

HOW THE ENGINE ACTUALLY WORKS

Eight chapters, each written twice: once for a reader who does not write software, once for an engineer reading with the source open. Every number on this page was read out of the shipping code, and the file it came from is printed next to it.

// 01 WHY MULTIPLAYER GPS IS HARD

What breaks when everyone on a map can see everyone else?

// PLAIN LANGUAGE

Showing one car on a map is easy. Showing a hundred cars to each other, live, is not – and the reason is arithmetic, not programming skill.

If every driver has to be told where every other driver is, the amount of talking does not grow with the number of drivers. It grows with the number of PAIRS of drivers. Twelve drivers make 132 pairs. Two hundred drivers make 39 800. Add one more driver and you have not added one conversation, you have added two hundred.

That curve is what kills most attempts. It looks fine while you and three friends are testing it, and it falls over the first evening a real crowd shows up – which is exactly the evening you needed it to work.

StreetWizard does not try to route those messages more cleverly. It sends fewer of them. Everyone publishes to one shared channel, so the message count grows in a straight line instead of a curve – and then four separate mechanisms, described in the next chapters, cut what remains.

// UNDER THE HOOD

The naive topology is a full mesh: N participants, $N(N-1)/2$ edges, and each edge carrying its own position stream. Whether those edges are peer connections or server fan-out, the message volume is $O(N^2)$ and the per-client inbound rate is $O(N)$.

A shared broadcast channel collapses the publish side to $O(N)$: each client emits once per tick regardless of crowd size. The inbound side is still $O(N)$ per client – that is inherent to 'everyone sees everyone' and no topology removes it. What the remaining mechanisms do is attack the constant in front of that N , and the N itself at the render layer.

This is the load-bearing correction to the previous version of this document, which claimed a WebRTC peer-to-peer mesh. A full P2P mesh

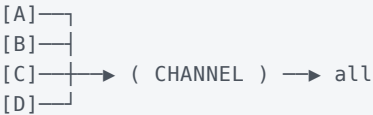
does not remove the quadratic term – it relocates it from the server to the phone, where each device would hold N-1 concurrent encrypted sessions while also running a 3D map and a GPS. That trade is examined in the last chapter.

NAIVE – every rider tells every rider



messages = $N \times (N-1) / 2$

STREETWIZARD – one shared channel



messages = N

riders	naive	shared
12	132	12
60	1 770	60
200	39 800	200

Publish volume. The inbound side still scales with the crowd – the next three chapters are about shrinking it.

Why does your avatar not weigh anything while you drive?

// PLAIN LANGUAGE

Two very different kinds of information travel between riders, and confusing them is the most common way to make a live map expensive.

Your name, your photo, your plate, the car you picked – that is who you are. It is comparatively large (a photo is a few thousand times bigger than a coordinate) and it almost never changes. Your position, heading and speed are the opposite: tiny, and different a moment later.

So they go down different roads. Identity is announced once when you join, and sits in a roster everyone already has. Position rides a separate, deliberately starved channel that carries nothing but the numbers that changed.

The practical effect: adding a photo to your profile costs the network one message, not one message per second forever.

// UNDER THE HOOD

The transport is Supabase Realtime, used in both of its modes at once on a single channel. Presence holds the roster: join/leave events plus a metadata payload per key, replayed to anyone who connects. Broadcast carries the position stream.

The heavy field is the avatar, stored as a data URL. It lives in the presence metadata only. A peer's photo is cached client-side in a `metaRef` map and re-merged into the drawable peer on every recompute, precisely so it never needs to ride a position message.

Presence doubles as a low-rate position fallback (refreshed no more often than once per 10 s), which covers the window between joining and receiving a peer's first broadcast – otherwise a rider who has not moved since you connected would be invisible rather than stationary.

Incoming broadcasts do not each trigger a re-render. They are coalesced into one recompute per 350 ms window: at N peers, a recompute per inbound message is O(N) work N times per tick, and the coalescing is what keeps that from becoming the bottleneck before the network does.

PRESENCE – sent once, on join

nickname	
avatar	~5 kB
vehicle / level	
plate, colour	

1x per session

BROADCAST – the hot path

lng, lat	
heading, speed	
tens of bytes	

2/s → 1 per 4s
adaptive, ch. 03



roster + positions	→ coalesced recompute / 350 ms
--------------------	--------------------------------

One channel, two modes. The avatar never touches the position stream.

```
// READ FROM SOURCE
```

```
TRANSPORT
```

```
Supabase Realtime – presence + broadcast, one channel
```

```
src/lib/fleet.ts
```

```
CHANNEL
```

```
wo-fleet-live src/lib/fleet.ts
```

```
AVATAR PATH
```

```
presence metadata only, cached in metaRef src/lib/fleet.ts
```

```
PRESENCE FALLBACK
```

```
position refresh ≤ 1 / 10 s src/lib/fleet.ts
```

```
RECOMPUTE COALESCE
```

```
350 ms window src/lib/fleet.ts
```

Why does bandwidth per rider go DOWN as the crowd grows?

// PLAIN LANGUAGE

How often you need to broadcast your position depends entirely on who is looking, and that is a fact about eyesight, not about networks.

Rolling three cars deep with your crew, everyone is close and every small correction is visible – a slight drift, a lane change. There, position updates need to be frequent or the convoy looks like a slideshow.

In the middle of a two-hundred-car meet, the same car is a dot among dots. Nobody can tell whether that dot moved half a second ago or two seconds ago. Sending four times more data to render something no eye can resolve is pure waste.

So the rate follows the crowd, in four steps, automatically. Small convoy: twice a second. Two hundred cars and up: once every four seconds. Nobody notices the difference, because at that density there is no difference to notice.

There is a second gate that matters just as much: you also have to have actually MOVED. A car parked at a meet with its engine off broadcasts nothing at all, no matter how long it sits there.

// UNDER THE HOOD

Emission is gated on two conditions simultaneously – elapsed time AND distance travelled since the last emitted fix. Both thresholds step with the observed fleet size, so the gate tightens exactly when the crowd it is protecting gets large.

The distance gate is the one that carries a stationary fleet. GPS on a parked phone jitters by several metres; a time-only throttle would happily broadcast that jitter forever. Requiring 5 m of real movement at the tightest step (40 m at the loosest) means a static meet costs the channel nothing, which is the state a car-scene app spends most of its evening in.

The ladder is deliberately coarse – four steps, not a continuous function of N. A smooth curve would make the emission rate a function of a number that itself flickers as riders join and leave,

so the rate would flicker with it and become impossible to reason about from a packet capture.

fleet size	min interval	min distance	the reasoning
≤ 12	500 ms	5 m	convoy – formation is visible
13 – 60	900 ms	10 m	street – still reads smooth
61 – 200	2 500 ms	25 m	crowd – prediction covers it
200 +	4 000 ms	40 m	swarm – no eye can resolve it

BOTH gates must open:

```

moved ≥ min distance  ┐
                        └─┬─> emit
elapsed ≥ min interval ┘

```

parked car, engine off, GPS jittering 3 m → emits nothing

Read straight out of fleet.ts. The distance gate is what makes a static meet free.

```

// READ FROM SOURCE

≤ 12 PEERS                                500 ms / 5 m src/lib/fleet.ts
-----
13 – 60                                    900 ms / 10 m src/lib/fleet.ts
-----
61 – 200                                  2 500 ms / 25 m src/lib/fleet.ts
-----
> 200                                     4 000 ms / 40 m src/lib/fleet.ts
-----
GATE                                     time AND distance, both required src/lib/fleet.ts

```

If a position arrives every 2.5 seconds, why does nothing stutter?

// PLAIN LANGUAGE

A position that arrives every two and a half seconds would, drawn naively, produce a car that jumps forward, freezes, jumps again. That is the classic multiplayer look, and it is unpleasant to watch for exactly as long as you watch it.

The fix does not involve sending more data. Each phone works it out locally. When a peer's position arrives, your device also knows which way that car is pointed and how fast it was going. From those it can carry the car forward on its own, continuously, while it waits for the next real fix.

So on your screen the car glides. Underneath, the network is delivering a fraction of the updates the motion implies.

There is a limit on how far ahead the guess is allowed to run – about a second and a half. Past that the guess stops, because a car that keeps sliding after its driver lost signal is worse than one that admits it stopped hearing.

And below walking pace the prediction is switched off entirely: at a red light, GPS noise looks exactly like slow movement, and a predicted car would drift out of its lane while standing still.

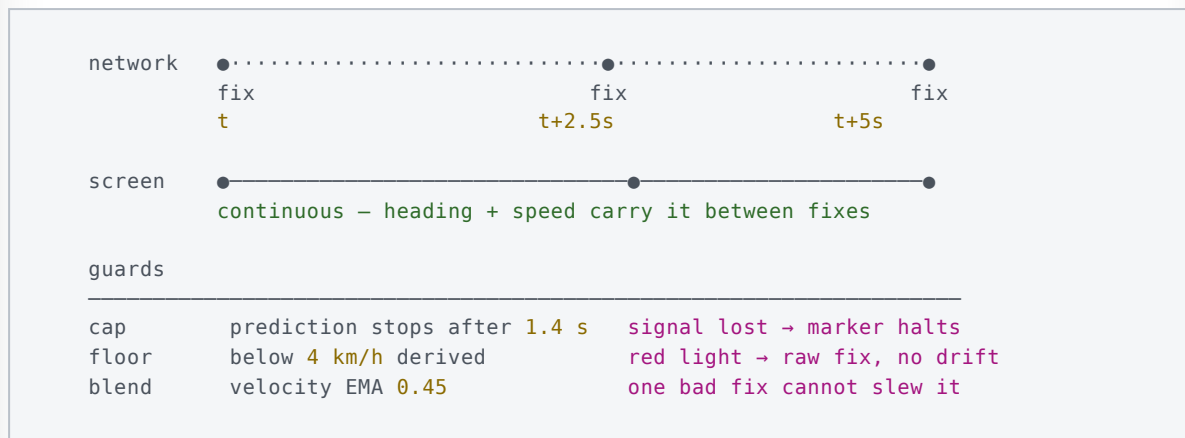
// UNDER THE HOOD

Each tracked entity – self and every peer – carries a small dead-reckoning state: a velocity vector in degrees per second, an EMA-smoothed speed estimate, and the age of the last real fix. Successive measurements derive the velocity; the render loop advances the target by that velocity times the frame delta.

Three guards make it safe rather than merely smooth. Prediction is capped at 1.4 s of extrapolation, so a dropped connection degrades into a stopped marker instead of a car driving off the map. Below 4 km/h of DERIVED speed the entity is treated as stationary and the raw fix is used unmodified. And the velocity estimate is blended at 0.45 rather than replaced, so a single noisy fix cannot slew the marker.

The derived speed is used rather than the reported one on purpose: a phone can announce a speed it is not achieving, and at a standstill the announced value is the least trustworthy number in the payload.

The same routine drives the self marker and the peer markers. That symmetry is the point – it means your own car and everyone else's degrade identically under bad signal, so nobody sees a map where only other people stutter.



Same routine for the self marker and every peer – they degrade identically.

```
// READ FROM SOURCE

DR_MAX_S          1.4 s prediction horizon src/components/map/camera.ts

DR_MIN_KMH
4 km/h floor (derived, not announced) src/components/map/camera.ts

DR_BLEND          0.45 velocity EMA src/components/map/camera.ts

APPLIED TO
self marker and every peer, same routine
src/components/map/NavExperience.tsx
```

What happens when a thousand riders are online?

// PLAIN LANGUAGE

Network cost and drawing cost are two different budgets, and solving one does nothing for the other. You can receive a thousand positions cheaply and still kill the phone trying to draw them.

Each car on the map is a real element on screen with a sprite, a name, a colour and a reputation badge. A thousand of those is a dead battery and a map at four frames a second – that is not a feature, it is a crash with extra steps.

So drawing is capped at the sixty nearest cars to you. Not the first sixty to connect, not a random sixty – the sixty closest, recomputed as you move.

That choice follows from what the map is for. The cars that matter are the ones you might actually see through the windshield. A rider four hundred kilometres away is a number in the online count, not a pin you need on your screen.

The online counter still shows the true total. The cap changes what is drawn, never what is reported – a thousand riders online reads as a thousand.

// UNDER THE HOOD

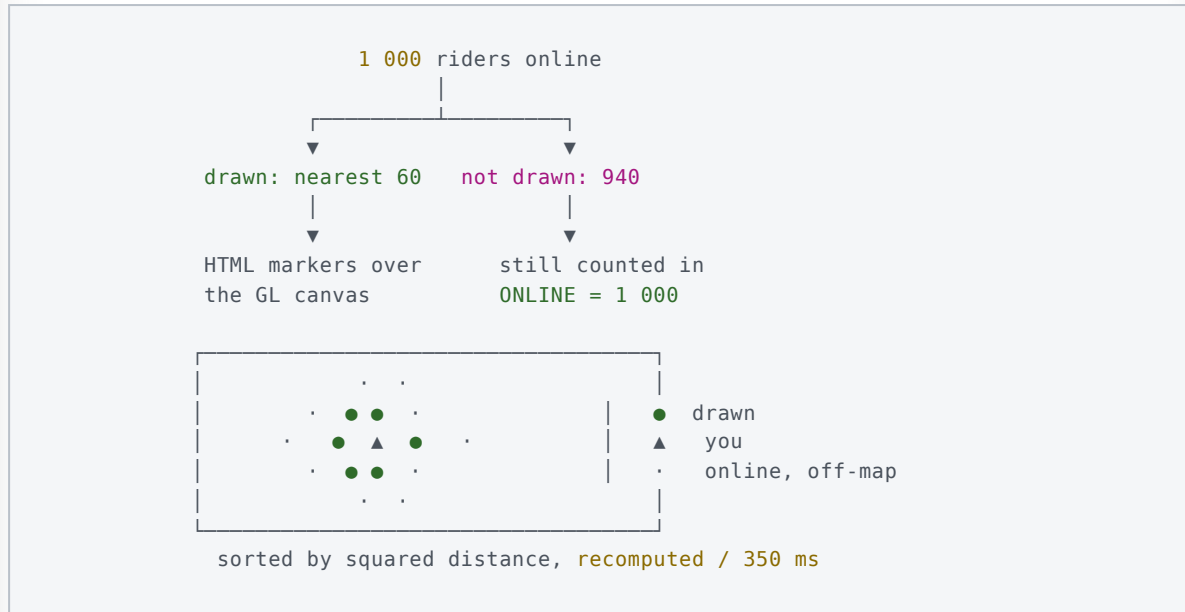
`MAX\_DRAWN = 60`. Above that the peer list is sorted by squared planar distance to the local fix and truncated. Squared distance, with a $\cos(\text{lat})$ correction on the longitude term, avoids a square root and a haversine per peer per recompute – at 60+ peers inside a 350 ms loop that arithmetic is worth removing.

The drawable set is the intersection of two sources: peers currently PRESENT (from presence) and peers for whom a position is known (from broadcast or the presence fallback). A peer missing from either side is not drawn – which is what prevents a stale marker from outliving the rider it belongs to.

The `online` count is computed from the presence set before truncation, so the cap is a rendering decision and never leaks into a displayed number. This matters more than it looks: a cap that

silently shrinks the reported population turns a performance guard into a lie about how busy the map is.

Every one of these entities is an HTML Marker positioned over the GL canvas, not a symbol layer. That is a deliberate trade – the cost is that DOM element count becomes the render budget, which is exactly why the cap exists; the benefit is real CSS, real accessibility and real tap targets on objects that carry buttons.



The cap changes what is drawn, never what is reported.

```
// READ FROM SOURCE

MAX_DRAWN                                60 nearest peers src/lib/fleet.ts
-----
SORT      squared planar distance, cos(lat) corrected src/lib/fleet.ts
-----
ONLINE COUNT true total, computed before truncation src/lib/fleet.ts
-----
MARKER TYPE
HTML Marker over the GL canvas src/components/map/NavExperience.tsx
```

Who is allowed to assert how far the community has driven?

// PLAIN LANGUAGE

The odometer is the one number on the map that belongs to everybody: every kilometre driven by every rider, ever, never reset.

A shared counter like that is only worth showing if it cannot be gamed, and the interesting engineering question is not how to add numbers quickly. It is who gets to assert one.

Each phone measures only its own distance, from its own GPS trail, filtering out the shake a parked phone produces so a car standing still never banks kilometres. It writes that to its own row and nothing else.

The community total is then computed on the server by adding up those rows. No phone ever sends the total, so no phone can set the total – the number on screen is a sum of individual claims, not a claim in itself.

Between two server reads the drums keep rolling: the client estimates forward from the live rate of riders currently on the road, then re-settles onto the real total when it arrives. It never rolls backward – a counter that visibly loses kilometres reads as broken even when it is being more honest.

// UNDER THE HOOD

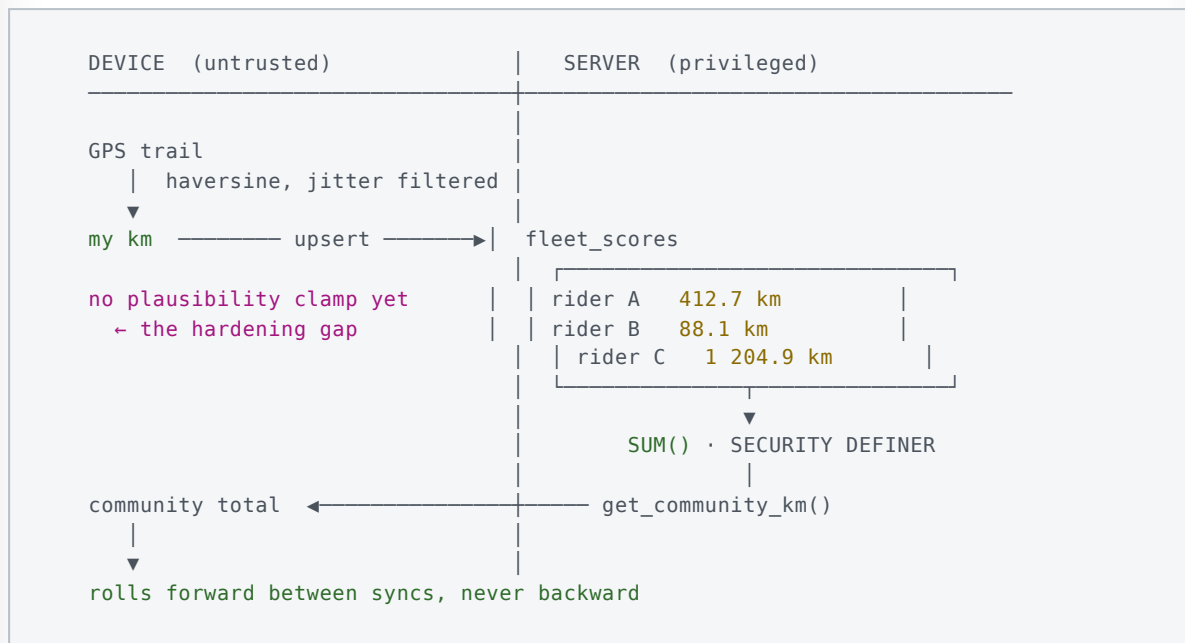
One row per rider in ``fleet_scores``, advanced only by its owner. There is no shared counter to contend over, so there is no write contention and nothing to resolve – the property the previous version of this document attributed to CRDTs is obtained here by not having a contended cell in the first place.

The total is ``get_community_km()``, a SECURITY DEFINER RPC summing those rows. Definer rights let the aggregate read across all rows without granting any client that read, so the sum is privileged while the underlying table stays locked down.

The client-side rolling is cosmetic and monotone by construction: it extrapolates from the aggregate live rate between syncs and clamps to never decrease, so a slow network shows a slightly

optimistic counter that converges rather than a counter that visibly rewinds.

HONEST LIMIT – this is where the current implementation is weaker than the design. A rider's own `km` is upserted directly from the client, and there is no server-side plausibility check on the delta. 'No client can submit the total' is true; 'no client can inflate the total' is not, because inflating your own row inflates the sum. Closing it is a database-side clamp – bound each write by elapsed time since the previous write times a maximum plausible speed – and it is the single highest-value hardening item on the engine.



The trust boundary is the vertical line. The gap marked in magenta is real and open.

```
// READ FROM SOURCE

STORAGE          fleet_scores – one row per rider src/lib/profile.ts

AGGREGATE
get_community_km() – SECURITY DEFINER SUM src/lib/community.ts

CLIENT ROLL      extrapolated between syncs, monotone src/lib/community.ts

OPEN GAP         no server-side delta clamp on km writes src/lib/profile.ts
```

How does a convoy hear the same track without a host?

// PLAIN LANGUAGE

The radio plays the same track to everyone tuned in, and a rider joining mid-song lands where the song already is – not at the beginning.

The naive way to build that is to elect one phone as the host and have it stream to the others. It works in a demo and fails on a road: the host drives into a tunnel and the music stops for everybody, so you need a re-election, and now every dropped connection is a renegotiation.

StreetWizard has no host. The server holds two facts – which track is playing, and the instant it started. A joining client reads those, works out how long the track has been running, and jumps its own player straight to that point.

Nothing is streamed between riders. The audio file comes from the same cache everyone else uses, and the only thing that had to be agreed on was a timestamp.

Riders will not be sample-accurate with each other – network jitter is real and pretending otherwise would be a lie. They land in the same place in the same song, which is what a convoy actually wants.

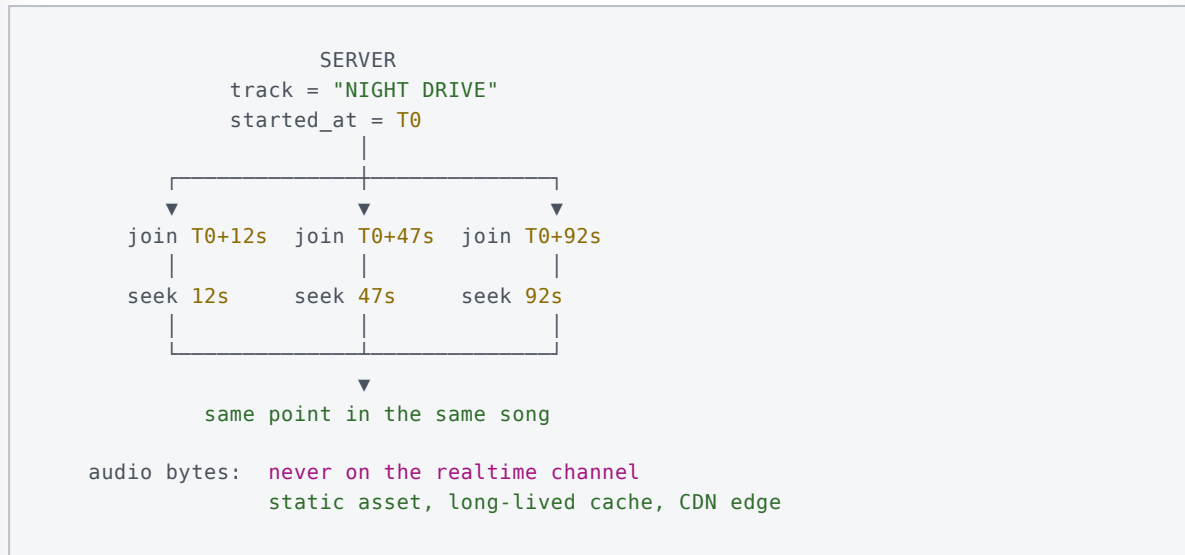
// UNDER THE HOOD

Authority is a server-held `(track, started\_at)` pair. A client computes `now - started\_at`, seeks its audio element to that offset and plays. No election, no host state, no renegotiation on churn – a dropping listener affects nothing but itself.

The realtime channel carries only that state, never audio bytes. The media itself is a static asset served with a long cache lifetime, so scaling listeners is a CDN problem rather than an application problem, and the marginal listener costs the application nothing at all.

The accuracy claim is deliberately modest. Convergence is bounded by clock skew between devices and by the network delay on reading the state – neither is corrected for, and the previous version of this document called this NTP synchronisation to the millisecond,

which it is not and was never built to be. What it is: good enough that a convoy hears one song, achieved without any of the machinery that a stricter claim would require.



No host to elect, so nothing to renegotiate when a rider drops.

The obvious upgrade is peer-to-peer. Is it?

// PLAIN LANGUAGE

Peer-to-peer sounds like the natural next step: skip the middleman, let the cars talk to each other. It is worth spelling out why that is the wrong first move here, because the intuition is reasonable and the arithmetic disagrees with it.

Start with the detail that makes this application different from the ones peer-to-peer was proven on. Those were two laptops on Wi-Fi, sitting still. This is a phone in a moving car, handing off from one cell tower to the next every few minutes – and each handoff can change its address and break the direct connection. An app built on peer connections would spend a drive re-establishing them instead of drawing the road.

What peer-to-peer saves is server bandwidth. But a position update is a handful of numbers – the entire live position stream is one of the cheapest things this application does. It would be trading real complexity for a saving that barely registers on a bill.

What it costs is worse. Direct connections between two phones on mobile networks often cannot be established at all, because carriers put their customers behind shared addresses. The standard fallback is a relay server – which means for those riders the data goes through a server anyway, and now you are paying for a relay you did not need before.

And the topology reintroduces exactly the problem this engine was built to avoid: sixty riders means each phone holding fifty-nine live connections, on a device already running a 3D map and a GPS. The quadratic cost does not disappear. It moves onto the phone, where there is far less headroom.

There IS a case for peer-to-peer here, and it is a different one: live VOICE between riders in a convoy. That is real-time media with strict latency requirements – the problem the technology was designed for. Positions are not.

// UNDER THE HOOD

MOBILITY IS THE DISQUALIFIER, before any of the arithmetic. A vehicle in motion hands off between cell towers continuously, and a handoff can change the device's apparent address – which invalidates the candidate pair a connection was established on and forces an ICE restart. The workload here is not 'peers on stable links'; it is 'every peer's link is expected to break repeatedly, by design, for the entire session'. Peer connections are the wrong primitive for that, independent of scale.

A full WebRTC mesh at N peers requires $N(N-1)/2$ peer connections, each with its own ICE agent, DTLS handshake and SCTP association. Per client that is $N-1$ concurrent encrypted sessions – the quadratic term relocated to the least capable node in the system, alongside a Mapbox GL render loop and continuous GPS. The current star topology is one connection per client regardless of crowd size, and the server does the fan-out.

Establishment is not free either. ICE gathering costs on the order of a second before a channel is usable, and the drawable peer set here churns continuously as riders enter and leave the nearest-sixty radius. A membership that turns over faster than connections establish spends its life renegotiating – and that is before the handoffs above add their own churn on top.

Mobile carriers deploy CGNAT widely, so a meaningful fraction of mobile-to-mobile pairs cannot hole-punch and fall back to TURN. TURN relays the media through infrastructure you operate – for those pairs the traffic costs MORE than the shared channel it replaced, plus the relay fleet, plus its operation.

Against that: the payload is on the order of tens of bytes at a rate already throttled to between 2/s and 0.25/s. The saving is close to noise; the added surface is a signalling service, a STUN/TURN deployment, per-peer connection lifecycle, and a second delivery path to keep correct.

VERDICT – not the first upgrade, and arguably not the fifth. The ranked list of what would actually harden this engine: (1) the odometer write clamp from chapter 06, since a claim about integrity is currently stronger than its enforcement; (2) moving Street Cred and Wiper Notes off localStorage onto the existing `fleet\_notes` table, since a reputation that lives on one phone is not a reputation; (3) instrumented measurements – bytes per rider per minute, wire-to-screen latency, frame rate at N peers – to replace

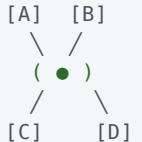
the design parameters on this site with numbers somebody took.
WebRTC belongs on this list only when the payload becomes voice.

FULL MESH – what it would take



per client	N-1 live sessions
at 60 peers	59 DTLS + SCTP
setup	~1s ICE, per pair
CGNAT	falls back to TURN
TURN cost	relays ALL media = MORE than today

STAR – what runs today



per client	1 session
at 60 peers	1
setup	once
relay	n/a
bandwidth	tens of bytes

AND THE ONE THAT DECIDES IT – the car is moving
tower handoff → address changes → ICE restart
every few minutes, every rider, all session long

the payload being fought over: lng, lat, heading, speed
2/s → 1 per 4s, already throttled

WHERE P2P WOULD EARN ITS KEEP

live VOICE between convoy riders – real-time media,
strict latency, exactly what the technology is for.
positions are not that problem.

The quadratic term does not vanish under P2P – it moves onto the phone.

// ON THE NUMBERS

These are design parameters – what the engine is configured to do – not benchmark results. They are verifiable against the source in a minute. End-to-end measurements (bytes per rider per minute, wire-to-screen latency, frame rate at N peers) would be stronger and are not on this page, because nobody has instrumented them yet.

// ENGINEERED BY



