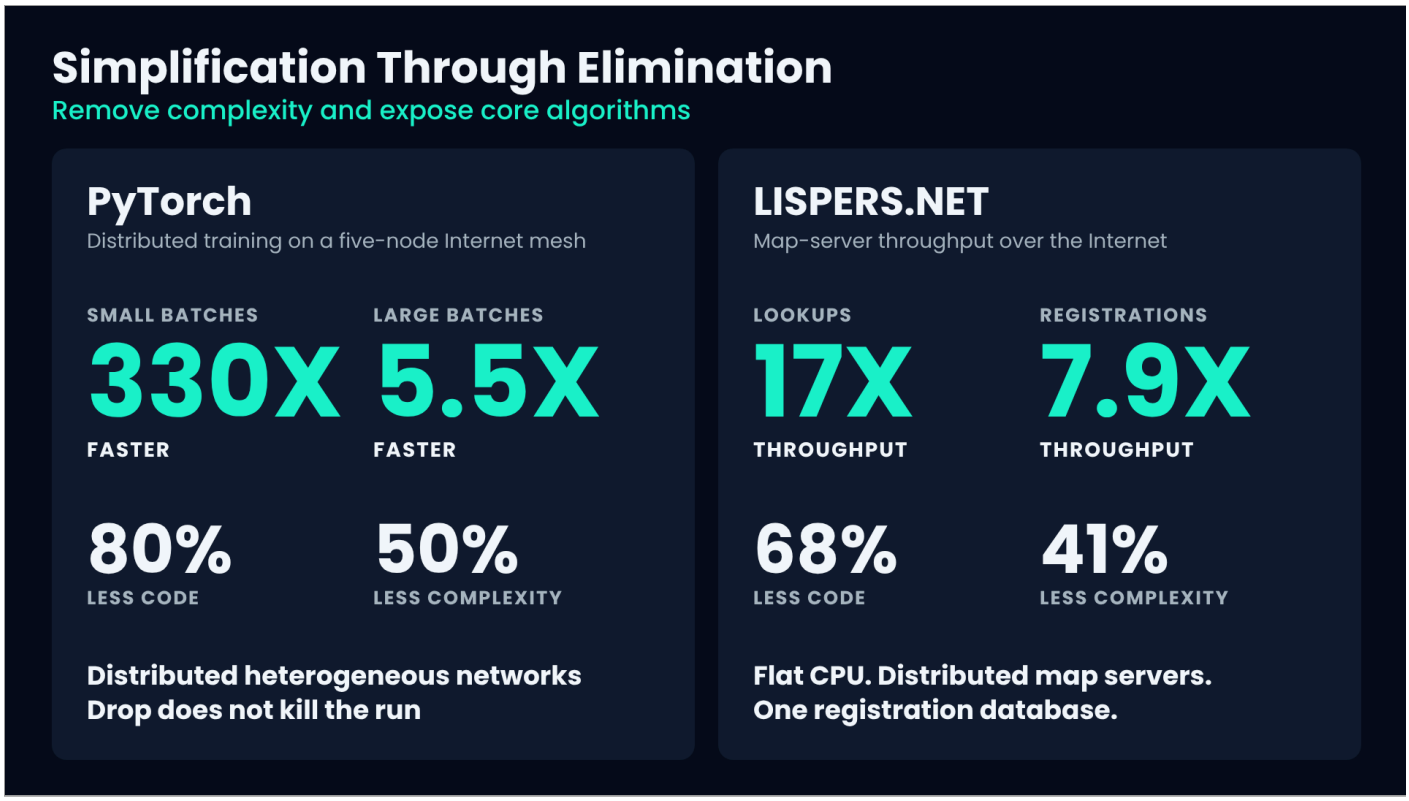
There are local copies of state, timeouts and retries, and complex synchronization code. It needs a tremendous amount of machinery, before the application even starts.



CroakOS provides a distributed operating system that eliminates huge portions of that infrastructure. It offers shared RAM and scheduling across the Internet, letting programmers treat the network as a single programming surface.

CroakOS changes the way programmers work with distributed systems. We no longer think of separate machines that have to be reached through custom protocols. Instead, we have a common memory model, familiar RAM storage and retrieval operations, and ways to trigger cross-network events when RAM changes.

Underneath, CroakOS discovers and assembles itself. Netowrks find each other over any transport, and each builds its own encrypted WireGuard overlay across the Internet. Shared RAM host addresses, and every other shared role, are set by deterministic rules, without configuration, at every split or merge. Every piece of state has exactly one writer, and consumers are independent of producers in both time and space. When a host fails, the survivors determine the replacement and the work carries on.



Existing systems are converted in situ. Existing APIs define the interface contract, and existing test suites are proof of compliance. Everything between those two is mine. I convert systems from messaging to stateful, lockless, event-driven entities, usually by removing huge swaths of code.

With PyTorch, the initial conversion maintained performance, with identical results to eight decimal places, using 80 percent less code with half the complexity. When I reorganized the algorithmic computation around the common multithreaded programming model, small-batch training across a 5-node Internet distributed mesh was 330X faster than stock DDP, and large-batch training was 5.5X. The system profiled the machines to place computation on the fastest available resource, relying on shared RAM for everything.

With Dino Farinacci's lispers.net, I cut code volume by about 70 percent and complexity by about 40 percent, while gaining 17 times the lookup throughput and 7.9 times the registration throughput over the Internet. Distributed map servers share one registration database, with flat CPU as the table grows.

The PyTorch conversion took five days. The lispers.net conversion took six, from requirements to delivery.



I have converted or built fourteen systems so far, with incredible code velocity. With the plumbing gone, I can work on the algorithms and how data moves, and I realize we are barely past "Hello, WOrld" in distributed programming.

I use AI as a coding assistant under strict lifecycle and test discipline, and the existing tests judge the result.



The code is open source. Bring your complexity: your interface, and the tests it has to pass. Let’s see what happens when we write programs instead of plumbing.

Join me for the follow-up on Tuesday, November 17, at 9:30 a.m. Pacific, 17:30 UTC.

With thanks to Dino Farinacci and John Day.