

Redundancy Without Reliability

Running a second server is meant to keep users online when the first one fails. A deliberate failure test showed it did not — and located the real constraint outside the operator's control, which changed what was worth building.

CONTEXT

The system was a small distributed VPN service: a central server coordinating several servers across different regions, delivering connectivity to end users through consumer apps running on their own devices. Two servers in one region were provisioned as a redundant pair, on the standard assumption that if one failed, users would carry on through the other. Before committing to the more elaborate automatic-failover system that had been planned, one question needed a real answer: did that redundancy actually deliver failover, or only the appearance of it?

THE ASSUMPTION UNDER TEST

The working belief — common in this space and rarely verified — was that offering a user two servers guarantees automatic failover: if one becomes unreachable, the user's software moves them to the healthy one without interruption. The entire build plan rested on this being true. Rather than build on top of the assumption, I chose to test the assumption itself first.

METHOD

The test was a controlled failure injection. One server was deliberately disabled at the network level, reproducing a genuine outage while every other variable was held constant. The identical failure was then repeated across four different client applications a user might reasonably choose, isolating the user's software as the single variable under examination. Recovery was judged solely by whether a live connection actually came back — not by any on-screen status indicator, several of which proved unreliable in this configuration and reported healthy servers as unreachable.

FINDINGS

Three results held consistently across every trial:

- **No automatic recovery.** In none of the four applications did the connection move to the healthy server on its own. Users were simply left stranded on the dead one.

- **The deciding layer sat outside the operator's control.** Which server a user connects to is determined by the application on the user's own device — software the operator cannot modify. Server-side engineering, however sophisticated, could not guarantee the outcome, because the component making the decision was never the operator's to command.
- **Where switching did occur, it was uncontrolled.** One application offered a manual switch but chose the next server by list order alone, with no regard for its location or health — an unpredictable result rather than a managed one.

OUTCOME

The finding inverted the plan. Because the deciding layer lay beyond the operator's reach, the elaborate failover system was *not built*: it could not have delivered on its promise end to end, and would most likely have failed silently in production. The redundant pair was kept as a manual fallback, and the freed effort was redirected toward work that was genuinely within the operator's control. The whole conclusion cost a single session of controlled testing on infrastructure carrying no live customers — in place of the weeks that would have gone into building a system engineered to fail.

TRANSFERABLE PRINCIPLE

Two points generalise well beyond this project. First, verify a load-bearing assumption with the cheapest available test *before* building on top of it. Second, map which parts of a system you genuinely control — because a component that works in isolation is not the same as a system that works across every layer, including the ones you do not own.