can-lite

The Design Booklet — Layers, Class Models, Message Flows and Corner Cases of a Zero-Heap CAN Protocol Stack

Gabriel Francisco dos Santos

gabriel.fra.santos@gmail.com

August 23, 2026

Summary

Part I

Part I — Orientation

Chapter 1

About This Booklet

1. What this booklet is

can-lite is a client-server application protocol over CAN 2.0B, delivered as a C++20 library for microcontrollers with no heap, no blocking and no dynamically sized storage anywhere in the data path.

This booklet is the **design view** of that library. It walks the stack one layer at a time and shows, in diagrams, how the pieces are composed, what state each of them keeps, and what happens when things go wrong. It then reproduces the project's normative documents, so the whole design — narrative and specification — is one document.

2. What it deliberately does not contain

The project's documents each own a subject, and this booklet does not restate any of them:

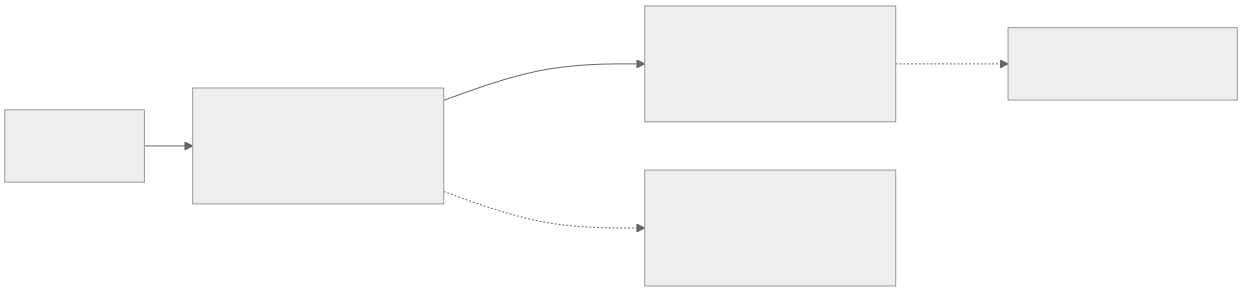
Subject	Owning document	Where in this booklet
Wire format: identifier layout, priorities, message catalogue, encoding, enumerations	Protocol Specification (Chapter 15)	Part IV
Firmware upgrade messages, states, error codes and flows	Firmware Upgrade Specification (Chapter 16)	Part IV
Architecture decisions, patterns, component relationships	Architecture and Design Decisions (Chapter 13)	Part IV
How to author an application category	Extending can-lite with Categories (Chapter 14)	Part IV
Formal requirements	`documents/requirements/*.yaml`	Appendix A, generated at build time

Parts I to III add what those documents do not have: the composition and ownership model, the per-layer state machines and pipelines, the catalogue of corner cases, the resource and timing

budgets, and the verification strategy. Where a fact belongs to one of the documents above, this booklet links to it rather than copying it.

There is also **no source code in this booklet**. It describes what each component is responsible for, what it keeps, and how it behaves; the repository is the authority on how that is written.

3. How the parts fit together



Reader	Suggested path
Integrating can-lite into a product	Part I, then the layer that carries your traffic, then Chapters 10 and 11
Writing an application category	Chapter 5, then the authoring guide in Part IV
Changing the protocol core	All of Parts II and III, then the specification in Part IV
Reviewing a change	Chapter 10 for the behaviour it must not break, Appendix A for the requirement it must satisfy

4. Conventions

Convention	Meaning
Component names	Written as they appear in the library, in the `services` namespace, which is dropped in prose
Chapter *n*	A cross-reference: a hyperlink on the web edition, a chapter number in the PDF
REQ-CAN-*nnn*	A requirement, listed in Appendix A
Design limit	A deliberate simplification, with its cost stated
Latent	Behaviour that is correct today but rests on an assumption worth knowing

Every diagram is generated from a diagram source held in the chapter it appears in, so the booklet moves when the design moves.

Part II

Part II — The Layers in Depth

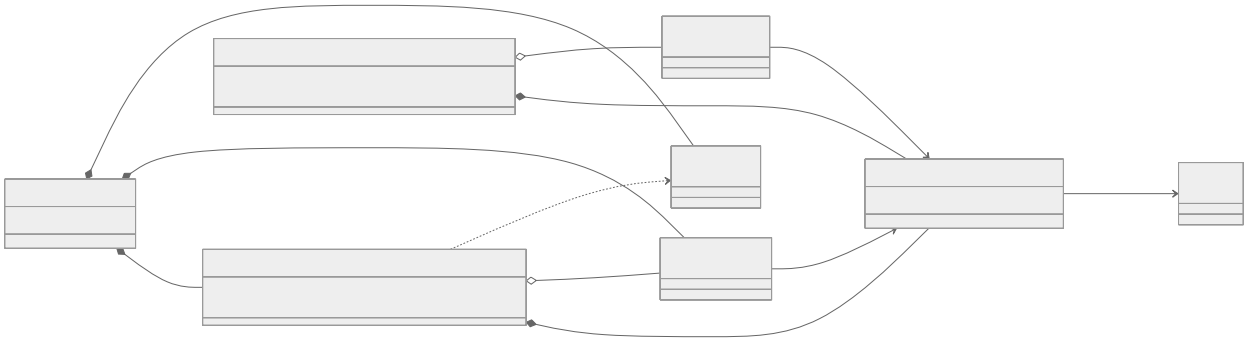
Chapter 2

The Layer Map

The layer stack itself, and the reasoning behind it, is in Architecture and Design Decisions (Chapter 13) §1–§3. This chapter adds the two views that document does not carry: **who owns whom at run time**, and **what links to what at build time**.

1. Composition and ownership

Nothing in can-lite is allocated, handed over or shared. Every object is a member of something the application declares, and every collaboration is a reference passed at construction.



Four consequences follow, and they are the whole of can-lite's lifetime model:

1. **Construction order is dependency order.** A category cannot be constructed before the frame transport it borrows, because it takes it by reference.
2. **Registration is separate from construction, and can fail.** Constructing a category always succeeds; registering it may not (Chapter 6).
3. **Observers bind for exactly their own lifetime** — they attach in their constructor and detach in their destructor.
4. **Destruction is the reverse of declaration.** A category must not outlive the protocol object whose transport it borrowed; declaring in dependency order gets this for free.

The single most consequential piece of that picture is that **one frame queue exists per node**, owned by the protocol object and lent to every category. It is what lets the protocol object observe

every outgoing frame, whoever produced it — the property the heartbeat rule depends on (Chapter 6).

2. Where the layering bends

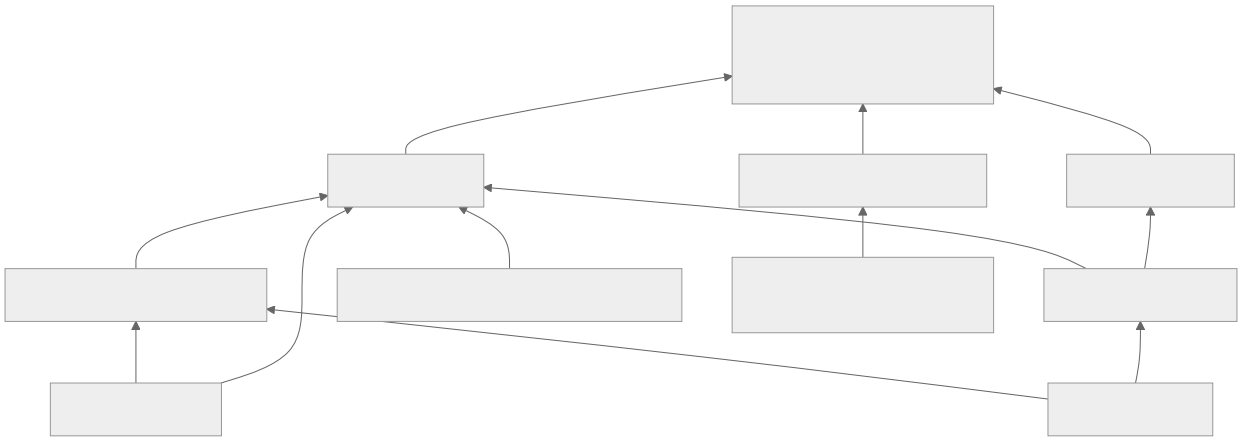
Two edges surprise readers who expect a strict cake, and both are deliberate:

Edge	Why it exists
Categories reach the core layer directly, not through the protocol layer	A category that borrowed the protocol object would be bound to one side of the bus; borrowing only the frame transport keeps it usable on both
The transport layer is attached at run time and sits beside, not under, the protocol layer	Segmentation is optional. A node that never sends payloads longer than a frame pays nothing for it, in code or in memory

The rule that keeps the rest honest: **no layer reaches two levels down for convenience.** A category that needs its node address asks the frame transport it already holds.

3. The build graph

The dependency rules are not conventions — they are edges in the CMake target graph.



Three rules govern it:

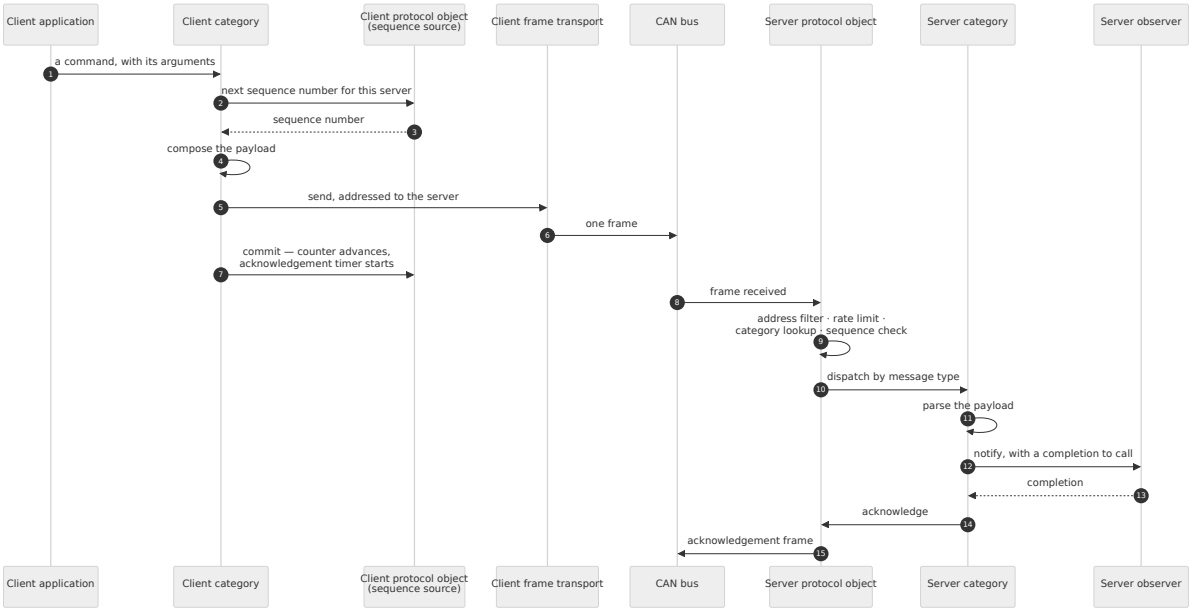
1. **A category links the core library and nothing else.** A category that links the server or the client library has quietly made itself unusable by the peer. This is the one build-level mistake worth watching for in review.
2. **The server and client libraries never depend on each other.** A node that is only a server does not carry the client's per-server state and timers.
3. **Dependencies are fetched only in a standalone build.** Consumed as a subdirectory, the infrastructure library must already be present.

One asymmetry is worth knowing: the server library does not link the transport library, even though it can attach segmentation, because the transport's interface is a header reached through

the core library's include path. A server that actually uses segmentation links the transport library itself — which is also what keeps that code out of a build that never uses it.

4. A frame's journey across the layers

The value of the layering is clearest when one command is followed from the client's application code to the server's observer. Nothing on this path allocates, and nothing blocks.



Each step belongs to exactly one layer, and each layer's contribution is visible in the frame: the category contributes the payload and its category identity, the protocol layer the sequence number and the acknowledgement, the core layer the identifier and the queueing, the HAL the bits on the wire.

5. Extension points

To add	Do this	Chapter
A functional group of messages	Author a category pair and register both halves	5, and the authoring guide in Part IV
Support for another bus	Implement the HAL interface, or the host adapter interface for tooling	3
Payloads longer than one frame	Attach the segmentation transport and give the message type a segmented handler	8
A reaction to protocol-level events	Implement the server or client observer interface	6, 7
Different timing	The configuration structures on the server, the client and the firmware upgrade category	11

Not in that table: the dispatcher, the identifier layout and the acknowledgement rules. Those are

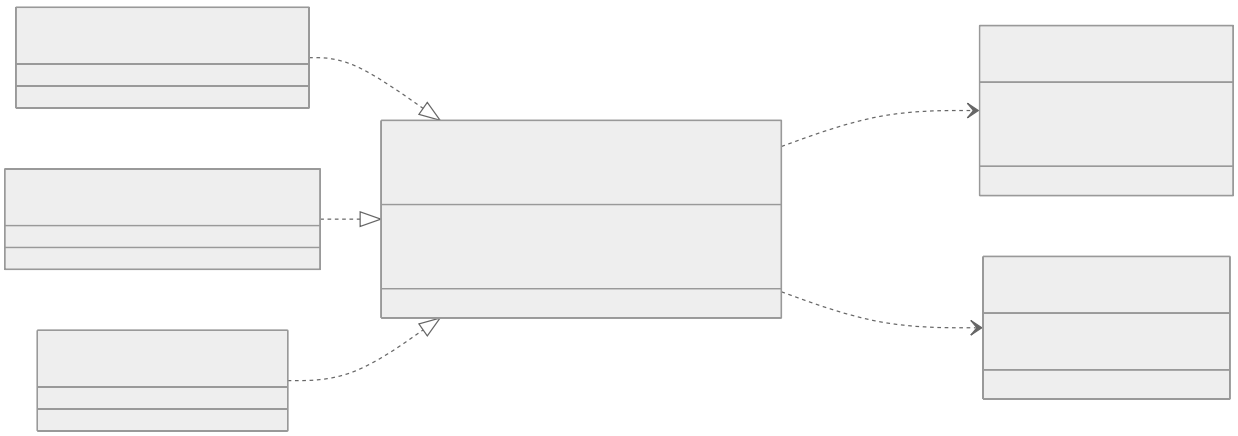
the protocol, and changing them is a specification change rather than an integration choice.

Chapter 3

HAL and Bus Drivers

The bottom of the stack is deliberately thin. can-lite asks the hardware for exactly two things — send a frame, deliver a received frame — and everything above is written against those two and nothing else.

1. What the library requires of a bus



Four properties of that contract shape every layer above it:

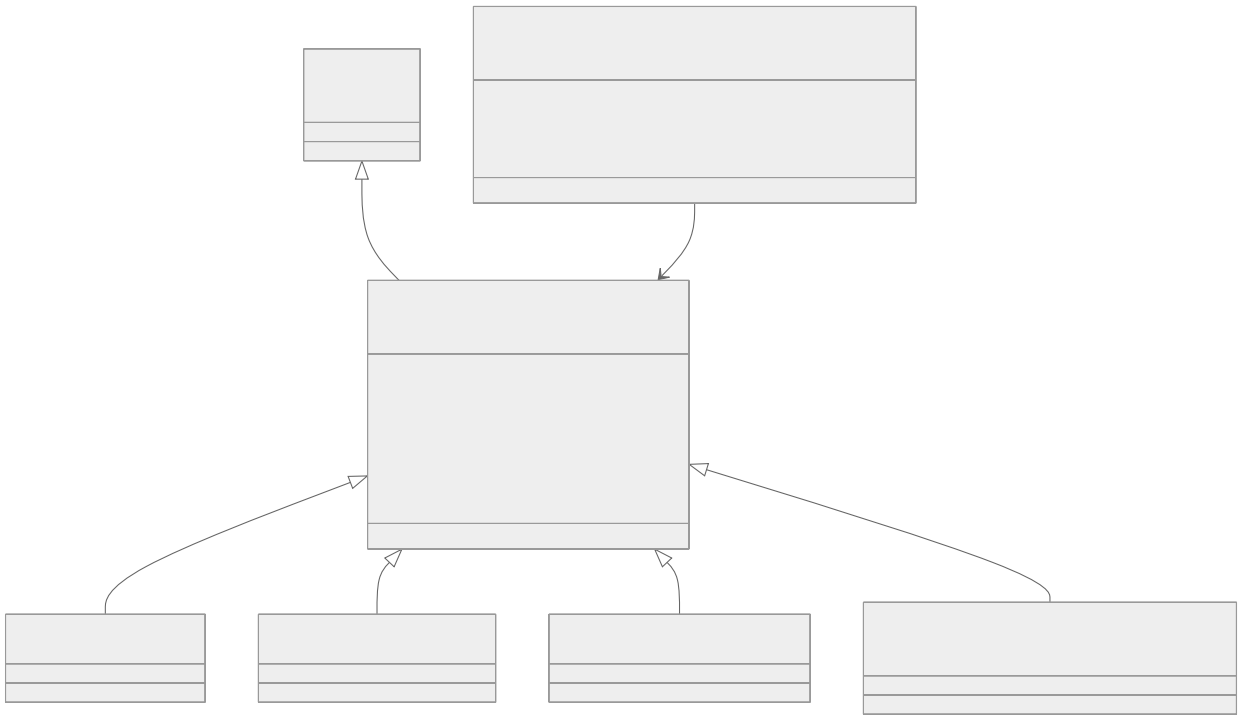
Property	Consequence
A payload is at most 8 bytes	Anything longer needs segmentation (Chapter 8), and payload composition is bounds-checked against the frame (Chapter 4)
Sending is asynchronous, completing through a callback	The core layer keeps its own queue and serialises sends (Chapter 4)
Exactly one receive callback exists per interface	The protocol object claims it at construction; a second consumer of the same bus is not supported

Property	Consequence
Identifier widths are distinguishable	can-lite uses extended identifiers exclusively and discards standard-identifier frames unseen, which is what lets it share a bus with other protocols (REQ-CAN-002)

The library neither provides nor requires a particular implementation. On a microcontroller the platform HAL supplies one; the library takes a reference to it when the protocol object is constructed.

2. The host adapter

Host-side tooling needs more than the bus interface offers: opening an interface, reporting connection state, enumerating what the machine has, and integrating with a poll loop. The adapter interface adds exactly that — and nothing the protocol layer would ever call.



Adapter	Availability
SocketCAN	Always, on Linux
PCAN-Basic	Windows, opt-in, requires the vendor SDK to be found
Kvaser CANlib	Windows, opt-in, requires the vendor SDK to be found
CANable over SLCAN	Windows, on by default

Each adapter is guarded both in the build and in its header, so including the wrong one for a platform yields an empty translation unit rather than a compile error.

The observer interface exists for tooling — a bus monitor attaches to it and receives every frame in both directions, driver error text, and connection transitions. It is an observation channel, not a data path: the protocol still receives frames through the bus interface.

Poll-loop integration. The adapter exposes a descriptor to select on and an entry point to call when it becomes readable, so a host application can service the bus from its own loop instead of dedicating a thread to it. Adapters also defer their send completions through the event dispatcher rather than calling them from inside the send. That detail matters more than it looks: it guarantees the core layer's queue is never re-entered from inside its own send (Chapter 10, §2).

3. The virtual bus used by the integration tests

The integration tests replace hardware with a pair of connected virtual buses: one instance's send lands in the other's receive callback, which models the two-node topology the tests use. A second entry point pushes a frame straight into the receive callback, bypassing the peer — that is how a test produces something no correct peer would ever send: a malformed acknowledgement, a wrong sequence number, an unknown category, a standard-identifier frame. Disconnecting models bus loss, which is how the liveness timeouts of Chapters 6 and 7 are exercised.

What the virtual bus does **not** model is the boundary of what the integration tests can prove:

Not modelled	Consequence for testing
Arbitration and priority	Frame order is send order, so priority effects are reasoned about (Chapter 11), not observed
Bus errors, error frames, bus-off	Sending always succeeds; failure paths are exercised through mocks instead
Transmission delay and bit timing	Bus-load budgeting is arithmetic, not measurement
More than two endpoints	Multi-server behaviour is covered at the unit level

4. What can-lite assumes of a driver

A driver that satisfies the interface but breaks any of these produces protocol-level misbehaviour rather than an obvious failure, so they are worth stating:

1. **Each send completes exactly once, and not from inside the send call itself.** The core layer dequeues the next frame from within that completion; a synchronous completion turns queue drain into recursion.
2. **Whole frames are delivered**, with the identifier already separated from the payload.
3. **The receive callback runs in the application's own context.** A driver that calls it from an interrupt must marshal through the event dispatcher first.
4. **Frames arrive in the order they were received.** Sequence validation rejects reordered commands, so a reordering driver produces spurious sequence errors.
5. **Acceptance filtering, if configured, must admit the broadcast address.** The client's heartbeat is broadcast; a filter that matches only the node's own address will make the server report its client offline on an idle bus.

Point 5 is the one that bites during MCU integration, and it is catalogued in Chapter 10, §6.

Chapter 4

The Core Layer: Frame Flow

The core layer owns four concerns: what an identifier means, how outbound frames are queued, how payload bytes are composed and consumed, and how numbers are represented on the wire. The identifier layout, the priority values and the number encoding are specified in the Protocol Specification (Chapter 15) §5, §6 and §9 and are not repeated here. What follows is the part that is not in any specification: how the layer behaves.

1. The identifier is the routing table

Everything a receiver needs in order to decide what a frame is — its urgency, its functional group, its specific message, and who it is for — lives in the identifier. **No routing decision requires reading a payload.**

Three properties follow, and they are why the layout is worth its complexity:

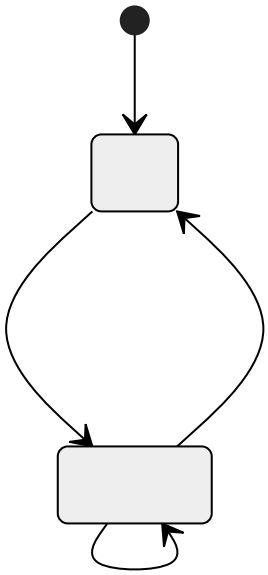
- **Hardware can filter.** An acceptance filter on the address field drops another node's traffic before the CPU sees it.
- **Software routes on integers.** The dispatch pipeline compares extracted fields; it never parses.
- **Priority is arbitration.** Because urgency occupies the most significant bits, the specification's priority ordering *is* the bus's ordering: emergency traffic beats commands, which beat responses, which beat telemetry, which beat heartbeats, regardless of which node sends them.

The layout is known in exactly one place. Composition and extraction helpers are compile-time evaluable, so identifiers known at build time — an acceptance filter table, for example — cost nothing at run time. Category code never manipulates the bits itself; that is what makes the layout changeable in one place.

The command/response split in the message-type field is the other structural decision: it lets a server discard responses without a lookup, which is what stops two servers on one bus from reacting to each other's answers.

2. One outbound queue per node

The bus interface accepts one frame at a time. Without a queue, a category wanting to send two frames back to back would have to know whether the controller was busy — so the core layer owns that knowledge, once, for the whole node.



Four behaviours of that small machine are load-bearing:

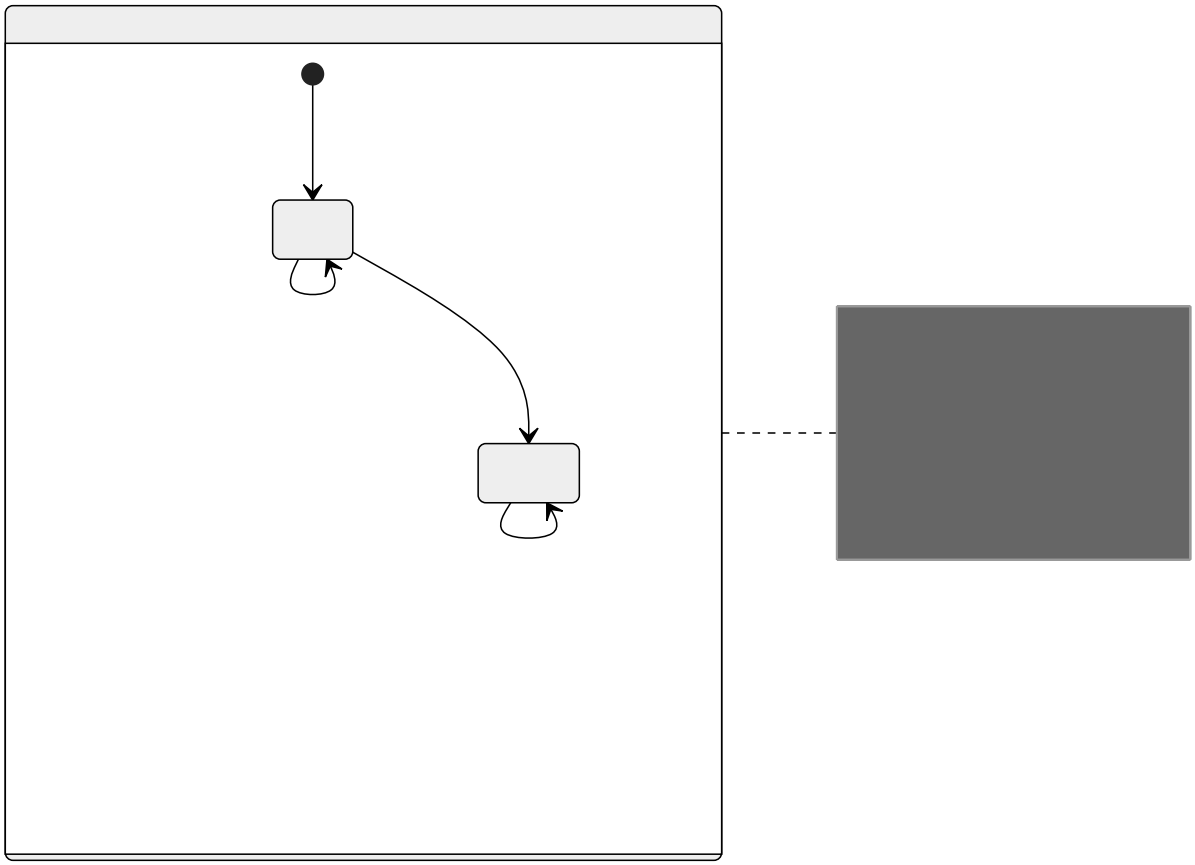
Behaviour	Why it matters
The queue advances **before** the finished frame's completion runs	A completion that itself sends finds the transport in a consistent state, so chained sends are safe
The queue holds a fixed eight frames; the ninth is **refused, not buffered elsewhere**	Refusal is visible to the caller, who decides. A category's send reports failure; a client command reports failure **without consuming a sequence number**; an acknowledgement is simply lost
Every accepted frame — queued or immediate — raises a send notification	This is the hook the protocol layer uses to defer its heartbeat, and it is why a refused frame does not postpone one
The notification has exactly one owner, enforced at run time	The protocol object claims it at construction; a second claimant is a programming error, and fails loudly

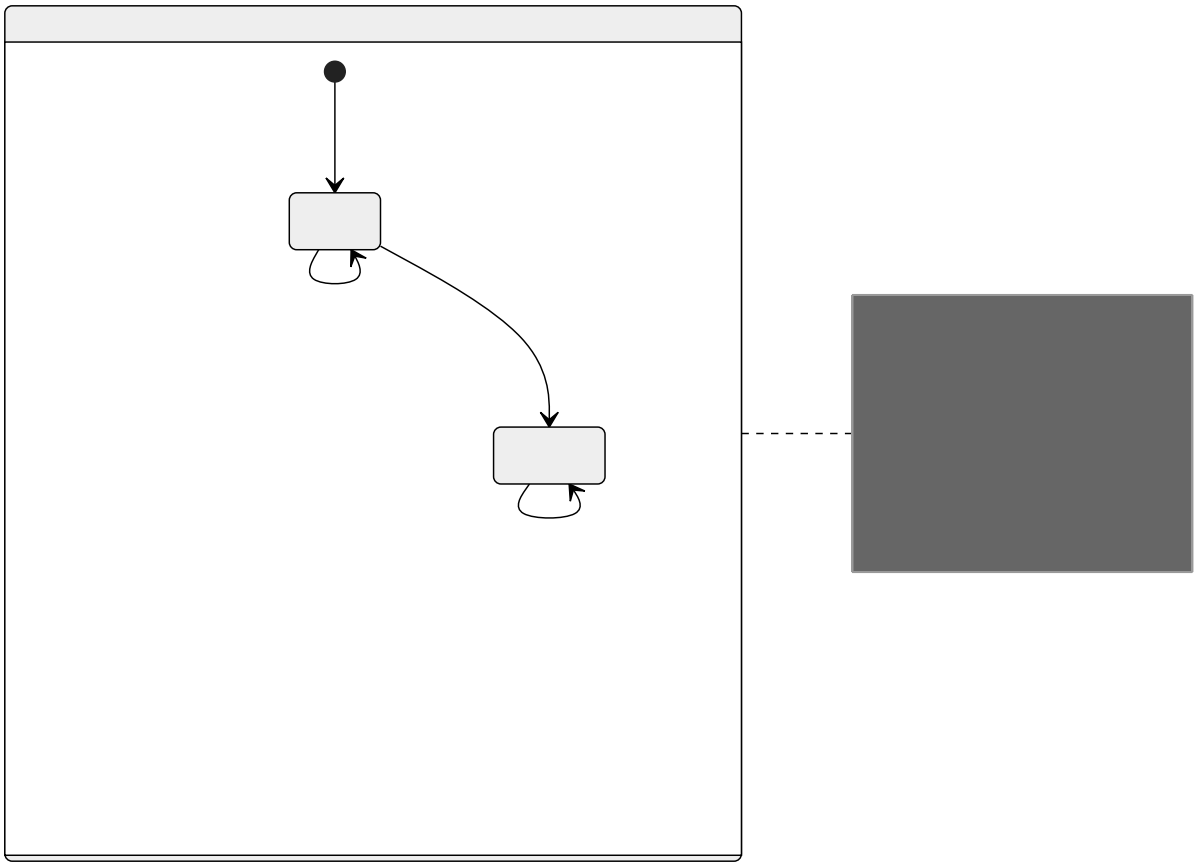
The address stamped into an outbound frame differs by role, and the asymmetry is worth fixing in mind: a **server** stamps its own address, because a response must say who answered; a **client** stamps the target's address, because a command must say who it is for. The same identifier field therefore reads as *source* on a response and *destination* on a command. The direction is never ambiguous, because the message type already says which it is.

A node whose address is configured late — from a switch, from stored settings, from a bootloader parameter — can change it after construction. Frames already queued keep the identifier they were built with.

3. Payload composition: bounded, big-endian, sticky

Two small components handle payloads, and between them they make it impossible to write past the end of a frame or read past the end of one.





The **sticky** part is the design decision. Rather than making every field access return a status the caller must check, the first out-of-range access poisons the whole operation and later ones become no-ops. A caller therefore writes the natural sequence of fields and checks once at the end, and the failure mode is a payload that is never sent or a command that is never acted on — not a half-formed frame or a plausible-looking wrong value.

Two guards complete the picture. The send helpers refuse an invalid composition, so a category cannot transmit a truncated frame even by ignoring the check. And a reader borrows the frame it reads rather than copying it, with binding to a temporary made impossible at compile time, so a reader can never outlive its data.

The consequence a category author must remember is the one thing this layer cannot check: a handler that reads its fields and **forgets to check validity** will act on zeros. That is catalogued in Chapter 10, §9.

4. Numbers on the wire

Multi-byte fields are big-endian, and floating-point values cross the bus as scaled integers, so that a target without a floating-point unit and a host with one produce byte-identical frames. The scale is chosen per message and is part of that message's specification.

Two properties of the conversion are worth carrying into a design review, because they decide what a peer sees when a value is out of range:

- **Out-of-range values saturate; they do not wrap.** A control loop that receives a saturated set-point is behaving as commanded-but-clipped, never as commanded-with-the-sign-flipped. The 32-bit conversion needs particular care at its upper bound, because the bound is not exactly representable in a 32-bit float — the comparison is deliberately inclusive so that the conversion cannot overflow past it.
- **Not-a-number maps to zero** on the 32-bit path. A scaled integer has no encoding for it, and zero is the least surprising set-point.

Chapter 5

Category Dispatch and Sequence Policy

A category is the unit of extension: a pair of classes, one per side of the bus, owning a functional group of messages. The type hierarchy, the observer pattern and the bidirectional pairing are described in Architecture and Design Decisions (Chapter 13) §3, §5 and §7, and how to author a pair is the subject of Extending can-lite with Categories (Chapter 14). This chapter covers the two things neither document treats in depth: **what dispatch does with a frame**, and **how the two halves of a pair must agree on sequence policy**.

1. Message types are members, not classes

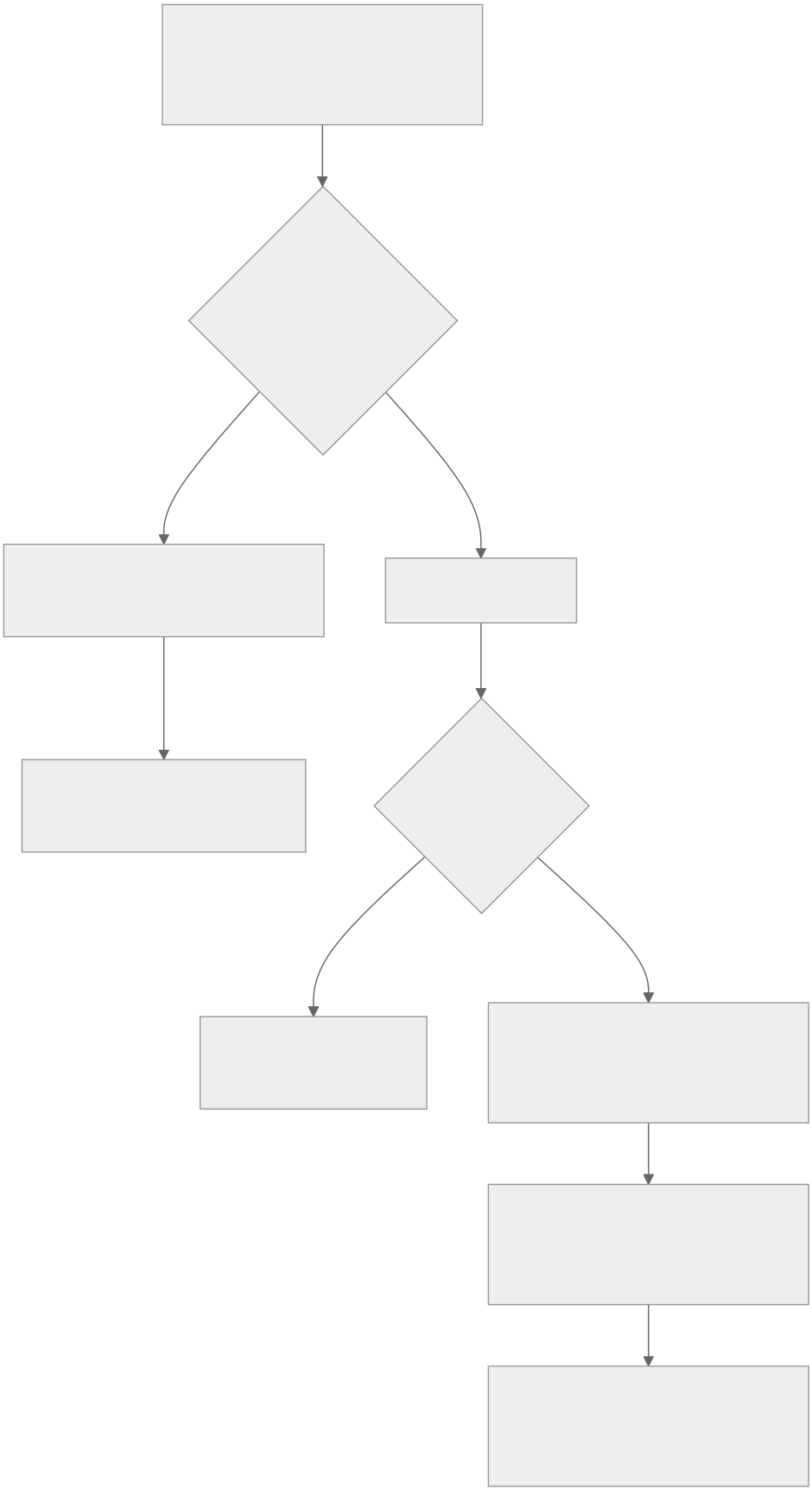
Inside a category, a message type binds a message identifier to one of the category's own member functions. Three values — the identifier, the owning category and the member to call — are all a binding holds.

The alternative, one small class per message, was what the library used first. Replacing it removed a type and about a dozen lines per message, kept handler bodies as ordinary private members of the category, and — because the bindings are members of the category object — needs no allocation and no back-pointer bookkeeping. A handler that needs state keeps it in the category, which is the only object with a lifetime long enough to hold it.

Do not reintroduce a class per message. It is the one deviation from the authoring guide that still compiles and works, and it costs code size in a library whose whole premise is bounded cost.

2. What dispatch does

Lookup inside a category is a linear scan over its registered message types — a handful of comparisons, on a bus that delivers at most a few thousand frames a second. What matters is not the scan but the **outcome**, because the outcome is the contract between the category layer and the protocol layer.



The distinction that catches people out is between the two failure paths. **Not handled** means the category has no binding for that message type, and the protocol layer turns it into an acknowledgement on the category's behalf. A handler that runs and fails reports *itself* — through an acknowledgement status or, when the fixed statuses cannot express the failure, through the category error message type that every category reserves for the purpose.

3. The segmented path

A message type may additionally opt in to handling a reassembled multi-frame payload. One that does not opt in inherits a default that **declines**.

That default is a deliberate choice with a visible consequence: sending a multi-frame payload to a message type that only understands single frames yields an "unknown command" acknowledgement rather than a truncated read of the first eight bytes. Silence would have been worse; a partial read would have been much worse.

Everything else on the segmented path is identical — the same address filter, rate limit, sequence check, category lookup and acknowledgement statuses — which is what makes segmentation transparent to a category (Chapter 8).

4. Sending: what each side gets

Neither half of a category composes an identifier. The base classes fill in the category identity and the priority, so a category never touches the frame layer.

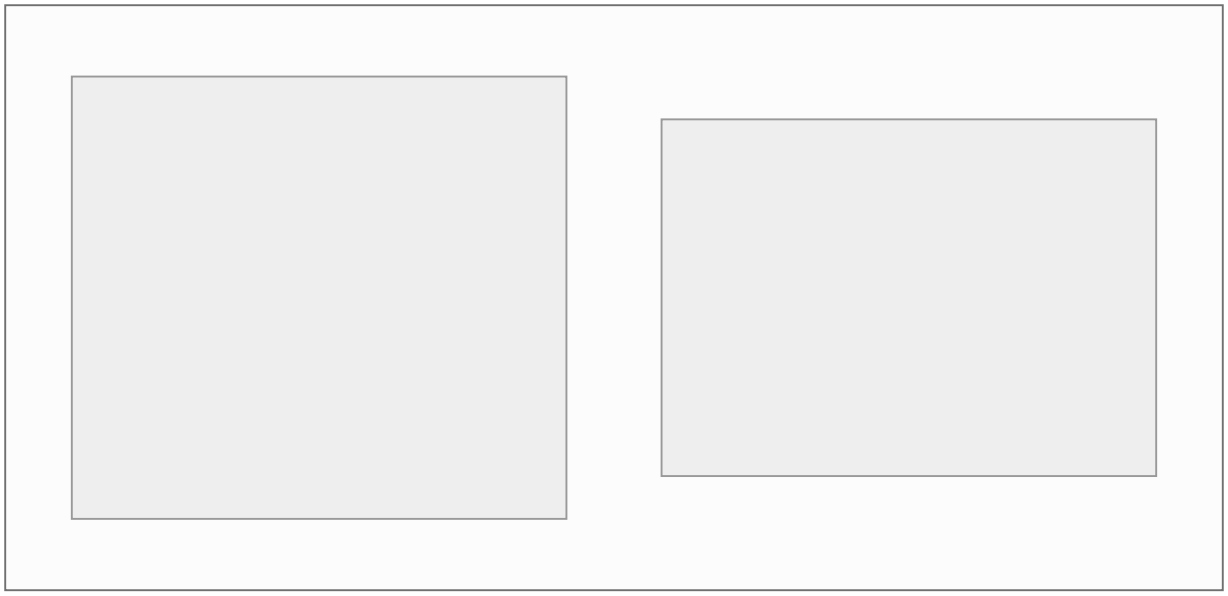
Side	Can send	Cannot send
Server	responses, telemetry, category errors, and acknowledgements routed through the protocol object	commands
Client	commands, addressed to a specific server	anything unsolicited

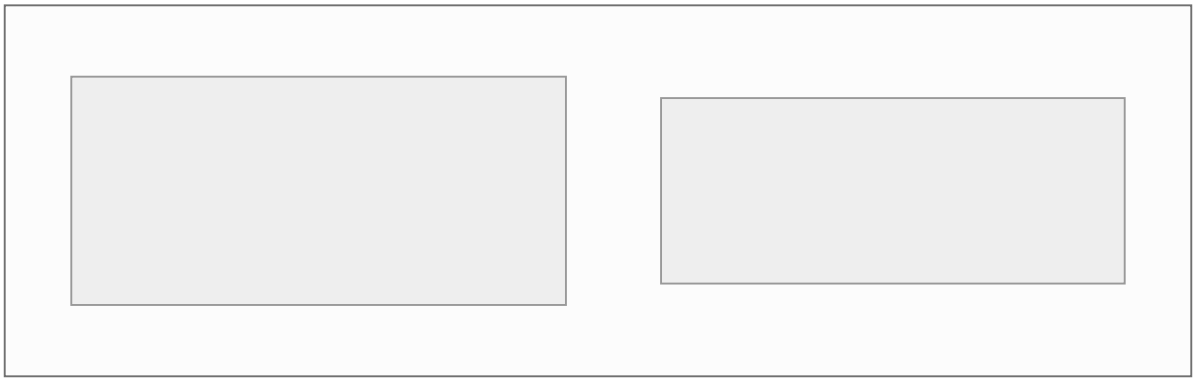
Acknowledgements are the interesting one. An acknowledgement is a *system* category message, and a category is not allowed to speak for another category — so a server category asks the protocol object, which it is given at registration, to acknowledge on its behalf. A category that tries to acknowledge before it has been registered fails loudly; that is only reachable when a test drives a category directly, and Chapter 12 notes how the tests arrange it.

5. The sequence-policy contract

Whether commands in a category carry a sequence number is a **per-category decision**, declared by the server half. The reasoning for each built-in category's choice is in Chapter 9; the mechanism is in Architecture and Design Decisions (Chapter 13) §9.

What matters here is that the two halves must agree, and nothing enforces it:





Both disagreements compile, link and run. Neither produces a diagnostic. They are the most confusing configuration errors in the library, which is why they head the category-level entries in Chapter 10.

Two further consequences of a validated category are easy to forget:

- The sequence number occupies the **first payload byte**, so a validated command carries at most **seven** bytes of category payload.
- The protocol layer inspects that byte but does **not** remove it. A validated server handler must skip it before reading its own fields — and a handler that forgets produces plausible, wrong values rather than an error.

6. Why the client half needs an interface, not the client

A client category is handed its sequence numbers through a small interface implemented by the client protocol object, rather than being handed the protocol object itself. That indirection is the reason a category library can link the core library alone: taking the client would drag the client library — its per-server state, its timers — into every category, and with it the coupling to one side

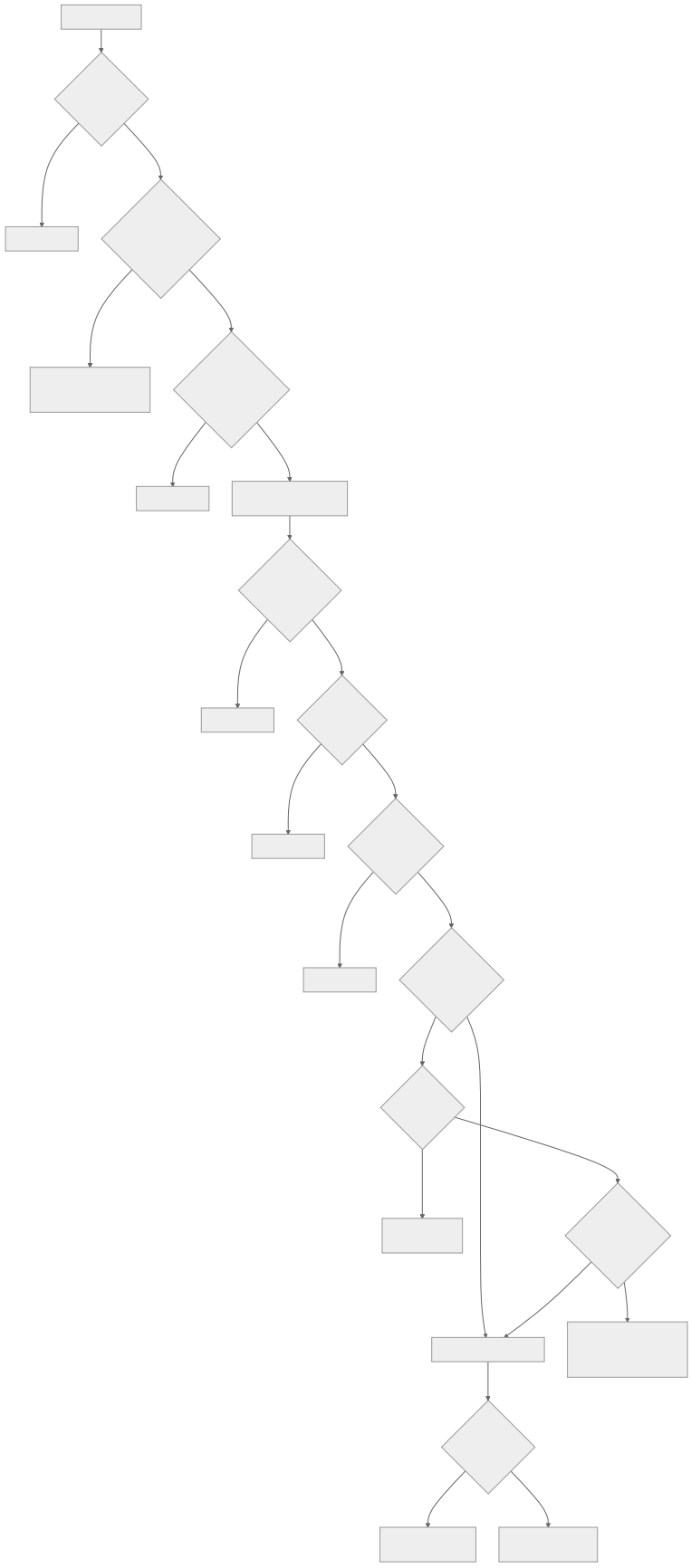
of the bus that the whole category design exists to avoid.

Chapter 6

The Server: Receive Pipeline

The server is the node that answers. Everything it does is a reaction — to a received frame, or to a timer it allowed to expire. Its responsibilities, its configuration defaults and the reasoning behind sequence validation, rate limiting and liveness are in Architecture and Design Decisions (Chapter 13) §6 and §9, and the message catalogue is in the Protocol Specification (Chapter 15) §8. This chapter adds the view neither has: **the pipeline every received frame runs, and why each gate answers the way it does**.

1. The pipeline



2. Why some rejections are silent and others are answered

This is the design decision that a reader of the code most often questions, so it is worth stating as a rule: **a frame is answered only when it was unambiguously meant for this server and this category.** If it was, silence would be a bug. If it was not, an answer would be noise.

Gate	Answer	Reasoning
Standard identifier	Silent	The frame belongs to another protocol sharing the bus
Another node's address	Silent	Answering would produce noise proportional to the number of servers on the bus
Over the rate limit	Silent	An acknowledgement is itself a frame; answering a flood would double it
A response, not a command	Silent	Servers do not consume responses; answering would create a loop between servers
Unregistered category	Silent	The frame may be legitimate traffic for another server that does implement that category
Empty payload, validated category	Answered	Addressed here, category exists — the client deserves to know
Sequence mismatch	Answered, with the expected number	The client needs that number to resynchronise without a round trip (Chapter 7)
Unknown message type in a known category	Answered	The client is talking to a category that does not implement that command

Two ordering choices in the pipeline are equally deliberate:

- **Liveness is marked before the rate limit is applied.** A client that floods the bus is still, evidently, alive; marking liveness afterwards would make a flooding client appear to vanish.
- **The rate limit is applied before the category is looked up.** What is being limited is the cost of *processing* frames, and that cost is paid before the category is known.

One consequence of the rate limiter is worth designing around rather than discovering: it counts within a fixed window that resets on a timer, so a burst straddling a reset can deliver up to twice the configured number in quick succession. Chapter 10, §7.2 walks through it, and Chapter 11 turns it into a sizing rule.

3. Registration, and why it can fail

A category is admitted only if the server has a free slot and no other category already claims its identity — and the answer is a value the application must check, not an assertion.

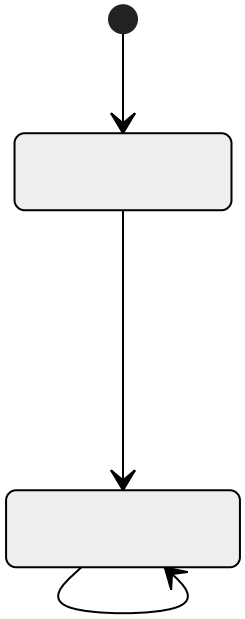
That choice is deliberate: the set of categories a product registers can be configuration-dependent, and a node that finds at start-up that it cannot register everything may prefer to run degraded rather than not at all.

The cost of ignoring the answer is that an unregistered category is **indistinguishable on the wire from a category that does not exist**: its commands are dropped silently, by the gate

above, and its client waits for responses that never come. This is entry 3.11 in Chapter 10, and it is the most common integration mistake.

The server's own system category occupies one slot from construction, which is why an application has one fewer than the total available.

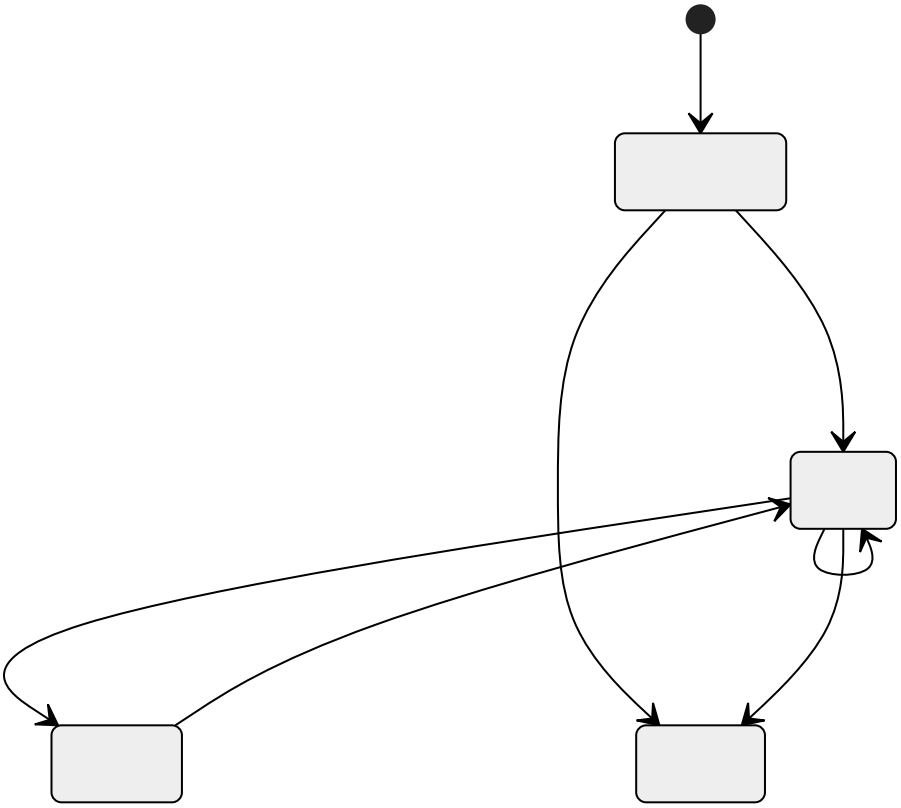
4. Sequence validation as a state machine



Four properties define the behaviour, and each is a deliberate simplification:

- **The first command sets the baseline.** A client that restarts from zero does not need the server to restart too — but only until the server has seen its first command.
- **Rejection is idempotent.** A rejected command does not advance the counter, so ten misordered commands produce ten identical answers rather than drift.
- **The counter wraps naturally** at the end of its range.
- **One counter is shared by every validated category.** Interleaving commands across two categories is therefore fine — they share one ordering — but two *clients* commanding one server interleave their counters and reject each other's traffic. That is the one-client-per-server rule (REQ-CAN-006.1) showing through, and it is catalogued in Chapter 10, §4.6.

5. Liveness, in both directions



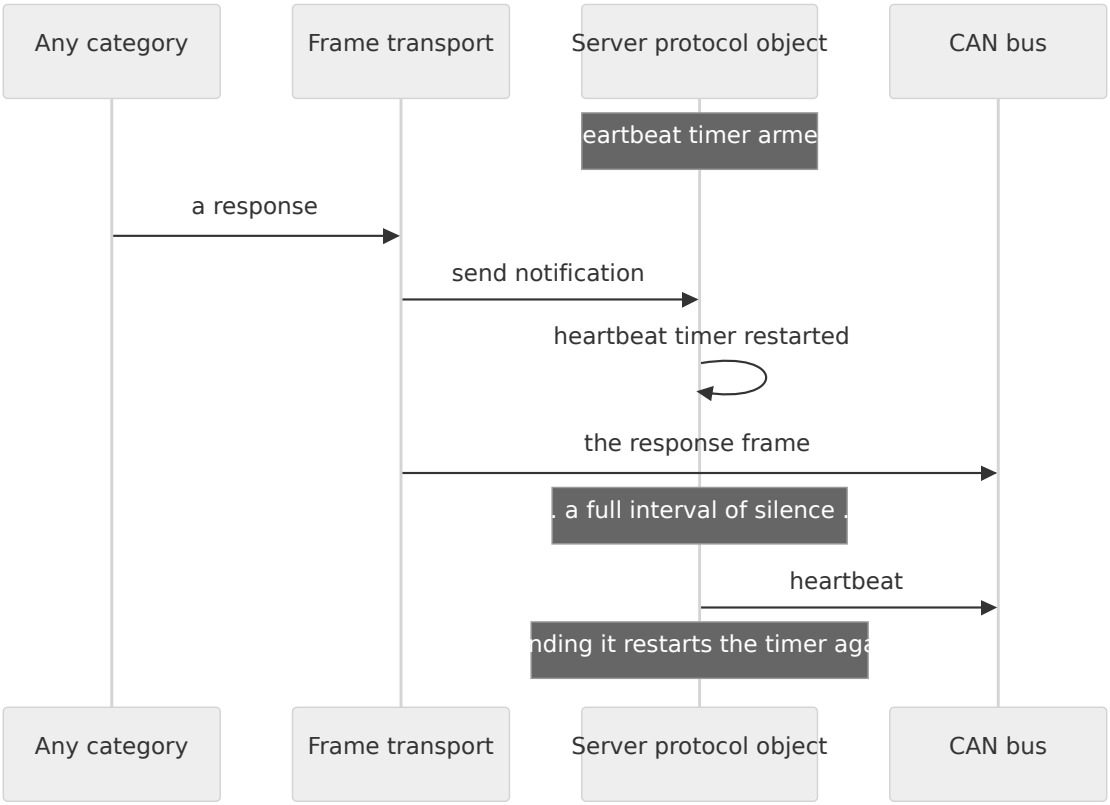
Two distinct signals are in play, and conflating them is the usual confusion:

- **Any correctly addressed frame** proves the client is present and restarts the timer. This matters because the client's heartbeat is itself deferred by its own outgoing traffic: a client that commands continuously may not send a dedicated heartbeat for a long time, and must not be declared offline for being busy.
- **Only a heartbeat** raises the "online" notification, because that is the specific, meaningful announcement.

The offline notification is raised at most once per outage rather than once per timer expiry.

6. The heartbeat is a silence guard

The heartbeat timer is restarted after **every** outgoing frame, whatever produced it — which is possible only because the whole node shares one frame queue (Chapter 2). A heartbeat is therefore emitted only after a full interval of complete silence from this node.



On a busy bus this reduces heartbeat traffic to nothing, because responses and telemetry already prove the node is alive. On an idle bus it produces exactly one heartbeat per interval. The same mechanism runs on the client, with the difference described in Chapter 7.

7. Attaching segmentation

Attaching the transport changes the pipeline in exactly one place — the second gate — and nothing else about the server. A reassembled payload runs the same gates in the same order, ending in the segmented dispatch path rather than the single-frame one, which is what makes segmentation invisible to a category.

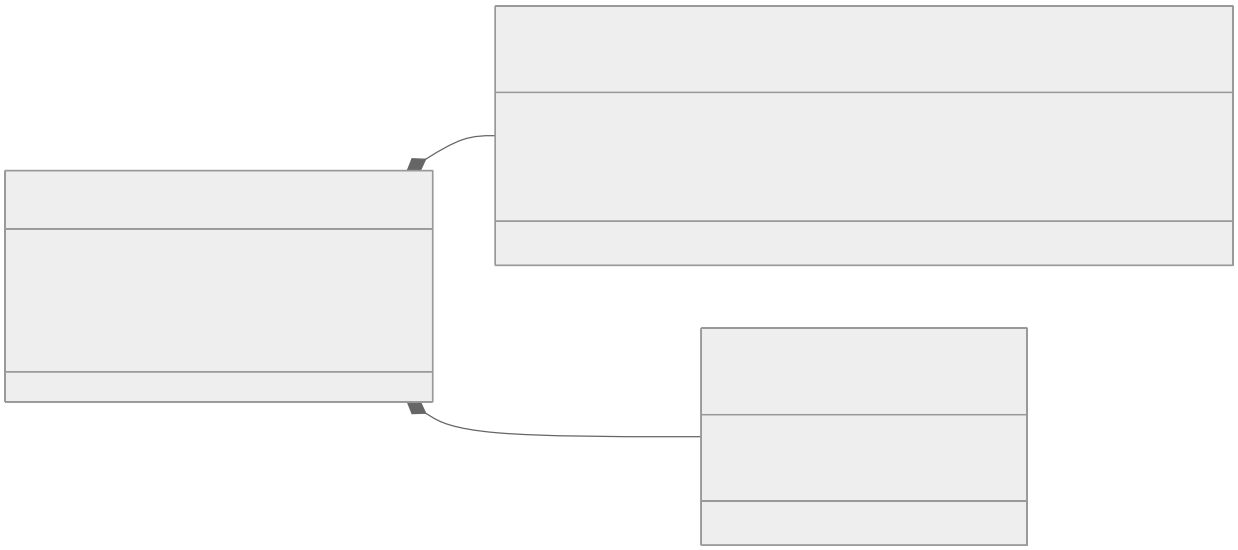
The server also arranges for a transfer that fails part-way to release its channel, so a failed transfer frees its slot instead of holding it forever. Chapter 8 covers when transfers fail and what the peer observes.

Chapter 7

The Client: Per-Server State

The client is the node that asks. It has no address of its own, it addresses servers by node identity, and it keeps a small amount of state for each server it deals with. What that state is *for* — sequence tracking, resynchronisation, liveness, acknowledgement timeout — is described in Architecture and Design Decisions (Chapter 13) §9.1 to §9.4. This chapter shows how it **behaves**: three state machines and the pipeline that feeds them.

1. Two independent tables



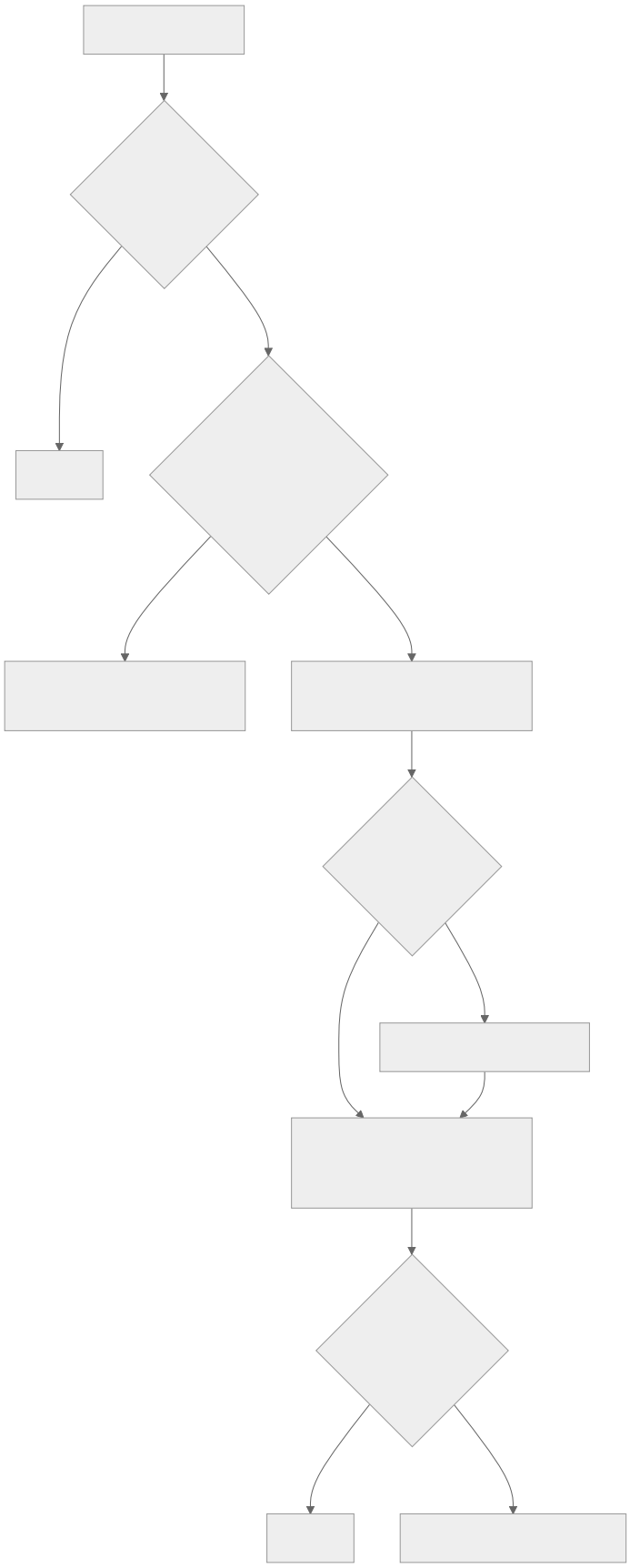
The two tables are **separate and independently populated**, and that is a design decision rather than an accident of implementation. Sequence state is created by *sending*; liveness state is created by *receiving*. A server that is commanded but never answers occupies a sequence slot and no liveness slot; a server that only emits telemetry occupies a liveness slot and no sequence slot. Merging them would mean allocating sequence state for a server that has only been heard from, or tracking liveness for one that has never spoken.

The client's own address is never used as a source: every command is addressed to its target, and

the one broadcast the client sends — its heartbeat — is broadcast because the client has no address a server could be expected to know.

2. The receive pipeline

Markedly shorter than the server's, and every difference is deliberate.

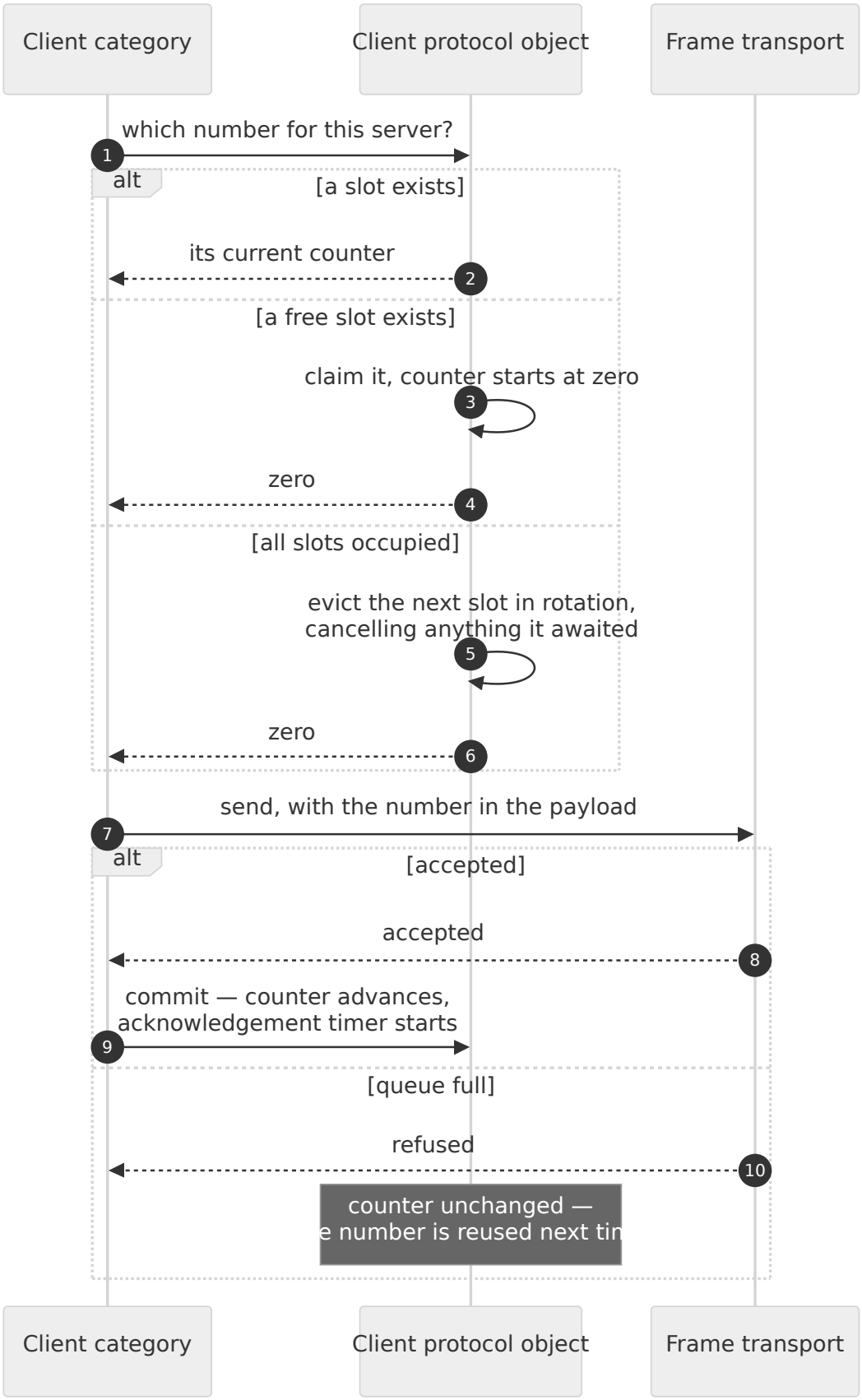


Difference from the server	Reason
No address filter	A client consumes frames from every server; the address on an incoming frame is the *source*, not a destination to match
No rate limiting	The client issues the traffic; limiting what it may receive would drop its own answers
No command/response check	A client legitimately observes broadcast heartbeats as well as responses
Acknowledgements are intercepted before dispatch	The interception needs the **source identity**, which category dispatch does not carry down
Nothing is ever acknowledged	Clients do not acknowledge

The interception is the structurally interesting part. It runs on every frame, recognises acknowledgements itself, and ignores any that is malformed or claims an impossible source — cancelling no timer and changing no counter, so the command's own timeout fires normally, which is the right outcome for a frame that cannot be trusted.

The frame still continues to category dispatch afterwards, where the system category recognises it and does nothing. That is layering rather than redundancy: the category's registered message types stay an accurate description of what it understands, while the protocol object does the part that needs the source identity.

3. Sequence numbers: peek, then commit

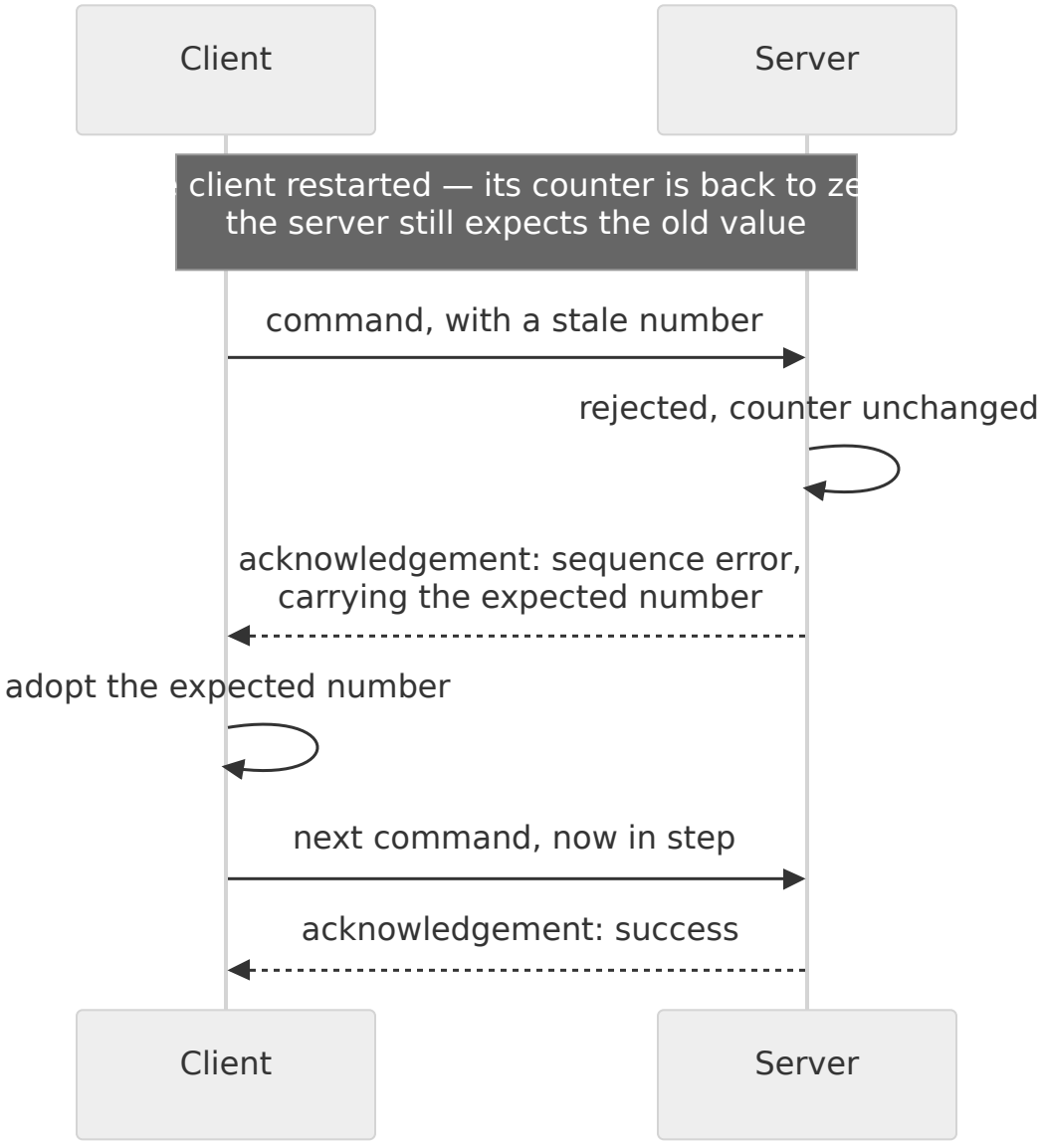


Splitting the operation in two is what makes a refused send harmless. Were the counter advanced when the number was taken, a command refused by a full queue would leave a permanent gap, and every later command would be rejected until a resynchronisation cleaned it up.

Committing for a server that was never asked for a number is a programming error, and fails loudly — reachable only by bypassing the category's own send path.

4. Resynchronisation

A rejected command comes back with the number the server expected, which turns what would be a deadlock into a single lost command.



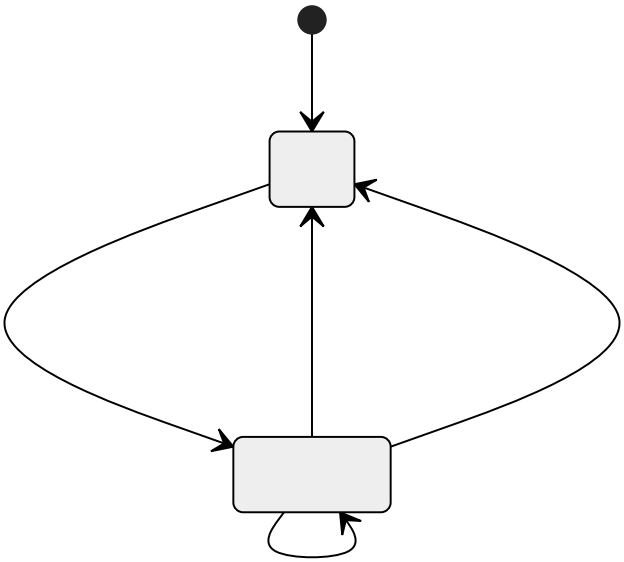
Three properties make this robust:

- **The server is never asked to reset.** It stays authoritative and the client adapts, which

keeps the recovery one-sided and therefore race-free.

- **The rejected command is not retried automatically.** Replaying a command whose side effects the application may not want repeated is not a decision the protocol layer is entitled to make. The application learns of it from the missing response, or from the acknowledgement timeout.
- **Nothing is reported.** From the application's point of view nothing has gone wrong that it can act on.

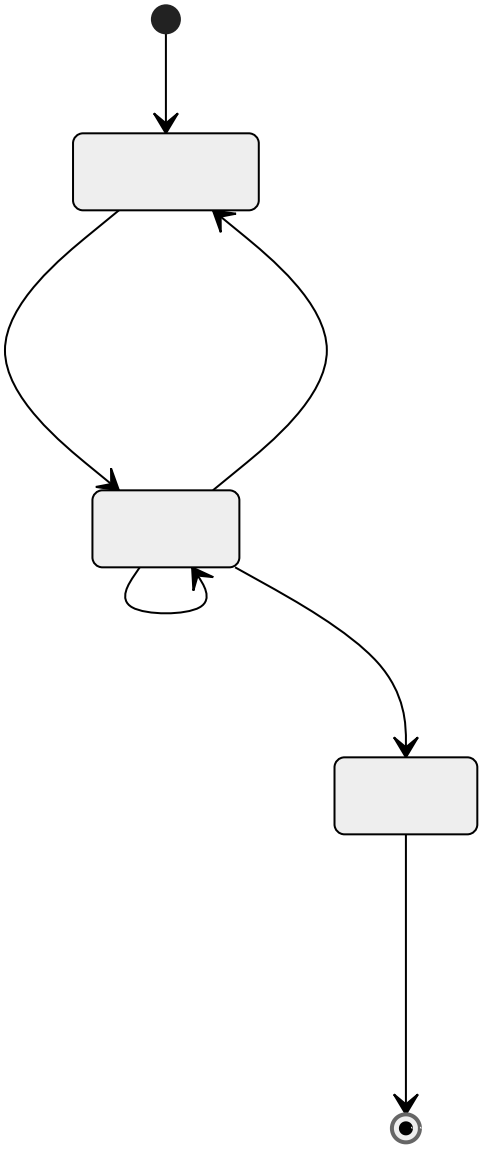
5. One outstanding command per server



The last transition is the one to know about. A second command to the same server **replaces** what is being waited for: the first command's acknowledgement, arriving afterwards, no longer matches and does not cancel the timer, which now belongs to the second command. In ordinary request/response use this never arises, and the alternative — a queue of outstanding commands per server — costs memory a one-at-a-time protocol does not need. Its sharper edge, where two consecutive commands share an identity, is entry 5.4 in Chapter 10.

Nothing is ever retried automatically. The notification names the command that was lost so the application can retry it, escalate, or mark the server suspect.

6. Tracking servers



Any frame from a server counts as proof of life — responses, telemetry and heartbeats alike — matching the server's rule and for the same reason: a busy peer defers its heartbeat.

Eviction is explicit rather than "ignore the newcomer", and it reports **both** transitions, so the application's picture stays consistent: at most eight servers online, every change reported. On a bus with more active servers than slots, the observer sees churn — which is the honest signal that the client is under-provisioned. The slot count is a compile-time property of the library, so raising it is a library change rather than a configuration choice.

7. Discovery, and its two limits

Asking a server what it implements is the one client operation whose answer the application must interpret, which is why it has a first-class entry point with a completion, while the rest of the

system conversation is bookkeeping the client does on the application's behalf.

Two limits follow from there being a single pending completion, and both are simplicity rather than oversight:

- **One discovery at a time.** A second request before the first answer overwrites the pending completion, and the first caller is never called.
- **The completion does not name the responding server.** An answer from a different server than the one asked will satisfy it.

Discovery also has **no timeout**: if the server never answers, the completion stays pending. An application that needs a deadline arms its own timer. All three are catalogued in Chapter 10, §5.

Chapter 8

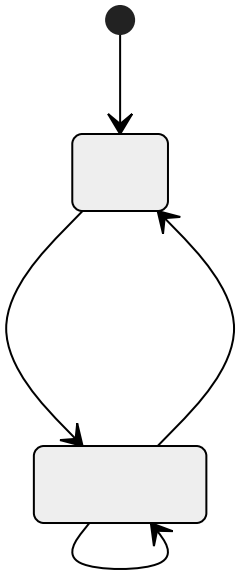
Segmentation: The ISO-TP State Machines

A classic CAN frame carries eight bytes; a firmware block or a calibration table does not fit. The transport layer solves that with ISO 15765-2 segmentation. The frame formats, the flow-control fields and the timing parameters are specified in the Protocol Specification (Chapter 15) §4.1, and the class inventory and storage pattern are in Architecture and Design Decisions (Chapter 13) §10.1. This chapter covers what neither has: **the two state machines, and the failure paths through them**.

The layer is optional and orthogonal. A node that attaches it gains multi-frame payloads for the message types that opt in; every other message type continues to travel as a single frame, unaware the transport exists.

1. Channels

A channel is a pair of identifiers: one carrying data in one direction, one carrying flow control back. Each channel owns one sender and one receiver, which are independent — a channel can be transmitting one payload while reassembling another, because the two use different identifiers.

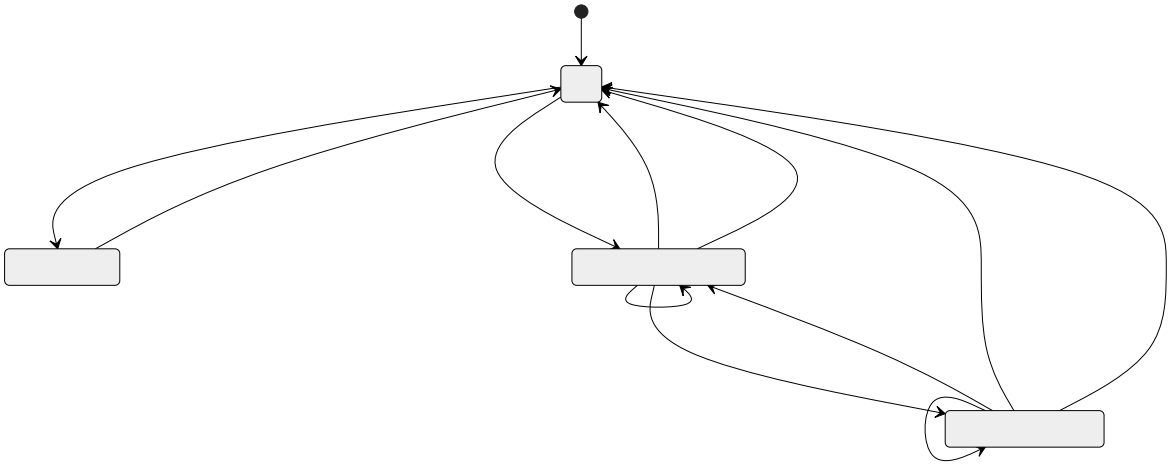


Two properties of that lifecycle matter in integration:

- **Channels are claimed, not pooled per transfer.** A channel stays occupied after a transfer completes, ready for the next one on the same identifier pair, until it is explicitly released. This is why the protocol layer releases a channel when a transfer aborts (Chapter 6, §7) — otherwise a failed transfer would hold a slot forever.
- **Identifier pairs may not overlap.** Registration is refused if either identifier already belongs to another channel, because overlapping pairs would make routing ambiguous.

Routing a received frame is a decision the channel makes and reports upward: the protocol layer offers every frame to the transport and continues its own pipeline only if the transport declines. That single boolean is the whole mechanism by which segmentation is transparent.

2. The sender



The sender refuses a transfer up front rather than failing part-way: an empty payload, one beyond the protocol's length field, one beyond the configured buffer, or a second transfer while the channel's sender is busy are all refused at the point of asking. What is accepted is then **copied into the channel's own buffer**, so the caller's buffer may be reused immediately — which matters when the source is a stack temporary or a flash read buffer.

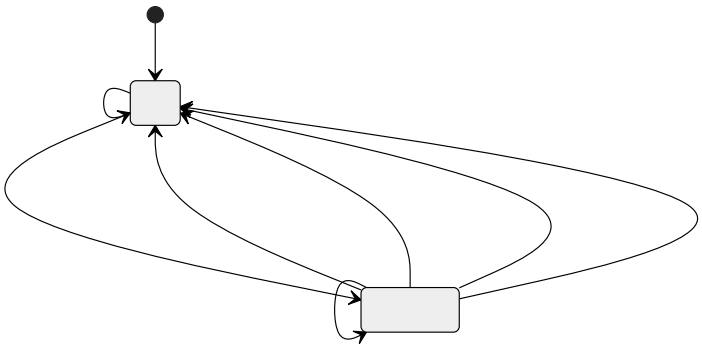
Pacing is the peer's to choose. can-lite's own receiver never asks a peer to slow down or to pause between blocks, because reassembly is a copy into a preallocated buffer and needs no pacing. Its sender nevertheless implements the general case — honouring whatever block size and separation time a peer asks for — so it interoperates with conventional stacks that do pace transfers. A pacing request it cannot parse is treated as the slowest legal value, which is the standard's conservative reading: an unparseable request must not be read as "go as fast as you like".

Aborts, and the completion that does not run

> **A transfer's completion runs on success only.** A transfer that aborts reports through the abort path instead. An application that handles only the completion will wait forever — which is why the protocol layer subscribes to the abort path and releases the channel there.

The abort reasons and what raises each are listed in the glossary. One deserves a note: a failure to hand a frame to the bus reports the same reason as an unparseable frame, because the sender genuinely cannot distinguish a full queue from a bus problem. The reason code is diagnostic; the outcome — abort, release, tell the application — is what matters.

3. The receiver

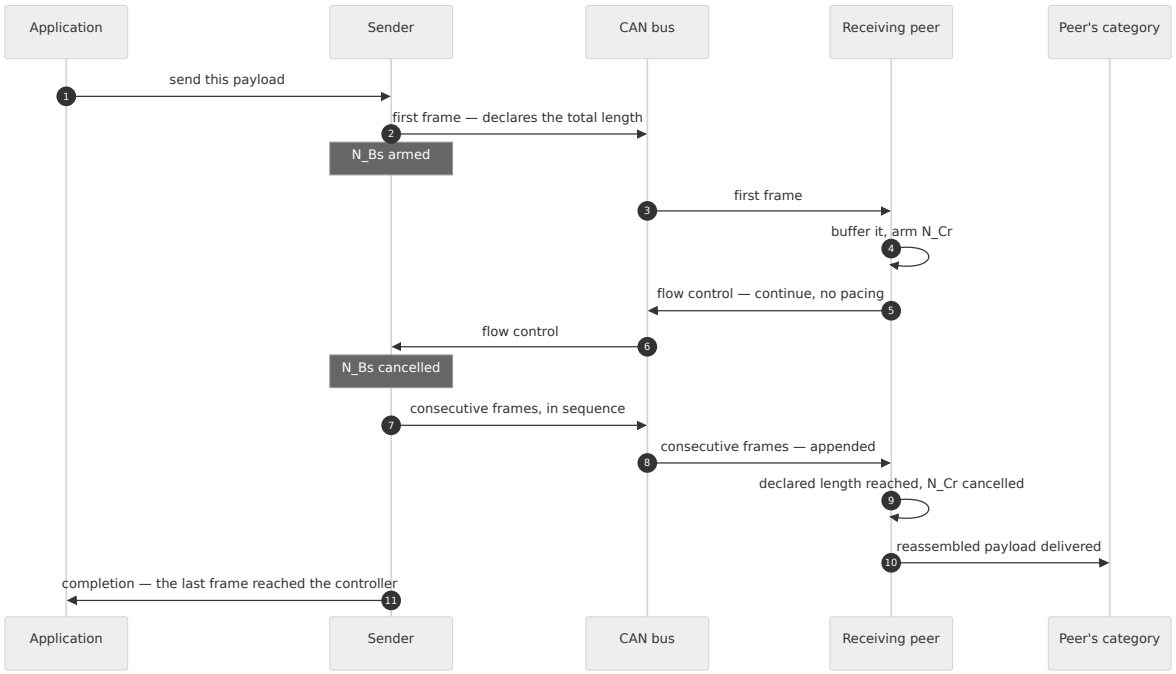


The validation it performs, in order, is where most of the interesting failure behaviour lives, and Chapter 10, §8 tabulates every case. Two asymmetries are worth understanding here rather than there:

- **An oversized first frame is answered; an oversized single frame is not.** A peer that sent a first frame is *waiting* for flow control and must be told to stop. A peer that sent a single frame has already finished and has nothing to stop.
- **An unsolicited continuation frame is ignored, not treated as an error.** Silence is the correct response to a frame that belongs to a transfer this receiver is not part of.

A single frame arriving mid-reassembly discards the partial transfer and is delivered as a complete payload — the pragmatic reading, since the peer has evidently moved on.

4. A multi-frame transfer, end to end



The two completions are independent, and neither is an end-to-end acknowledgement: the sender's says the last frame reached its own controller, the receiver's says the payload is complete. Confirming that the *peer* acted on it is the job of the category's own protocol, one layer up.

5. Wiring segmentation into a category

Three steps, none of which touches the protocol core:

1. **Attach the transport** to the server or the client, choosing the payload size and channel count — this is the sizing decision of Chapter 11.
2. **Register the receive channel** for the identifier pair the category's message will use.
3. **Give that message type a segmented handler**, alongside its single-frame one.

Because a reassembled payload runs the same pipeline as a single frame, a validated category still finds its sequence number in the first byte and still skips it. The only difference a category sees is that the payload may be longer than eight bytes.

6. Deliberate omissions

Beyond the parameters in the specification, four things are simply not implemented, and each has a cost worth knowing before choosing this transport:

Omission	Cost
Frames are never padded to full length	A peer that requires 8-byte frames will not interoperate; short frames keep a loaded bus shorter

Omission	Cost
The receiver never requests pacing	A receiver that needed it would have to become a configurable policy; nothing in the library does
Only normal addressing	No address extension, no mixed addressing
Classic CAN only	The length field and frame sizes assume 8-byte frames

The first two are the pair worth revisiting first if can-lite ever has to talk to a third-party stack, because they are the ones a conventional diagnostic tester is most likely to insist on.

Chapter 9

The Built-in Categories: Design Rationale

can-lite ships two categories. Their messages, payload layouts, states and error codes are specified in the Protocol Specification (Chapter 15) §8 (system) and the Firmware Upgrade Specification (Chapter 16) (upgrade), and the system category's role inside the protocol objects is described in Architecture and Design Decisions (Chapter 13) §6. This chapter answers the questions those documents do not: **why each is built the way it is, and what that costs.**

1. The system category is a category

Nothing forced the protocol-level conversations — presence, acknowledgement, status, discovery — to travel the same road as application messages. They could have been special-cased inside the protocol objects.

Making them an ordinary category buys three things:

- **One dispatch path.** A received frame reaches its handler by exactly one route, whatever it is for.
- **One set of rules.** The system category occupies a registration slot, uses the same payload handling and the same observer pattern as any other. That is a useful forcing function: a rule that is awkward for the system category is probably awkward for everyone.
- **Nothing special to learn.** A reader who understands one category understands this one.

What it costs is one of the eight registration slots, permanently.

Why the halves are not mirror images

The two halves handle disjoint sets of messages, and the asymmetry is informative rather than accidental:

Conversation	Server half	Client half
Presence	handles the client's broadcast	—
Acknowledgement	sends them	recognises them, acts nowhere
Status request	handles it	—

Conversation	Server half	Client half
Discovery	answers it	consumes the answer

A server's heartbeat arriving at a client finds no handler, and nothing goes wrong: the client already extracted that frame's value — proof of life — before dispatch (Chapter 7), and clients never acknowledge, so an unhandled message type on the client side is genuinely inert.

The acknowledgement handler on the client half is deliberately empty. The work happens in the protocol object, which runs first because it needs the source identity that dispatch does not carry. The handler exists so that the category's registered message types remain an accurate description of what it understands — a small piece of hygiene, not dead code.

What the application does and does not see

The server exposes **nothing** of the system category: everything it does is already reflected in the server's own interface — online and offline notifications, automatic answers to status and discovery. Exposing it would let an application answer a status request differently from the protocol's definition.

The client exposes **only discovery**, because it is the one answer an application must interpret.

Two consequences follow, and both are simplifications rather than oversights:

- **Acknowledgement statuses do not reach the application.** A client learns a command's fate from the category's own response, from the acknowledgement timeout, or — for a sequence error — not at all, because the counter resynchronises silently. An application that needs to distinguish, say, "invalid state" from "not implemented" must carry that in its own category's response, which is what the shipped categories do.
- **The heartbeat's protocol version is not checked.** It is on the wire for future negotiation; today two firmware generations with different versions will talk to each other regardless. Worth knowing before deploying a mixed bus (Chapter 10, §6.7).

2. The firmware upgrade category keeps no state

This is the library's fullest worked example of a category whose state lives in the **application**. The category owns message framing, one timer and response encoding; the flash layout, the buffering, the checksum and the bank switch belong to the product.

The division is what makes the category reusable across products with completely different storage. It is also what makes the application's obligations non-negotiable, listed in §4 below.

Why it does not validate sequence numbers

Two reasons, both practical:

1. **The block index already orders the transfer.** A duplicated or reordered block is detected by the application against its own expectation and answered with a category error that says which block was expected — which a protocol-level sequence error could not.
2. **A sequence number would cost a payload byte in the message that sends the most bytes.** One byte out of seven, on the longest transfer the protocol ever performs, is a sixth of the throughput.

The trade-off is real: this category has no protocol-level replay protection. It is bounded instead by the checksum at the end — a replayed or lost block produces a mismatch and the image is not activated — and by the authenticity check the application is expected to add.

Why polling does not extend the session

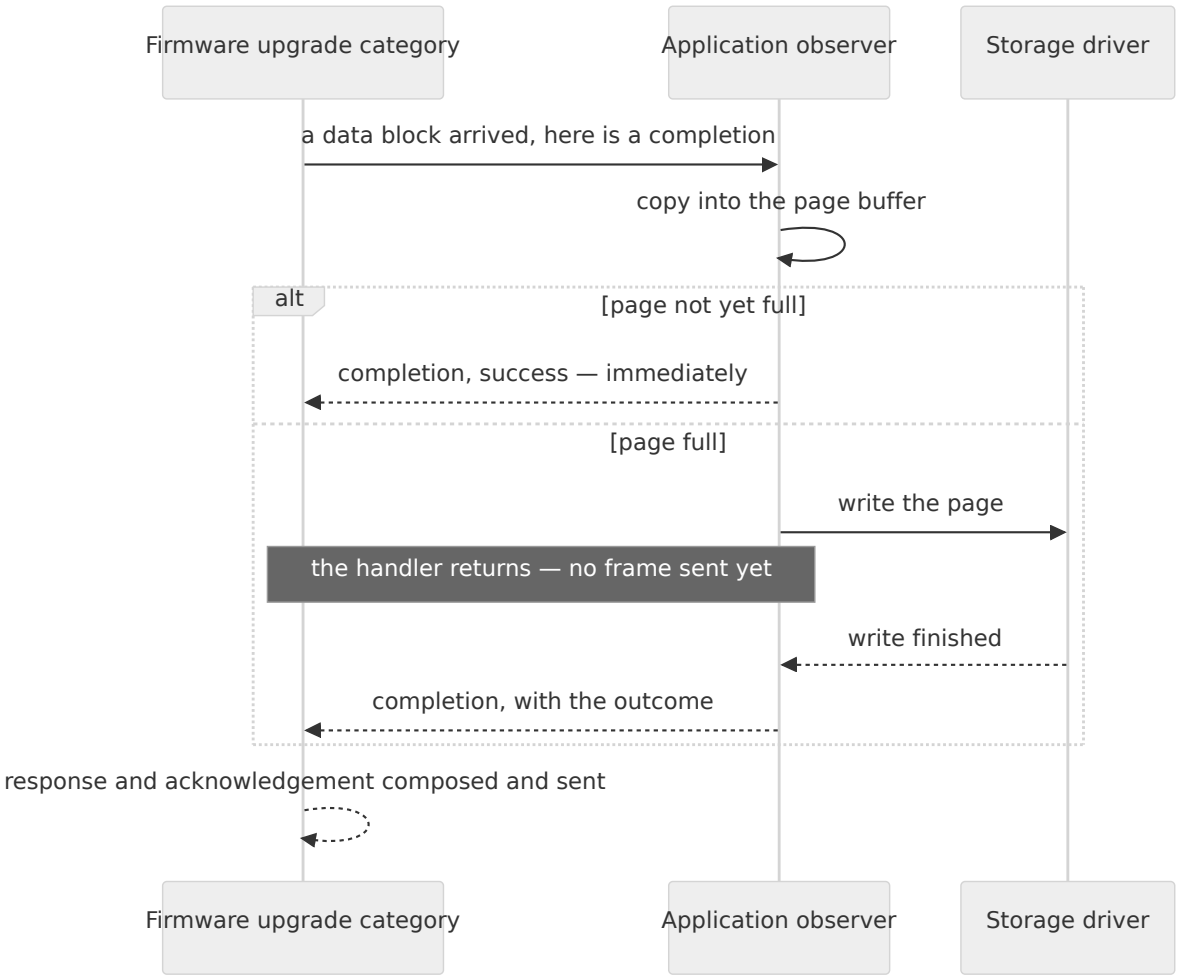
The session timer is extended by the commands that make progress and stopped by the ones that end the transfer. Asking for progress does **neither**.

That is the design decision worth remembering: a client that has crashed mid-transfer, but whose supervisor still polls for progress, must not be able to hold the server's staging area open indefinitely.

When the timer does expire the category notifies the application and does nothing else — no frame is sent, because there is nobody to tell, and no state is changed, because the state that matters belongs to the application.

The asynchronous completion pattern

Flash operations take milliseconds to seconds, and nothing in can-lite may block for that long. Every notification therefore carries a completion for the application to call when it is done, and the category retains nothing across the gap — everything it needs to compose the answer is captured when it notifies.



Two obligations fall on the application, and neither can be checked:

- **Call the completion exactly once.** Never calling it leaves the client waiting for its timeout; calling it twice sends two answers to one command.
- **The category does not serialise.** A second command arriving before the first completion has run will be dispatched. An application that cannot cope must refuse the overlapping command itself.

Why every command produces two frames back

A category response and a protocol acknowledgement answer different questions: the acknowledgement says the command was well-formed and reached a handler; the response says what happened. When a handler fails, the acknowledgement reports a category-level failure and the detail travels in the response — which is exactly the split described in Chapter 5, §2.

The cost is bus traffic: acknowledgements are nearly as expensive as the commands they answer (Chapter 11, §6). A category that carries its own status, as this one does, is the case where switching them off would save a third of the traffic — a change the protocol does not currently allow, and a fair candidate for a future extension.

3. Throughput is bounded by round trips, not by the bus

The transfer is stop-and-wait: each block waits for its acknowledgement before the next is sent. The consequence is that **round-trip latency, not bitrate, sets the throughput** — making the bus twice as fast barely helps.

That is why all three extensions recorded in the specification — windowed acknowledgement, page addressing, and carrying blocks over segmentation (Chapter 8) — attack the number of round trips rather than the frame time. The arithmetic is in Chapter 11, §4.

4. What the application must provide

Concern	What the category expects
Storage layout	Somewhere to stage an image that is not the running one
Size policy	Refuse an image that does not fit
Concurrency policy	Refuse a second session while one is open
Block ordering	Compare each index with the expected one and report a gap
Buffering	Accumulate small blocks into whatever unit the storage writes
Verification	Check the image against the client's checksum
Activation	Switch to the new image, ideally with rollback if it fails to run
Timeout recovery	Discard staging state when the session expires
Authenticity	**Verify a signature, if the product needs one**

The last row is the one to read twice. can-lite authenticates nothing: any node that can put frames on the bus can start an upgrade. Where that matters, the image must carry its own signature and the application must check it before reporting success.

Part III

Part III — Behaviour Under Stress

Chapter 10

Corner Cases and Failure Modes

This is the catalogue of what happens when things go wrong: bounds are reached, peers disappear, frames arrive out of order, configurations disagree. Each entry states the **condition**, the **mechanism** that handles it, and the **observable outcome** — what an engineer with a bus analyser and a debugger would actually see.

Entries marked **latent** are correct today but rest on an assumption worth knowing. Entries marked **design limit** are deliberate simplifications with a stated cost.

1. Frame level

#	Condition	Mechanism	Observable outcome
1.1	Standard-identifier frame received	Identifier-width gate, first in both pipelines	Ignored entirely: no answer, no counter, no notification. This is what lets can-lite share a bus with another protocol
1.2	Empty payload on a validated category	Emptiness checked before the sequence check	Answered “invalid payload”; the sequence counter is untouched
1.3	Empty payload on a non-validated category	Reaches the handler, which under-runs while reading	The handler’s validity check produces “invalid payload”; a handler that omits the check acts on zeros
1.4	Full eight-byte payload on a validated category	The sequence number occupies the first byte	Only seven bytes reach the category — a payload designed for eight silently loses its last field
1.5	Payload longer than eight bytes	Impossible: the frame type is bounded	—

#	Condition	Mechanism	Observable outcome
1.6	Driver delivers frames out of order	None — ordering is assumed	Validated commands draw spurious sequence errors; the client resynchronises each time and progress stalls. **Latent**: ordering is a driver requirement (Chapter 3, §4)
1.7	Driver completes a send from inside the send call	None	Queue drain recurses once per queued frame — at most eight deep. **Latent**: the shipped host adapters defer completions for exactly this reason

2. The outbound queue

#	Condition	Mechanism	Observable outcome
2.1	A ninth simultaneous send	Queue-full refusal	The frame is dropped and never retried. What the caller sees depends on who it was: a category's send reports failure; a client command reports failure **without consuming a sequence number**; an acknowledgement is simply lost
2.2	An acknowledgement lost to a full queue	Its completion discards the outcome	The client's acknowledgement timeout fires instead. Deliberate: retrying would deepen a queue that is already saturated
2.3	A send completion that itself sends	The queue advances before the completion runs	Safe: the new frame goes out immediately if the queue emptied, otherwise it is appended behind the one just started
2.4	Two owners claim the send notification	Checked at run time, in release builds too	Immediate assertion. Only the protocol object may claim it
2.5	The node's address is changed with frames already queued	None	Queued frames keep the identifier they were built with; only later frames carry the new address
2.6	The driver reports a transmission failure	The outcome propagates to the caller	Categories ignore it; segmentation treats it as an abort. **Design limit**: there is no bus-off recovery machine in the library

3. The server's receive pipeline

#	Condition	Mechanism	Observable outcome
3.1	Frame addressed to another node	Address filter	Silent — answering would produce noise proportional to the number of servers
3.2	Broadcast frame	The filter admits it	Accepted and dispatched; this is how the client's heartbeat reaches every server
3.3	Rate limit reached	Rate gate	Silent, including any acknowledgement that would have followed. The specification defines a "rate limited" status, but it is deliberately never sent
3.4	Burst straddling a window reset	The window is fixed and resets on a timer	Up to **twice** the configured number can be accepted in quick succession. **Design limit**: a sliding window would need a timestamp per frame (§7.2)
3.5	A response arrives at a server	Command/response gate	Silent; prevents a response being dispatched as a command
3.6	One server sees another's acknowledgement or discovery answer	Those message types are numerically in the command range, so the **address filter**, not the type gate, rejects them	Silent, because such frames carry the sending server's own address. **Latent**: the protection here is addressing, not typing
3.7	Frame for an unregistered category	Category lookup fails	Silent — the frame may be legitimate traffic for a different server
3.8	Known category, unknown message type	Dispatch reports "not handled"	Answered "unknown command"
3.9	One category too many registered	Slot limit	Registration refused; the system category holds one slot from construction
3.10	Two categories claiming the same identity	Identity scan	Registration refused — including re-registering the same object
3.11	Registration failure ignored by the application	None	The category is invisible: its commands are dropped silently (3.7) and its client waits for answers that never come
3.12	A category destroyed while still registered	None	The registration list holds a dangling entry, and the next frame for it dispatches into freed memory. **Latent**: lifetime is the application's responsibility
3.13	A category acknowledges before being registered	Checked at run time	Assertion. Only reachable when a test drives a category directly

4. Sequence validation

#	Condition	Mechanism	Observable outcome
4.1	First validated command after the server starts	The baseline is adopted	Accepted whatever its value: a client that restarts from zero needs no help from the server
4.2	A gap in the sequence	Rejection	Answered "sequence error" carrying the **expected** number; the counter does not advance
4.3	Repeated gaps	Rejection is idempotent	Every rejection names the same expected number — no drift, no lockout
4.4	The counter wraps	Modular arithmetic	Transparent
4.5	The client process restarts mid-session	Resynchronisation from the expected number	One command lost, one round trip wasted, traffic continues (§7.1)
4.6	Two clients commanding one server	One shared counter	Each breaks the other's ordering; both see near-continuous sequence errors. **Design limit**: one client per server (REQ-CAN-006.1)
4.7	One client using two validated categories on one server	One counter per server on each side	Correct: one interleaved stream, one ordering
4.8	A client sending without a sequence number to a validating server	None	The payload's first byte is read as a sequence number: sporadic rejection plus a mis-parsed payload
4.9	A client sending with a sequence number to a non-validating server	None	The sequence byte is delivered as payload, and every field is shifted by one
4.10	A send refused by a full queue	The number is taken and committed separately	The number is not consumed; the next attempt reuses it and the server sees no gap

Entries 4.8 and 4.9 are the two configuration errors with no diagnostic at all (Chapter 5, §5).

5. Client state

#	Condition	Mechanism	Observable outcome
5.1	A ninth server commanded	Round-robin eviction of a sequence slot	The evicted server's counter restarts; its next command draws one sequence error and resynchronises (§7.3)

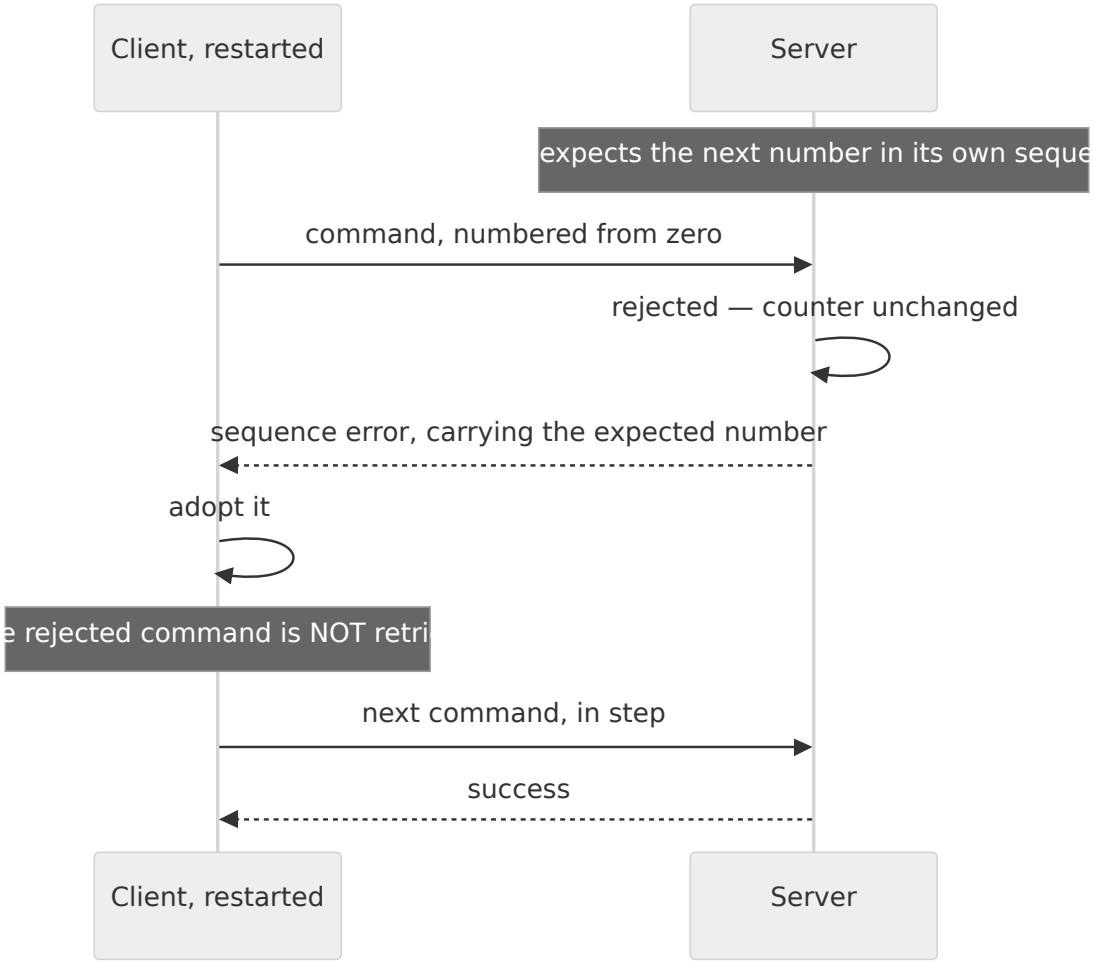
#	Condition	Mechanism	Observable outcome
5.2	A ninth server heard from	Round-robin eviction of a liveness slot	The evicted server is reported offline and the new one online. On a bus with more active servers than slots the application sees continuous churn — the honest signal that the client is under-provisioned
5.3	A second command to a server before the first is answered	The tracked command is replaced	The first answer no longer matches and does not cancel the timer, which now belongs to the second command
5.4	Two consecutive commands with the **same** identity	Matching is by category and message type only	The first command's acknowledgement cancels the second command's timer, so a lost answer to the second goes unreported. **Design limit**: one outstanding command per server
5.5	A malformed acknowledgement, or one claiming an impossible source	Guarded before it is acted on	Ignored: no timer cancelled, no counter changed; the command's timeout fires normally
5.6	An acknowledgement from a server other than the one commanded	State is kept per server	Only that server's state is touched; the commanded server's timer keeps running
5.7	A stale sequence-error acknowledgement arrives late	Applied unconditionally	The counter jumps to a value the server may have moved past, costing one further round trip. Self-correcting
5.8	Committing a number for a server never asked about	Checked at run time	Assertion. Only reachable by bypassing the category's send path
5.9	A second discovery before the first answer	A single pending completion	The first completion is overwritten and never called
5.10	A discovery answer from a different server	The completion carries no identity	The pending completion is satisfied by the wrong server's answer
5.11	Discovery never answered	No timeout exists	The completion stays pending forever; an application needing a deadline arms its own timer
5.12	No observer attached	Notification checks first	Safe: events are discarded silently

6. Liveness and heartbeat

#	Condition	Mechanism	Observable outcome
6.1	The node is busy sending	The heartbeat timer restarts on every outgoing frame	No heartbeat at all; liveness is carried by the traffic itself
6.2	A client commands continuously and never sends a heartbeat	The server restarts its timer on **any** correctly addressed frame	The client is not declared offline for being busy
6.3	The bus goes silent	Liveness timers expire	Both sides report their peer offline
6.4	A prolonged outage	The offline state is remembered	Reported once, not once per timer expiry
6.5	A server heartbeat reaches the client	Liveness is marked before dispatch	Reported online; the frame then finds no handler in the client's system category, which is harmless
6.6	A hardware acceptance filter that excludes the broadcast address	None	The server never sees its client's heartbeat and reports it offline once per timeout on an idle bus. **Latent** (Chapter 3, §4)
6.7	Peers with different protocol versions	The version is carried but not checked	They communicate regardless. **Design limit**: version negotiation exists on the wire only

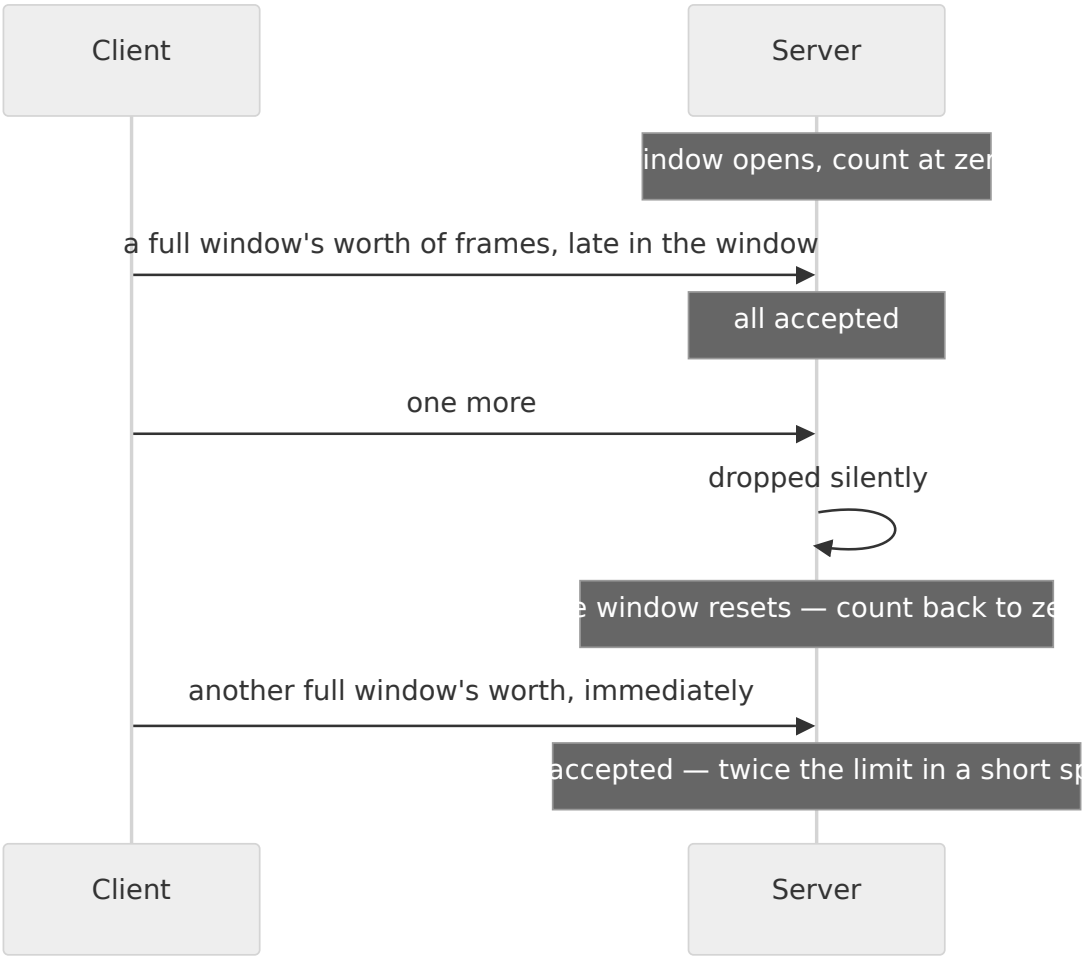
7. Four walkthroughs

7.1 A client restart mid-session



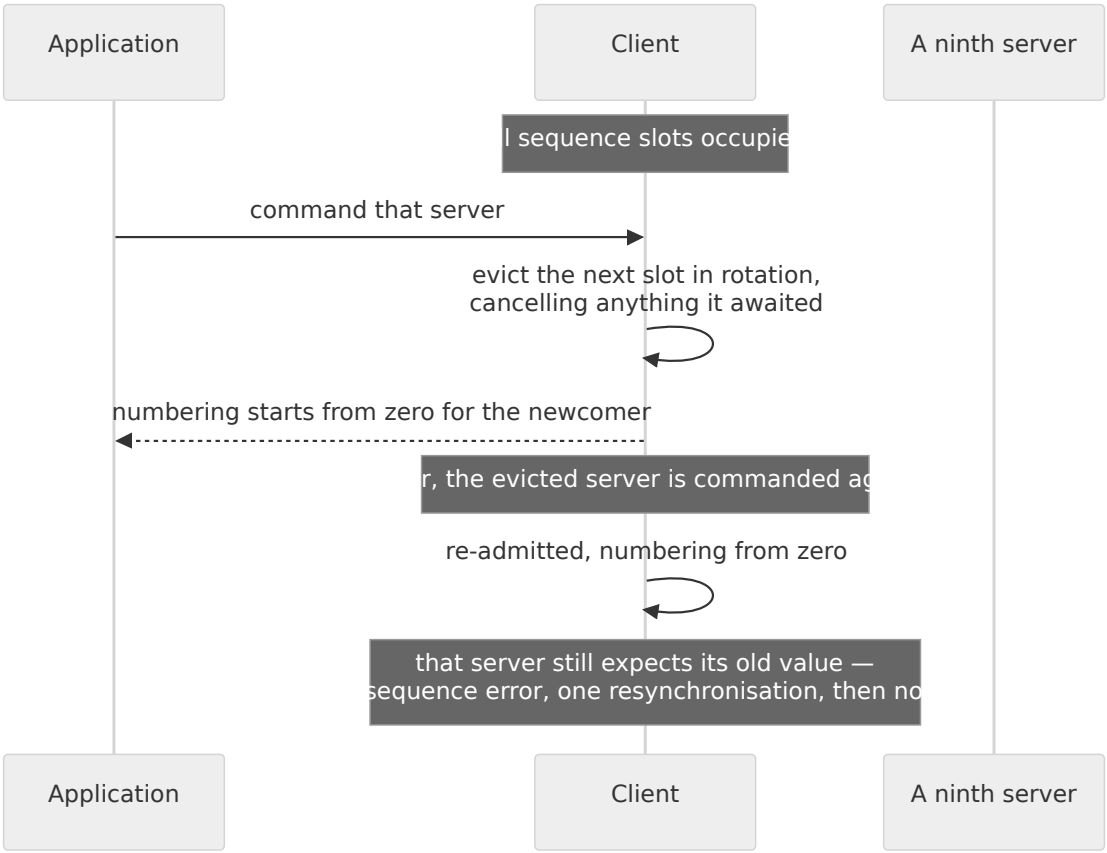
The lost command is the application's problem, deliberately: replaying a command with side effects is not the protocol layer's decision to make. The application learns of it from the missing response, or from the acknowledgement timeout.

7.2 The rate-limit window boundary



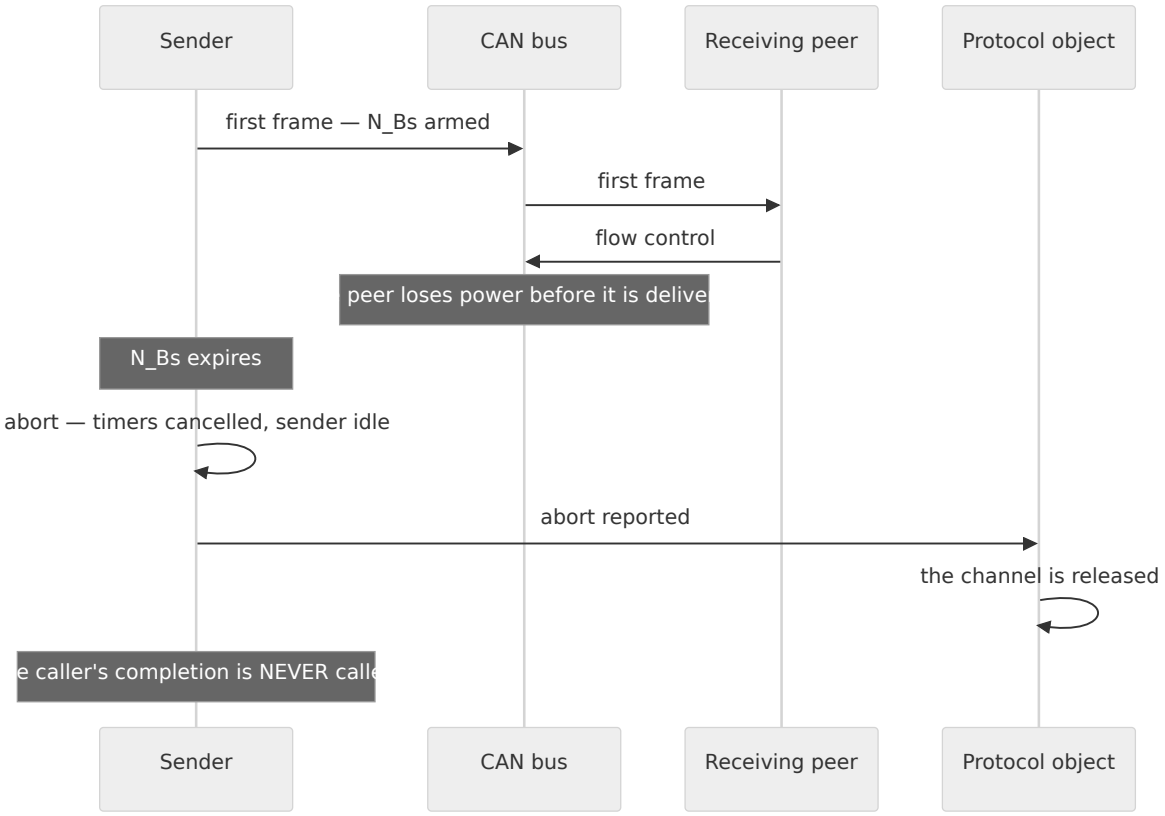
The limit is a **tumbling** window, not a sliding one. Sizing a server therefore means budgeting for twice the number in the worst case, or configuring half of what the hardware can absorb.

7.3 The ninth server



The cost of eviction is bounded and self-healing: one wasted round trip per re-admission. Refusing to talk to a ninth server would fail harder for no benefit.

7.4 A transfer abandoned mid-flight



The last line catches applications out. The completion means “the transfer succeeded”; failure arrives on the abort path, and an application that handles only the completion waits forever.

8. Segmentation

#	Condition	Mechanism	Observable outcome
8.1	Payload beyond the protocol’s length field, or beyond the configured buffer	Refused when the transfer is requested	Nothing is transmitted
8.2	Empty payload	Refused	Segmentation has no zero-length encoding
8.3	A second transfer while the channel’s sender is busy	Refused	One transfer at a time per channel; more concurrency means more channels
8.4	No free channel	Refused	Both registration and transmission report it
8.5	Registering an identifier already used by another channel	Lookup matches either identifier of a pair	Refused; overlapping pairs would make routing ambiguous
8.6	A channel never released after an abort	Application responsibility	The slot stays occupied. The protocol layer releases it automatically, so this bites only a direct user of the transport

#	Condition	Mechanism	Observable outcome
8.7	No flow control within N_Bs	Sender timer	Abort
8.8	No continuation frame within N_Cr	Receiver timer	Abort; the partial payload is discarded
8.9	The peer asks to wait too many times in a row	Wait limit	Abort
8.10	The peer reports overflow	Flow-status check	Abort
8.11	A continuation frame out of sequence	Sequence check	Abort; the partial payload is discarded
8.12	The frame sequence number wraps	Modular on both sides	Transparent; long payloads are ordinary
8.13	A first frame declaring a length that would fit in one frame	Length check	Abort — such a payload must be sent as a single frame
8.14	A first frame shorter than a full frame	Size check	Abort
8.15	A first frame declaring more than the buffer holds	Capacity check	Overflow **is signalled** to the peer, then abort
8.16	A single frame longer than the buffer	Capacity check	Abort, **without** signalling — the peer has already finished
8.17	A single frame arriving mid-reassembly	The partial transfer is discarded	The single frame is delivered as a complete payload
8.18	An unsolicited continuation frame	State check	Ignored silently — not an error
8.19	An extra continuation frame after completion	The receiver is idle again	Ignored
8.20	A frame carrying more bytes than remain	Bounded by the declared length	Surplus dropped, so padding from a padding-using peer is harmless on the last frame
8.21	A send fails mid-transfer	The failure is reported as an abort	The reason is imprecise, the outcome correct
8.22	A pacing request the standard reserves	Mapped to the slowest legal value	Conservative by design
8.23	A peer that requires padded frames	can-lite never pads	No interoperation. **Design limit** (Chapter 8, §6)

9. Category level

#	Condition	Mechanism	Observable outcome
9.1	A validated handler that does not skip the sequence number	None	Every field is shifted by one byte; values are plausible and wrong. Not detectable at compile time
9.2	A handler that does not check payload validity	Reads return zeros	The command executes with zeroed parameters
9.3	A completion never called	None	No response, no acknowledgement; the client's timeout fires
9.4	A completion called twice	None	Two answers to one command
9.5	Overlapping commands in a stateful category	The category does not serialise	A second command is dispatched while the first is pending; a category that cannot cope must refuse it itself
9.6	A client crashing mid-upgrade	The session timer	The application is told to release its staging state. No frame is sent — there is nobody to tell
9.7	A supervisor polling progress while the client is dead	Polling does not extend the session	The session still expires, as intended
9.8	A firmware block lost or replayed	The application compares block indices	Reported as a category error; the checksum at the end is the backstop
9.9	An observer that allocates or blocks	None	Heap use on a no-heap target, or a stalled receive path. The rule is stated, not enforced

10. Configuration and topology

#	Condition	Mechanism	Observable outcome
10.1	A server configured with the broadcast address	Checked at construction	Assertion
10.2	Two servers sharing an address	None	Both accept the same commands and both answer; the client attributes every answer to one node and their counters diverge. Addresses must be unique by construction
10.3	Two protocol objects on one bus interface	The interface holds a single receive callback	The second construction replaces the first's callback, and the first stops receiving entirely
10.4	A liveness timeout no longer than the peer's heartbeat interval	None	On an idle bus the peer times out before the heartbeat arrives, producing an online/offline flap (Chapter 11, §2)
10.5	An acknowledgement timeout shorter than the server's worst-case handler latency	None	Spurious timeouts for commands that are merely slow

#	Condition	Mechanism	Observable outcome
10.6	A rate limit below the client's steady-state command rate	Rate gate	Commands are dropped silently, and the client sees timeouts with no explanation on the bus

11. What is deliberately not handled

Not handled	Consequence	Mitigation
Authentication and integrity	Any node on the bus can command any server, including a firmware upgrade	Application-level signature checks; physical bus security
Confidentiality	Payloads are visible to every node	Application-level encryption
Replay by an attacker	Sequence numbers deter accidents, not adversaries	Application-level nonces or signatures
Bus-off recovery	Send failures are reported, but there is no recovery machine	Driver-level recovery, plus the liveness notifications to detect the outage
Duplicate address detection	Silent misbehaviour (10.2)	Assign addresses by construction or provisioning
Priority inside the outbound queue	The queue is first-in-first-out, so an emergency frame queued behind telemetry waits for it	Keep the queue shallow; prioritisation happens on the bus, not in the queue
More than eight servers, categories or queued frames	Eviction, refusal or a dropped frame as catalogued above	Compile-time limits; raising them is a library change

Chapter 11

Timing, Memory and Bus Budget

Everything can-lite consumes is fixed at build time or chosen at construction. This chapter collects the arithmetic: which timers exist, how the tunable parameters constrain each other, what the static memory is spent on, and how much of the bus a traffic pattern uses.

The parameters themselves and what each one means are documented with the components that own them — see Architecture and Design Decisions (Chapter 13) §9 and the Protocol Specification (Chapter 15) §12.

1. Timer inventory

Owner	Timer	Kind	Restarted by	On expiry
Server	heartbeat	single-shot	every outgoing frame from this node	send a heartbeat
Server	rate window	**repeating**	itself	reset the accepted-frame count
Server	client liveness	single-shot	every frame addressed to this node	report the client offline
Client	heartbeat	single-shot	every outgoing frame from this node	broadcast a heartbeat
Client	server liveness, one per slot	single-shot × 8	every frame from that server	report that server offline
Client	acknowledgement, one per slot	single-shot × 8	committing a command to that server	report the command unanswered
Firmware upgrade	session	single-shot	the commands that make progress	report the session expired
Segmentation sender	N_Bs	single-shot	sending a frame that awaits flow control	abort the transfer
Segmentation sender	separation	single-shot	each frame, when the peer asks for pacing	send the next frame

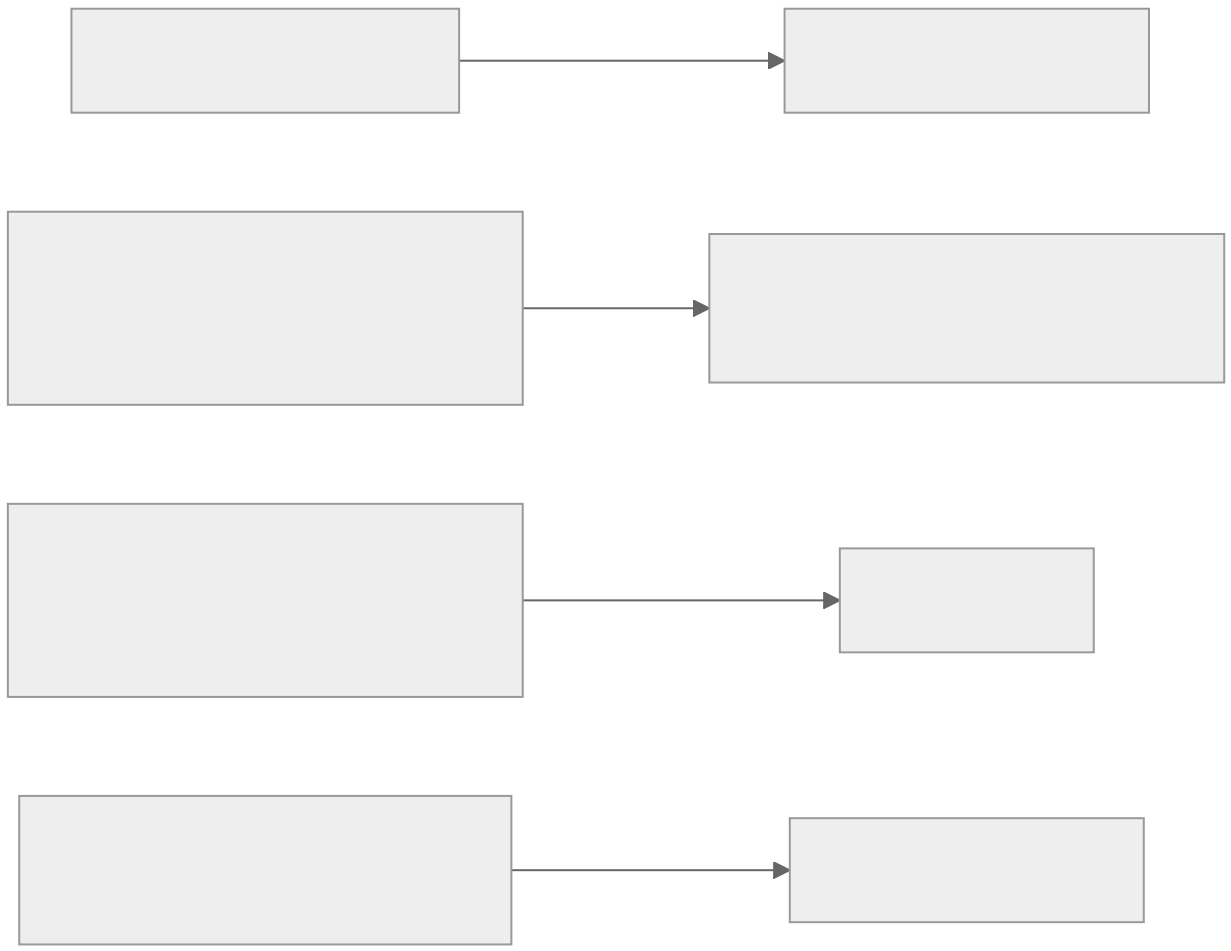
Owner	Timer	Kind	Restarted by	On expiry
Segmentation receiver	N_Cr	single-shot	each accepted frame	abort the reassembly

A fully loaded client — eight servers tracked, eight commands outstanding — holds **seventeen** live timers. A server holds three, plus one per firmware upgrade category and two per open segmentation channel. All of them are members; none is allocated.

Exactly one timer in the library is periodic. An idle server has two timers armed, an idle client one; everything else is dormant until something happens.

2. How the parameters constrain each other

The defaults work together; changing one in isolation is where trouble starts.



Relationship	Why
Liveness timeout at least three heartbeat intervals	A timeout must survive two lost heartbeats, or an idle bus flaps between online and offline (Chapter 10, §10.4)

Relationship	Why
Acknowledgement timeout above the worst-case handler latency	It must cover asynchronous work the observer defers — a flash write, a conversion — plus the queueing delay of a busy bus
Rate limit at twice the peak rate	The window is tumbling, so a burst straddling a reset delivers up to double (Chapter 10, §7.2)
Session timeout above the client's own block cadence	Usually dominated by the client's flash read or network fetch, not by the bus

The limits that are **not** tunable, because they are compile-time properties: registered categories per node, tracked servers per client, outbound queue depth, segmentation channel ceiling, and the two segmentation timeouts and wait limit. Raising any of them is a library change, and each has its own entry in Chapter 10.

3. Static memory

can-lite allocates nothing, so its footprint is the sum of its members. The counts are fixed by design; the per-element sizes depend on the toolchain, so the honest way to get exact numbers is to ask the compiler for the size of the composed objects on the target.

What the structure guarantees, independent of toolchain:

Component	Fixed-count storage
Frame transport	eight queued frames — identifier, payload and a completion each — plus two callback slots
Server	one transport, three timers, the system category, a handful of counters and flags
Client	one transport, one timer, the system category, eight sequence slots and eight liveness slots, each with a timer
Segmentation sender and receiver	one payload-sized buffer each, one or two timers, a few bytes of state
Segmentation channel	one sender, one receiver, three callback slots
Segmentation transport	its channels, a pointer array and two callback slots
Each category	its own members, plus one binding per message type

The dominant term is almost always segmentation:

```
segmentation bytes ≈ channels × (2 × maximum payload + per-channel overhead)
```

Four channels of one kibibyte is therefore about **8 KiB** of buffers, and two channels of 128 bytes about **512 bytes** — and the arithmetic is deliberately that obvious. This is the single most consequential memory decision an integrator makes, and it is a hard ceiling rather than a performance knob: a payload larger than the configured size is refused at both ends (Chapter 10, §8.1).

Stack use is bounded by the deepest call chain — receive, segmentation, category, observer, send. Nothing recurses except queue drain, and only if a driver breaks the asynchronous-completion

assumption (Chapter 10, §1.7).

4. Bus arithmetic

An extended CAN frame costs 67 bits of framing plus 8 bits per payload byte, before bit stuffing; worst-case stuffing adds roughly a quarter of the stuffable field.

Frame	Payload bytes	Nominal bits	Worst case
Heartbeat	1	75	≈ 91
Acknowledgement	4	99	≈ 121
Typical validated command	7	123	≈ 151
Full frame	8	131	≈ 160

Frame time for a full frame (nominal / worst case):

Bitrate	Bit time	Full frame
125 kbit/s	8 μs	1.05 ms / 1.28 ms
250 kbit/s	4 μs	524 μs / 640 μs
500 kbit/s	2 μs	262 μs / 320 μs
1 Mbit/s	1 μs	131 μs / 160 μs

What a rate limit means in bus load. Five hundred full frames per second at 500 kbit/s is about 131 ms of bus time — roughly 13 %, leaving ample room for other servers. The same limit at 125 kbit/s is over half the bus, and is certainly too high: the limiter protects a *server's processing* budget, but at low bitrates the bus is the scarcer resource.

What a command exchange costs. A validated command with a response is three frames — command, response, acknowledgement — about 750 μs of bus time at 500 kbit/s.

Firmware upgrade throughput. Three frames carry six payload bytes, so the line-rate bound is roughly 9 KiB/s at 500 kbit/s and 19 KiB/s at 1 Mbit/s. The bound that actually applies is different:

Bound	500 kbit/s	1 Mbit/s
Line rate only	≈ 9.3 KiB/s	≈ 18.7 KiB/s
With a 1 ms round trip, stop-and-wait	≈ 5.4 KiB/s	≈ 5.7 KiB/s

The second row is the real one, and the reason is in Chapter 9, §3: the transfer waits for each block to be acknowledged, so **round-trip latency sets the throughput**. A 256 KiB image takes about 48 seconds — a service-window operation, not a boot-time one.

Segmented transfer. A payload of n bytes costs one first frame plus $\lceil (n - 6) / 7 \rceil$ continuation frames and one flow-control frame. A kibibyte is 147 frames, about 39 ms at 500 kbit/s — some forty times more efficient per payload byte than the same data sent as firmware blocks, because there is no per-block turnaround.

5. Processing cost

Operation	Cost
Identifier decode	a few shifts and masks, evaluable at compile time
Category lookup	linear over at most eight entries
Message-type lookup	linear over the category's bindings, typically two to six
Sequence validation	one increment, one comparison
Rate limiting	one comparison, one increment
Payload access	a bounds check and byte moves; no allocation
Segmentation channel lookup	linear over at most sixteen channels

The fixed part of the receive path is well under a microsecond on a hundred-megahertz core — one to two orders of magnitude below the frame time even at 1 Mbit/s. **The bus, not the CPU, is the constraint.**

The caveat is that all of it runs inside the driver's receive callback, so a handler doing real work there delays the next frame. That is the mechanical reason behind the rule that observers neither allocate nor block, and that long work is deferred behind a completion.

6. A worked budget

A four-axis machine: one controller and four drives at 500 kbit/s, each drive receiving 200 set-points per second and returning telemetry at 100 Hz.

Traffic	Frames/s	Bits/s	Bus load
Set-points, 7 bytes	800	98 400	19.7 %
Acknowledgements, 4 bytes	800	79 200	15.8 %
Telemetry, 8 bytes	400	52 400	10.5 %
Heartbeats	≈ 0	≈ 0	≈ 0 %
Total	**2000**	**230 000**	**46 %**

Four observations worth carrying into a real design:

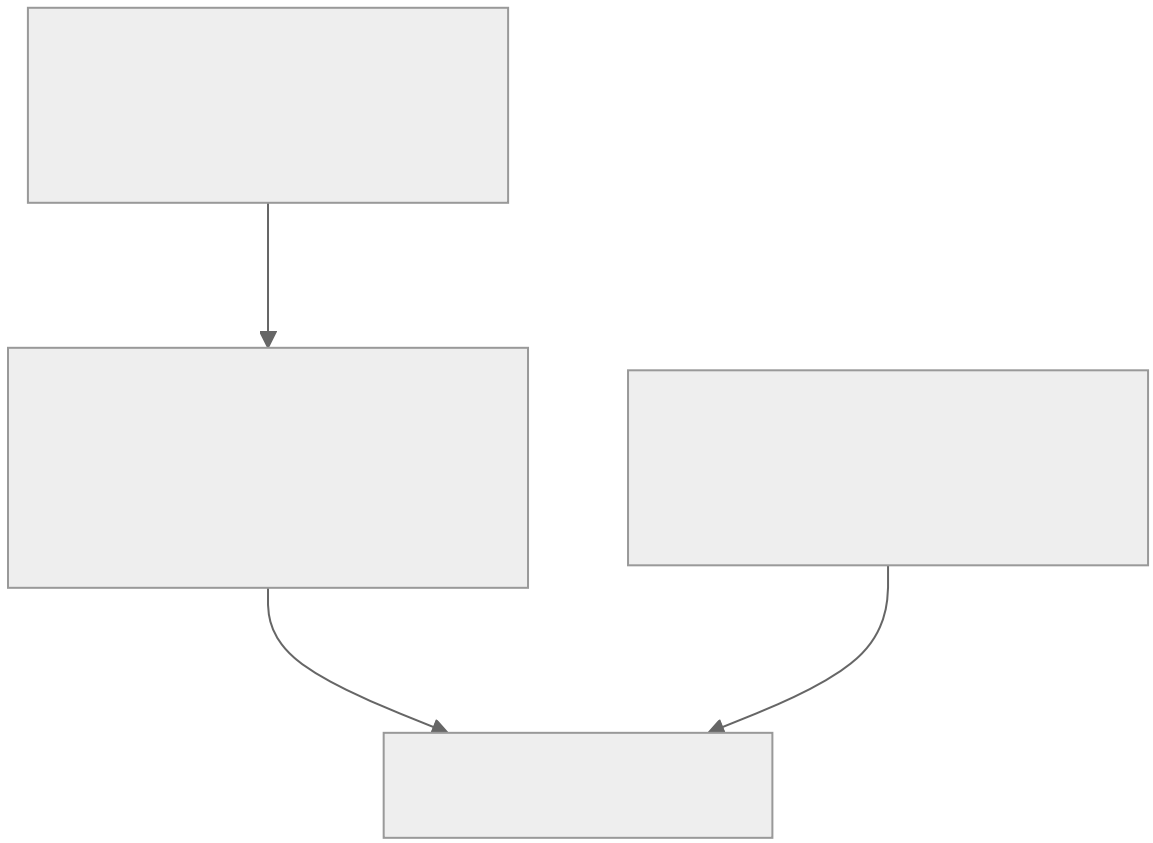
- **Acknowledgements are not free.** They cost nearly as much bus time as the commands they answer. A category that carries its own status — as the firmware upgrade category does — is where switching them off would save a third of the traffic.
- **Check the rate limit against the real rate.** Each drive here sits inside the default; a design with three times the set-point rate would be clipped silently (Chapter 10, §10.6).
- **Forty-six percent is a reasonable target.** Latency for low-priority frames grows quickly above roughly 60–70 % load, and telemetry is given a lower priority than commands and responses precisely so that it yields first.
- **Heartbeats vanish on a busy bus.** The silence-guard design contributes nothing to a loaded bus and appears only when the machine is idle — which is exactly when there is room for it.

Chapter 12

Verification Strategy

The integration test architecture — the shared fixture, the virtual bus, the strict-mock rule and the Gherkin layout — is described in Architecture and Design Decisions (Chapter 13) §12. This chapter adds the two things that document does not state: **how the three levels of verification relate to each other**, and **what is not covered**.

1. Three levels



Level	Question it answers
Unit	Does this component do what it promises, including at its bounds?
Integration	Do a real server and client, wired as an application would wire them, hold the right conversation?
Requirements	Is there a stated requirement behind this behaviour?

Coverage is organised so that each level answers what the others cannot: unit tests own the bounds and the failure paths (every abort reason, every refusal, every wrap-around); scenarios own the conversations (sequence validation, discovery, rate limiting, error handling, firmware upgrade, and the reference application category end to end); requirements own intent.

2. Three rules that shape every test

Only strict mocks. An unexpected call fails the test. That is what turns "the server sends *nothing* when the frame is for another node" — half of Chapter 10 — from an assumption into an assertion.

Mock the bus, not the protocol. Tests drive real protocol objects and observe the frames handed to a mocked bus. Nothing above the hardware interface is stubbed, so the tests exercise the dispatch, validation and timing code the target runs.

Time is controlled, never waited on. The timer service is replaced by a controllable clock, so a three-second liveness timeout and a thirty-second firmware session cost the same as a heartbeat: microseconds. No test in can-lite sleeps.

3. How the virtual bus earns its place

The virtual bus is a real implementation of the hardware interface rather than a mock, which lets a scenario do two different things:

- **Couple two nodes** so the conversation is genuinely end to end.
- **Inject a frame** that no correct peer would send — a malformed acknowledgement, a wrong sequence number, an unknown category, a standard-identifier frame — and disconnect to model bus loss.

The second is what makes most of Chapter 10 testable at the integration level rather than only in unit tests.

4. Traceability, honestly

Requirements are schema-validated on every push and pull request, and Appendix A of this booklet is generated from the same files at build time, so the booklet and the requirements cannot drift.

Two caveats are recorded in the requirements themselves and are worth repeating where a reader will see them:

- **Traceability is not enforced.** Scenario tags naming requirements are a convention; no CI step fails when a requirement has no covering test.

- **Tagging is partial by construction.** Tags exist where the mapping is unambiguous; much of the suite verifies behaviour that several requirements imply jointly.

Closing the gap needs a schema change plus a CI step that fails on an uncovered requirement — follow-up work, deliberately not approximated with an inaccurate mapping.

5. Where verification stops

Stated plainly, because a verification chapter that lists only what is covered is misleading:

Not covered	Why	What stands in for it
Arbitration and priority	The virtual bus delivers in send order	The bus-load arithmetic of Chapter 11
Bus errors, error frames, bus-off	Not modelled	Failure paths driven through mocked send outcomes
Timing on target hardware	Host tests use a virtual clock	Budgets computed in Chapter 11, validated per product
More than two nodes end to end	The fixture couples one server and one client	Multi-server behaviour covered at the unit level
Interoperability with third-party segmentation stacks	No external stack in the loop	The deliberate omissions of Chapter 8, §6
Long-run behaviour — counter wrap over days, timer drift	Not simulated	Wrap-around tested directly at its boundary

Each row is a place where Chapter 10 relies on reasoning rather than on a test, which is precisely why that chapter states the mechanism behind every outcome.

Part IV

Part IV — Reference Documents

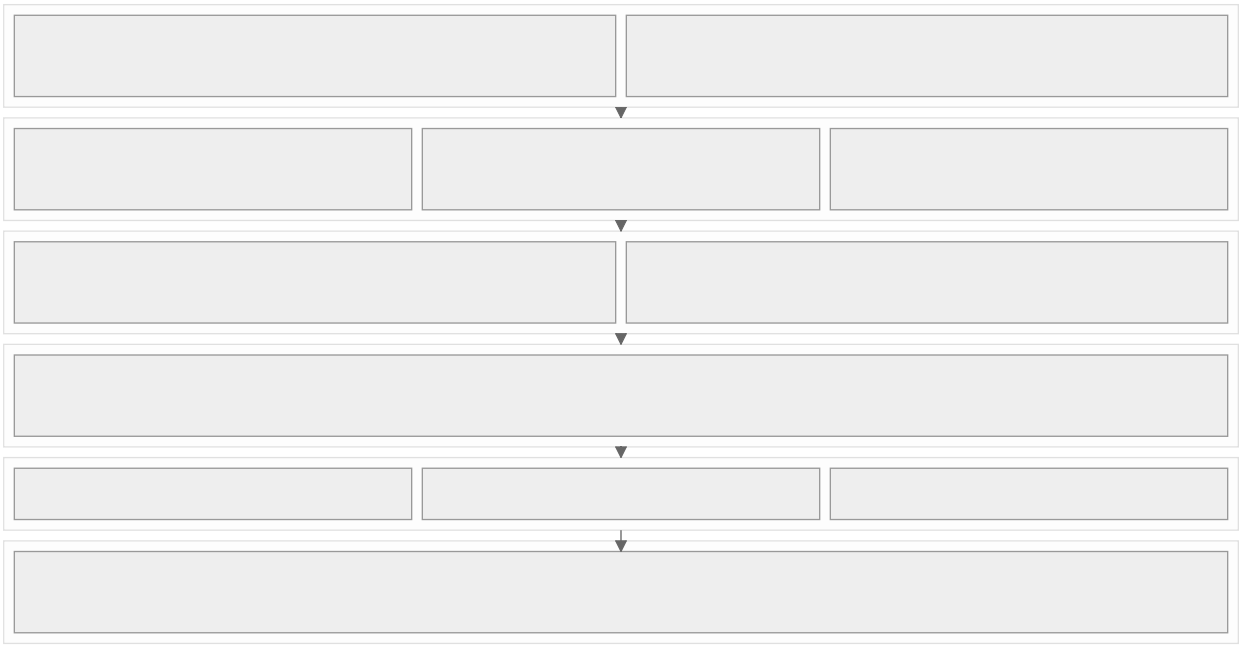
Chapter 13

can-lite: Architecture & Design Decisions

Status: Living document **Last updated:** 2026-08-20

1. Overview

can-lite is a lightweight CAN bus protocol library implementing a client-server model over CAN 2.0B (29-bit extended identifiers). It is designed for bare-metal embedded systems with strict constraints: no heap allocation, bounded memory, and deterministic timing.

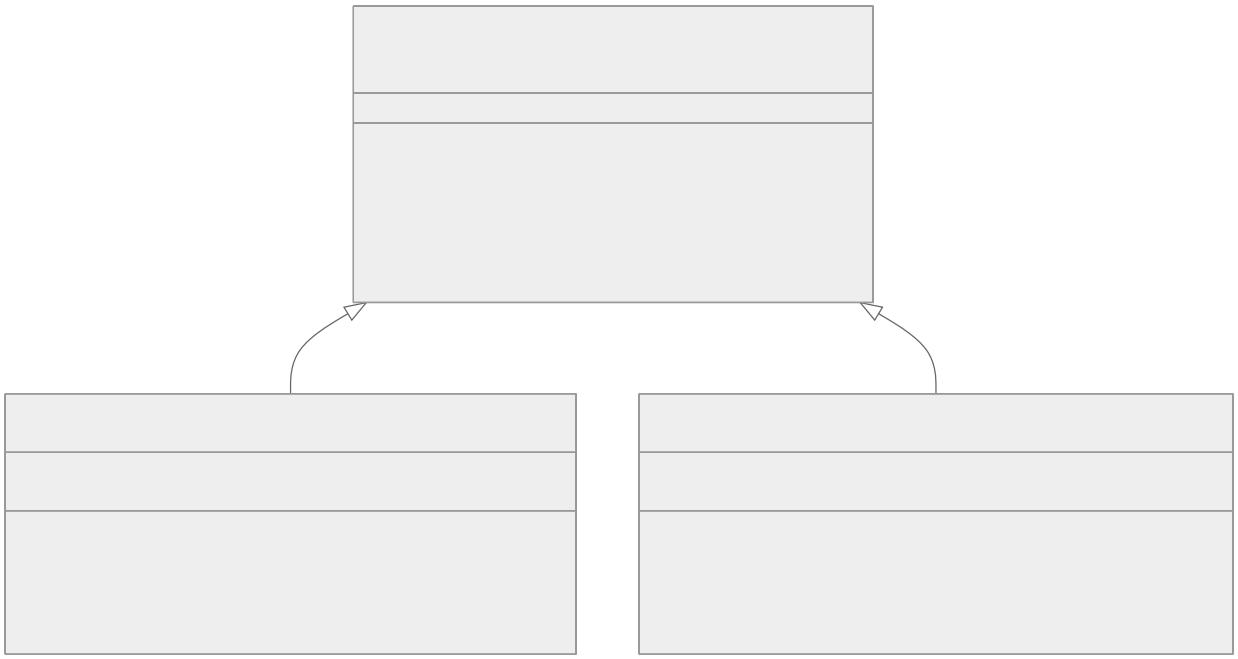


2. Design Principles

Principle	Rationale
No heap allocation	Target MCUs have limited SRAM; all containers use `infra::BoundedVector`, `infra::BoundedDeque`, `infra::Function`, etc. from embedded-infra-lib.
Type-safe server/client separation	Prevents accidental registration of a client-side category handler on the server (and vice versa) at compile time.
Observer pattern over callbacks	Consistent notification mechanism using `infra::Subject` / `infra::SingleObserver`. Avoids storing `infra::Function` objects for event dispatch; observers auto-attach and auto-detach on construction/destruction.
Fixed-point encoding	Floating-point values are transmitted as scaled integers to avoid FPU dependencies and ensure deterministic wire representation.
Domain-neutral library	can-lite ships only protocol-level categories (System, Firmware Upgrade). Anything application-specific lives in the consuming project as an application category.
Extensible via categories	New functionality is added by implementing a category handler — no protocol core changes required. See Extending can-lite with categories (Chapter 14).

3. Category Type Hierarchy

A key architectural decision is the **compile-time separation** of server-side and client-side categories. This prevents a `MyCategoryClient` from being accidentally registered on a `CanProtocolServer`.



`CanCategory` holds the shared logic: a list of `CanMessageType` handlers, message dispatch via `HandleMessage()`, and the pure virtual `Id()` and `RequiresSequenceValidation()`.

`CanCategoryServer` and **`CanCategoryClient`** each carry their own `IntrusiveList` node type, making them incompatible with each other's lists. `CanProtocolServer` holds an `IntrusiveList<CanCategoryServer>` and `CanProtocolClient` holds an `IntrusiveList<CanCategoryClient>`, so type safety is enforced at the compiler level.

Both base classes own the `CanFrameTransport` reference and expose protected send helpers that fill in the category ID and priority, so a concrete category never touches the frame layer directly. Client categories additionally take a `CanSequenceSource`, which supplies the per-server sequence byte — this keeps categories independent of `CanProtocolClient`, so they link `can_lite.core` only.

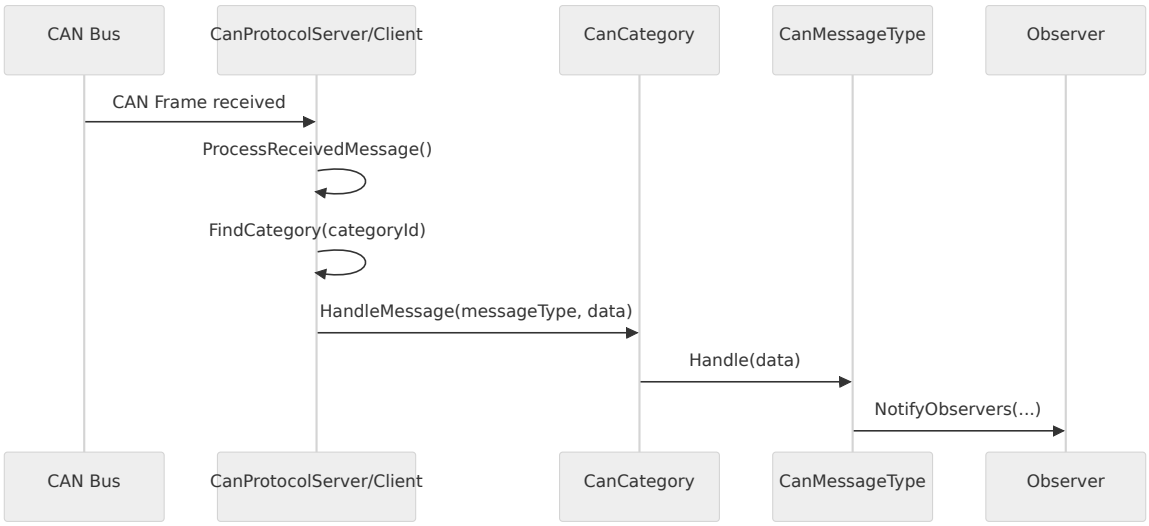
Default sequence validation: - **Server categories** default to `true` — incoming commands carry a sequence byte for replay protection. - **Client categories** default to `false` — responses do not require sequence validation.

4. Message Type Dispatch

Each category contains a set of `CanMessageType` handlers, registered via `AddMessageType()` / `AddMessageTypes()` in the category constructor. In practice these are `CanMessageHandler<Owner>` instances, which bind a message type ID to a member function of the owning category — an ID, a reference and a member pointer, with no allocation and no nested class per message. When a frame arrives:

1. `CanProtocolServer`/`CanProtocolClient` extracts the category ID and message type from the 29-bit CAN identifier.
2. The corresponding category's `HandleMessage()` is called.
3. `HandleMessage()` iterates the registered message types and dispatches to the matching handler.

4. The handler parses the payload and notifies the observer.



5. Observer Pattern

All category handlers use `infra::Subject<Observer>` / `infra::SingleObserver<Observer, Subject>` for event notification. This replaces earlier `infra::Function` callbacks.

Key properties: - **Single observer for categories**: Every category subject supports exactly one observer (`infra::SingleObserver`). This matches the 1:1 relationship between a category instance and its consumer. - **Multiple observers at the protocol layer**: `CanProtocolServerObserver` and `CanProtocolClientObserver` derive from `infra::Observer`, so a `CanProtocolServer`/`CanProtocolClient` accepts more than one. The protocol subject legitimately has two interested parties — the application and an optional `TracingCanProtocol*Observer` (§13) — whereas a category has one consumer. `infra::Subject` selects its single- or multi-observer specialisation from the observer's `SingleHelper` typedef, so the two forms differ only in the base class; `NotifyObservers()` and `Attach()`/`Detach()` are identical. - **Auto-attach/detach**: The observer attaches in its constructor and detaches in its destructor — no manual registration needed. - **Zero-cost when unobserved**: `NotifyObservers()` checks for a null observer pointer (single) or iterates an empty list (multiple) before dispatching, making it safe to call with no observer attached.

Example — an application category (server side):

```
// Observer interface (pure virtual callbacks for each command)
class MyCategoryServerObserver
    : public infra::SingleObserver<MyCategoryServerObserver, MyCategoryServer>
{
public:
    virtual void OnSetParameters(int16_t first, const infra::Function<void()>&
     ↪  onDone) = 0;
};

// Subject (the category handler)
```

```
class MyCategoryServer
    : public CanCategoryServer
    , public infra::Subject<MyCategoryServerObserver>
{ ... };

// Application attaches by constructing an observer
class MyHandler : public MyCategoryServerObserver
{
public:
    explicit MyHandler(MyCategoryServer& server)
        : MyCategoryServerObserver(server) {}

    void OnSetParameters(int16_t first, const infra::Function<void()>& onDone)
     ↪  override { ... }
};
```

6. System Category Design

The System category (ID `0x00`) is a **built-in** category that handles protocol-level concerns: heartbeat, status request, command acknowledgement, and category discovery. It is split into server and client variants.

Server side: `CanSystemCategoryServer`

The system category on the server is **fully automatic** — no public API is exposed to the application developer. `CanProtocolServer` creates an internal observer that reacts to system messages:

Message	Internal behavior
Heartbeat received	Notifies `CanProtocolServerObserver::Online()`
Status request	Sends heartbeat response
Category list request	Sends list of registered category IDs

`RegisterCategory()` returns `false`, without registering, if the category's ID is already registered on that server or if `canMaxRegisteredCategories` (8) categories are already registered. The application interacts with `CanProtocolServer` only through `RegisterCategory()` and the `CanProtocolServerObserver` (Online/Offline). All system plumbing is hidden.

Client side: `CanSystemCategoryClient`

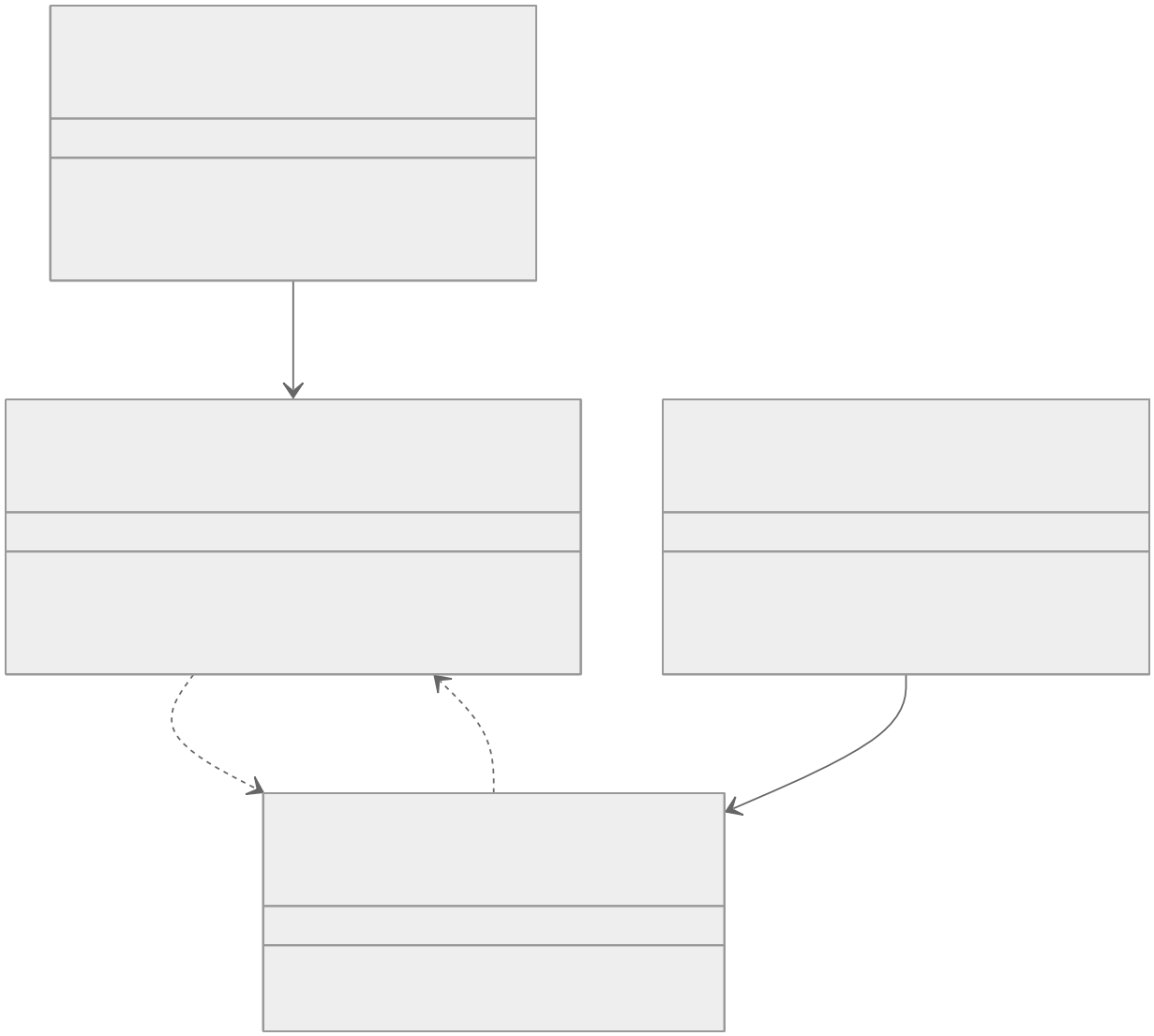
The system category on the client exposes **only category discovery** through its observer:

Exposed via observer	Description
`OnCategoryListResponse(categoryIds)`	Notifies when a category list is received from a server

Command acknowledgement handling (§9.4) happens in `CanProtocolClient` itself, not in `CanSystemCategoryClient`, because it needs the source node ID, which the category dispatch layer does not pass down to individual message handlers. Category discovery is exposed because the application may want to enumerate a server's capabilities.

7. Bidirectional Category Pattern

Each application category is split into two classes that mirror the client-server protocol:



- **Server category**: Registers handlers for **commands** (IDs `0x00`–`0x7F`). Uses the inherited `SendResponse()` / `SendTelemetry()` / `SendCategoryError()` helpers, which fill in the category ID and priority.
- **Client category**: Registers handlers for **responses** (IDs `0x80`–`0xFF`). Uses the inherited `SendCommand(uint16_t targetNodeId, ...)` helper, which directs each frame to a specific server and prepends the per-server sequence byte.

Server categories take a `CanFrameTransport&` in their constructor. Client categories take a `CanFrameTransport&` and a `CanSequenceSource&`; `CanProtocolClient` implements the latter, so categories depend only on `can_lite.core`.

See Extending can-lite with categories (Chapter 14) for the full authoring guide.

8. CAN Identifier Layout

All 29 bits of the extended CAN ID encode routing information:

```
[28:24] Priority     (5 bits)  — message urgency
[23:20] Category     (4 bits)  — functional group (0x0 = System)
[19:12] Message Type (8 bits)  — specific command/response within category
[11:0]  Node ID      (12 bits) — target server address (0x000 = broadcast)
```

Priority values (lower = higher priority on the CAN bus):

Priority	Value	Usage
Emergency	0	Reserved for safety-critical messages
Command	4	Client-to-server commands
Response	8	Server-to-client responses
Telemetry	12	Periodic data streams
Heartbeat	16	Presence detection

9. Sequence Validation

Server-side categories validate an 8-bit sequence number in `data[0]` of every command frame:

- The server tracks the last accepted sequence number.
- A command is accepted if its sequence number equals `(previous + 1) mod 256`.
- On sequence error, the server sends a `CommandAck` with `sequenceError` status.
- The sequence counter wraps from 255 → 0.

Client-side categories skip sequence validation (responses are stateless).

The server keeps **one** counter, shared by all sequence-validated categories. This is deliberate: the supported topology is one client to many servers, and a server serves exactly one client (REQ-CAN-006.1). Two clients commanding the same server concurrently interleave their counters and are rejected with `sequenceError`.

9.1 Per-Server Sequence Tracking

`CanProtocolClient` maintains **independent sequence counters per server node** in a fixed-size array (`maxServers = 8`) and exposes them through the `CanSequenceSource` interface. `PeekSequence(nodeId)` returns the next sequence byte for the given node, and `CommitSequence(nodeId, category, messageType)` advances it once the frame has been accepted by the send queue — so a rejected frame does not burn a sequence number. This ensures that commands directed to different servers do not share or interfere with each other's replay protection state. `CommitSequence` also

starts that server's command-ack timeout (§9.4), which is why it takes the category and message type of the command just sent.

Sequence Resynchronization

A `sequenceError` acknowledgement carries the sequence number the server expected (§9.4). `CanProtocolClient` parses every acknowledgement frame as it arrives (before category dispatch, since it still has the source node ID at that point) and, on `sequenceError`, sets that server's counter to the reported value. This recovers a client whose sequence state has drifted from the server's — for example after the client process restarts and its in-memory counter resets to 0 while the server, unaware of the restart, still expects its old counter to continue — without requiring the server to also reset.

9.2 Server Liveness Detection

`CanProtocolClient` detects server online/offline transitions:

- Every received frame with a non-broadcast source node ID is passed to `MarkServerAlive(nodeId)`.
- On first reception from a node, `CanProtocolClientObserver::OnServerOnline(nodeId)` is called.
- A `TimerSingleShot` (configurable, default 3 s) is (re)started on each frame. If no frame is received before the timer fires, `CanProtocolClientObserver::OnServerOffline(nodeId)` is called.
- Up to 8 server liveness slots are tracked simultaneously.

Applications connect a `CanProtocolClientObserver` to receive these events and react accordingly (e.g., stop issuing commands, alert the UI).

9.2.1 Client Liveness Detection (Server-side)

`CanProtocolServer` detects the reverse direction: whether its client is still present. `CanProtocolClient` broadcasts its own heartbeat (node ID `0x000`) following the same quiet-period rule as the server's heartbeat (§9.3); since that heartbeat is deferred by any other outgoing traffic, a client sending commands continuously might not emit a dedicated heartbeat frame for a long time. `CanProtocolServer` therefore restarts its `clientLivenessTimer` (configurable, default 3 s via `Config::clientTimeout`) on any frame correctly addressed to it, not only heartbeats — mirroring how `CanProtocolClient::MarkServerAlive` treats any received frame as proof of a server's liveness (§9.2). Only a received heartbeat notifies `CanProtocolServerObserver::Online()`, since that is the specific, meaningful signal; if the timer fires without further traffic from the client, `CanProtocolServerObserver::Offline()` is notified. A server tracks liveness for one client only, consistent with the single sequence counter (§9).

9.3 Heartbeat Timer (Silence Guard)

Both `CanProtocolServer` and `CanProtocolClient` use a `TimerSingleShot` instead of a `TimerRepeating` for heartbeat emission. The timer is restarted (`ResetHeartbeatTimer()`) after every outgoing frame on that side. This means a heartbeat is only sent when that side has been **silent**

for the full heartbeat interval, preventing unnecessary heartbeat traffic on active buses. The client's heartbeat is a broadcast, since a client has no node ID of its own on the bus.

9.4 Command Acknowledgement Timeout

When `CanCategoryClient::SendCommand` sends a sequence-validated command, `CanProtocolClient::CommitSequence` starts a per-server `ackTimer` (configurable, default 1 s via `Config::commandAckTimeout`) alongside the category and message type of the command sent. Any acknowledgement frame from that server matching that (category, messageType) cancels the timer, regardless of its status. If the timer fires first, `CanProtocolClientObserver::OnCommandAckTimeout(nodeId, category, messageType)` is notified; the command is not automatically retried. Because a server can have at most one outstanding command tracked this way, sending a second sequence-validated command to the same server before the first is acknowledged replaces the tracked (category, messageType) — an acknowledgement for the first command that arrives afterward will not match and will not cancel the timer for the second.

10. Directory Structure

```
can-lite/
|-- core/                          # Protocol primitives
|   |-- CanCategory.hpp/cpp        # Base class + Server/Client subclasses
|   |-- CanMessageType.hpp         # Message handler interface
|   |-- CanMessageHandler.hpp      # Binds a message type ID to a member function
|   |-- CanPayload.hpp/cpp         # Bounds-checked big-endian payload reader/writer
|   |-- CanSequenceSource.hpp      # Per-server sequence supply for client categories
|   |-- CanProtocolDefinitions.hpp # CAN ID layout, constants, enums
|   |-- CanFrameCodec.hpp/cpp      # Fixed-point encoding helpers
|   |-- CanFrameTransport.hpp/cpp  # Async send queue over hal::Can
|   |-- test/                      # Unit tests + can_lite.test_util doubles
|-- categories/
|   |-- system/                    # Built-in System category (0x00)
|   |   |-- CanSystemCategoryServer.hpp/cpp
|   |   |-- CanSystemCategoryClient.hpp/cpp
|   |-- firmware_upgrade/          # Firmware Upgrade category (0x01)
|       |-- FirmwareUpgradeDefinitions.hpp
|       |-- FirmwareUpgradeCategoryServer.hpp/cpp
|       |-- FirmwareUpgradeCategoryClient.hpp/cpp
|       |-- test/
|-- transport/                      # ISO-TP segmentation layer
|   |-- IsoTpTransport.hpp         # Abstract interface
|   |-- IsoTpTransportImpl.hpp/cpp # Non-template concrete impl (WithStorage)
|   |-- iso-tp/                    # ISO 15765-2 internals
|       |-- IsoTpChannel.hpp       # Non-template abstract channel interface
|     |-- IsoTpChannelImpl.hpp/cpp # Non-template concrete channel (WithStorage<MaxPduSize>)
|      |-- IsoTpSender.hpp/cpp    # Non-template transmit FSM (WithStorage<MaxPduSize>)
|      |-- IsoTpReceiver.hpp/cpp  # Non-template receive FSM (WithStorage<MaxPduSize>)
|       |-- IsoTpFrameCodec.hpp/cpp # PCI encoding/decoding
```

```
|      |-- IsoTpTypes.hpp       # Enums, constants (FrameType, FlowStatus, AbortReason)
|-- server/                        # CanProtocolServer
|   |-- CanProtocolServer.hpp/cpp
|   |-- test/
|-- client/                        # CanProtocolClient
|   |-- CanProtocolClient.hpp/cpp
|   |-- test/
|-- tracing/                       # Optional tracing decorators (§13)
|   |-- TracingCan.hpp/cpp                        # hal::Can decorator
|   |-- TracingIsoTpTransport.hpp/cpp             # IsoTpTransport decorator
|   |-- TracingCanProtocolServerObserver.hpp/cpp  # protocol-layer events
|   |-- TracingCanProtocolClientObserver.hpp/cpp
|   |-- test/
|-- drivers/                       # Hardware driver adapters

examples/                          # Not built by default (CAN_LITE_BUILD_EXAMPLES)
|-- foc_motor/                     # Reference application category
```

Application categories live in the consuming project, not in `can-lite/categories/`.

10.1 ISO-TP Transport Layer

The transport layer provides multi-frame PDU segmentation and reassembly following ISO 15765-2 without coupling it to any specific application category. It is an optional, orthogonal mechanism: categories that need large payloads attach `IsoTpTransportImpl` to the server/client; categories that fit in 8 bytes continue using raw `CanFrameTransport` directly.

Key classes:

Class	Role
`IsoTpTransport`	Abstract interface — `RegisterReceiveChannel`, `SendPdu`, `ProcessFrame`, `SetOnPduReceived`
`IsoTpTransportImpl`	Non-template concrete implementation; channel pool via `WithStorage<MaxPduSize, MaxChannels>`
`IsoTpChannel`	Non-template abstract channel interface used by `IsoTpTransportImpl`
`IsoTpChannelImpl`	Non-template concrete channel; composes `IsoTpSender` + `IsoTpReceiver` via `WithStorage<MaxPduSize>`
`IsoTpSender`	Non-template transmit FSM (SF → FF → wait-for-FC → CFs; N_Bs timeout); `WithStorage<MaxPduSize>` provides buffer
`IsoTpReceiver`	Non-template receive FSM (SF dispatch or FF → wait-for-CFs; N_Cr timeout); `WithStorage<MaxPduSize>` provides buffer
`IsoTpFrameCodec`	Stateless PCI encode/decode helpers

WithStorage pattern — zero-heap construction chain:

All ISO-TP classes are non-template at their API surface. Sizes are injected via nested `WithStorage` type aliases that use EMIL's `infra::WithStorage` to compose storage ownership:

```
// IsoTpSender takes BoundedVector<uint8_t>& — WithStorage provides the buffer
IsoTpSender::WithStorage<MaxPduSize>  // IS-A IsoTpSender

// IsoTpReceiver — same pattern
IsoTpReceiver::WithStorage<MaxPduSize>  // IS-A IsoTpReceiver

// IsoTpChannelImpl composes sender + receiver storage in a nested struct
IsoTpChannelImpl::WithStorage<MaxPduSize>  // IS-A IsoTpChannelImpl IS-A
 ↪  IsoTpChannel

// IsoTpTransportImpl composes a BoundedVector of channels
IsoTpTransportImpl::WithStorage<MaxPduSize, MaxChannels>  // IS-A IsoTpTransportImpl
```

Attachment pattern:

```
IsoTpTransportImpl::WithStorage<64, 4> isoTp{ canFrameTransport };
protocolServer.AttachIsoTpTransport(isoTp);
```

`CanProtocolServer::ProcessReceivedMessage` offers each incoming frame to the ISO-TP layer first (`isoTpTransport->ProcessFrame(canId, frame)`). If the transport claims it (a registered channel matches), normal category dispatch is skipped. This keeps the transport layer transparent to existing category handlers.

11. Build System

- **CMake presets** are the primary interface (see `CMakePresets.json`).
- Libraries follow the `can_lite.<component>` naming convention.
- Category libraries link `can_lite.core`; protocol libraries link the relevant category libraries.
- Unit tests use GoogleTest/GMock and run on the host.
- Standalone builds fetch `embedded-infra-lib` via FetchContent.

12. Integration Testing

Integration tests validate end-to-end behavior across components using cucumber-cpp-runner v4.0.0 (BDD / Gherkin). Feature files are in `integration_tests/features/`; step definitions in `integration_tests/steps/`.

Single Common Fixture

All scenarios share a single fixture type — `ApplicationFixture` — that simulates a real application with a server, client, and virtual CAN bus. This ensures tests exercise the same initialization and interaction paths as production code.

```
integration_tests/
|-- features/                    # Gherkin .feature files
|-- hooks/                       # Scenario lifecycle hooks
|-- steps/                       # Step definitions (GIVEN/WHEN/THEN)
|-- support/
    |-- ApplicationFixture.hpp    # VirtualCan, ApplicationFixture, mocks
```

VirtualCan

`VirtualCan` is a concrete `hal::Can` implementation that replaces hardware drivers in integration tests. Two `VirtualCan` instances are connected via `ConnectTo()`: frames sent by one are delivered to the other's receive callback, simulating a shared CAN bus without mocking `SendData`/`ReceiveData`. `InjectFrame()` allows direct frame injection for testing error paths.

ApplicationFixture

`ApplicationFixture` inherits from `infra::ClockFixture` (providing `EventDispatcher`, `TimerService`, and `ForwardTime()`) and composes:

- A pair of connected `VirtualCan` instances (server-side and client-side).
- `CanProtocolServer` and `CanProtocolClient` wired to their respective CAN interfaces.
- `StrictMock` observers for the server and (optionally) the demo category.
- Optional demo category components (`CanFrameTransport`, `DemoCategoryServer`/`Client`, observers) activated via `RegisterDemoCategory()`. The demo category (`0x3`) is the reference application category used to validate the extension API end to end.
- Optional Firmware Upgrade components (`FirmwareUpgradeCategoryServer`/`Client`, observers) activated via `RegisterFirmwareUpgrade()`.
- Dynamic test categories (`SequencedTestCategory`, `SimpleTestCategory`) for sequence and discovery testing.

StrictMock Everywhere

All mock observers use `testing::StrictMock`. Unexpected calls cause immediate test failure, ensuring every interaction is explicitly expected.

Cucumber Context API

- `context.Emplace<T>(args...)` creates the fixture (returns `std::shared_ptr<T>`).
- `context.Get<T>()` retrieves it by reference (returns `T&`).
- Captured `shared_ptr` in lambdas on fixture members must be avoided to prevent circular references that leak the fixture across scenarios.

13. Observability

Tracing is provided by **decorators the consumer opts into in the composition root**. Nothing in the library holds a `services::Tracer&`; a build that never constructs a decorator pays nothing. Threading a tracer through `CanFrameTransport`, the protocol classes and every category was rejected: it would change every constructor signature and add a dependency to code that currently links `can_lite.core` only. `.github/linters/goodcheck.yml` enforces the same conclusion by forbidding `services::GlobalTracer()` in production code.

Layers and seams

Layer	Seam	Class
HAL	`hal::Can`	`TracingCan`

Layer	Seam	Class
Transport	`IsoTpTransport`	`TracingIsoTpTransport`
Protocol	`CanProtocolServerObserver` / `CanProtocolClientObserver`	`TracingCanProtocolServerObserver` / `TracingCanProtocolClientObserver`

`TracingCan` sees **all** traffic in both directions, including frames dropped later by node-ID filtering, rate limiting, or unregistered categories. The protocol observers cover what the frame log structurally cannot show: `OnCommandAckTimeout` and both `Offline` transitions are timer-driven and put nothing on the bus, so without them a quiet bus and a bus whose acknowledgements are being missed look identical.

The category layer has no decorator. `CanCategory::HandleMessage` and `HandlePduMessage` are non-virtual, so decorating a category would mean making dispatch virtual in core — for information already derivable from the frame trace and the decoded `commandAck` line.

Wiring

```
services::TracingCan tracingCan{ realCan, tracer };
services::CanProtocolServer server{ tracingCan, config };

services::TracingIsoTpTransport tracingIsoTp{ isoTp, tracer };
server.AttachIsoTpTransport(tracingIsoTp);

services::TracingCanProtocolServerObserver serverTrace{ server, tracer };
```

Line format

Every line is `<ClassName>: <message>`, so a mixed log stays attributable to a layer and filters with a single `grep Tracing`:

```
TracingCan: TX id 0x4001123 prio command cat 0x0 system type 0x1 heartbeat node 0x123 dlc 3 data 010203
TracingCan: ack cat 0x3 type 0x5 status sequenceError expectedSeq 0x7
TracingIsoTpTransport: Abort dataId 0x18db33f1 reason nCrTimeout
TracingCanProtocolClientObserver: CommandAckTimeout node 0x123 cat 0x3 type 0x5
```

Targets

One target per layer, so a consumer links only what it instruments: `can_lite.tracing` (core), `can_lite.tracing_iso_tp` (transport), `can_lite.tracing_protocol` (server + client). All three additionally link `services.tracer`.

`services::Tracer::Trace()` returns `infra::TextOutputStream` under `EMIL_ENABLE_TRACING` and a no-op type under `EMIL_DISABLE_TRACING`, so every trace line is written as a single chained expression — the result is never stored, and `Continue()` is never used to append a conditional part.

Chapter 14

Extending can-lite with Categories

Status: Living document

can-lite ships two categories — System (`0x0`) and Firmware Upgrade (`0x1`). Everything domain-specific belongs in the consuming project as an **application category** using identifiers `0x2`–`0xF`. This document is the reference for writing one.

A complete worked example lives in `examples/foc_motor/`; a minimal one used by the integration tests lives in `integration_tests/support/TestCategories.hpp`.

1. What a category is

A category is a **server/client pair** that shares one 4-bit category ID:

Side	Base class	Owns	Message type range
Server	`CanCategoryServer`	Command handlers, `Send*Response()` methods	`0x00`–`0x7F`
Client	`CanCategoryClient`	Response handlers, `Send*Command()` methods	`0x80`–`0xFF`

The two base classes carry distinct `infra::IntrusiveList` node types, so the compiler prevents registering a client category on a server and vice versa.

Only `can_lite.core` is needed to write a category — do not link `can_lite.server` or `can_lite.client`.

2. Define the identifiers

Put every wire constant in one `*Definitions.hpp` so the wire format is reviewable in a single place.

```
#pragma once

#include <cstdint>
```

```
namespace services
{
    static constexpr uint8_t myCategoryId = 0x03;

    // Commands (client -> server), 0x00-0x7F
    static constexpr uint8_t mySetParametersId = 0x01;
    static constexpr uint8_t myQueryValueId = 0x02;

    // Responses (server -> client), 0x80-0xFF
    static constexpr uint8_t myValueResponseId = 0x82;

    enum class MyError : uint8_t
    {
        busy = 0,
        notSupported = 1
    };
}
```

`IsApplicationCategoryId()`, `IsCommandMessageType()` and `IsResponseMessageType()` from `CanProtocolDefinitions.hpp` are `constexpr`, so these can be asserted at compile time:

```
static_assert(IsApplicationCategoryId(myCategoryId));
static_assert(IsCommandMessageType(mySetParametersId));
static_assert(IsResponseMessageType(myValueResponseId));
```

3. Bind handlers with `CanMessageHandler`

`CanMessageHandler<Owner>` binds a message type ID to a member function. It stores an ID, a reference and a member pointer — no allocation, no vtable per message, and no nested class per message type.

```
class MyCategoryServer
    : public CanCategoryServer
    , public infra::Subject<MyCategoryServerObserver>
{
public:
    explicit MyCategoryServer(CanFrameTransport& transport);

    uint8_t Id() const override;

private:
    void HandleSetParameters(const hal::Can::Message& data);
    void HandleQueryValue(const hal::Can::Message& data);

    CanMessageHandler<MyCategoryServer> setParameters{ mySetParametersId, *this,
    ↪ &MyCategoryServer::HandleSetParameters };
    CanMessageHandler<MyCategoryServer> queryValue{ myQueryValueId, *this,
    ↪ &MyCategoryServer::HandleQueryValue };
```

```
};
```

Register them once in the constructor:

```
MyCategoryServer::MyCategoryServer(CanFrameTransport& transport)
    : CanCategoryServer(transport)
{
    AddMessageTypes(setParameters, queryValue);
}
```

For a message that also arrives as a reassembled ISO-TP PDU, pass a second member function: `CanMessageHandler<Owner>{ id, *this, &Owner::HandleFrame, &Owner::HandlePdu }`.

4. Read and write payloads with `CanPayload`

`CanPayloadReader` and `CanPayloadWriter` are cursor-based and big-endian, so no byte offsets are computed by hand. Both track validity as a **sticky flag**: one check covers the whole payload.

```
void MyCategoryServer::HandleSetParameters(const hal::Can::Message& data)
{
    CanPayloadReader reader{ data };
    reader.Skip(1);                        // sequence byte
    auto first = reader.ReadInt16();
    auto second = reader.ReadInt16();

    if (!reader.Valid())
    {
        SendCommandAck(mySetParametersId, CanAckStatus::invalidPayload);
        return;
    }

    // ...
}
```

Reads past the end return zero and clear `Valid()`; writes past 8 bytes are dropped and clear `Valid()`. `SendResponse()` and `SendCommand()` refuse to send an invalid payload, so overflow can never reach the bus.

Available: `ReadUInt8`, `ReadInt16`, `ReadUInt16`, `ReadInt32`, `ReadUInt32`, `ReadFixed16(scale)`, `Skip`, `ReadRemaining`, `Available` — and the matching `Write*` methods plus `WriteBytes`.

5. Send frames

The base classes own the transport and fill in the category ID and priority.

Server:

Method	Priority	Message type
`SendResponse(messageType, payload)`	`response`	as given
`SendTelemetry(messageType, payload)`	`telemetry`	as given

Method	Priority	Message type
`SendCategoryError(originatingCommandId, code)`	`response`	`0xFE`
`SendCommandAck(messageType, status)`	`response`	system ACK

Client:

Method	Sequence byte
`SendCommand(nodeId, messageType[, payload][, priority])`	prepended automatically
`SendCommandWithoutSequence(nodeId, messageType[, payload][, priority])`	none

```
bool MyCategoryClient::SendSetParameters(uint16_t targetNodeId, int16_t first,
 ↪  int16_t second)
{
    CanPayloadWriter payload;
    payload.WriteInt16(first).WriteInt16(second);

    return SendCommand(targetNodeId, mySetParametersId, payload);
}
```

Do **not** write the sequence byte yourself — `SendCommand()` prepends it and only advances the counter once the frame is accepted by the send queue.

Use `SendCommandWithoutSequence()` only when the matching server category overrides `RequiresSequenceValidation()` to return `false` (as Firmware Upgrade does, because block indices already order the transfer). `priority` defaults to `CanPriority::command`; pass `CanPriority::emergency` for safety-critical commands.

6. Report errors

Two mechanisms, used together:

- **`CanAckStatus`** — the universal outcome every command reports (`success`, `invalidPayload`, `invalidState`, `rateLimited`, …).
- **Category error response (`0xFE`)** — a category-defined error code when the universal status is not specific enough. Payload is `{ originatingCommandId, categoryErrorCode }`.

```
SendCategoryError(myFailingCommandId, static_cast<uint8_t>(MyError::busy));
SendCommandAck(myFailingCommandId, CanAckStatus::categoryError);
```

Client categories observe it by binding a handler to `canCategoryErrorResponseMessageTypeId`.

7. Expose events through an observer

Every category notifies its consumer through `infra::Subject` / `infra::SingleObserver`. Asynchronous work is handed back through an `infra::Function` completion callback so handlers never block.

```
class MyCategoryServerObserver
    : public infra::SingleObserver<MyCategoryServerObserver, MyCategoryServer>
{
public:
    using infra::SingleObserver<MyCategoryServerObserver,
     ↪ MyCategoryServer>::SingleObserver;

    virtual void OnSetParameters(int16_t first, int16_t second, const
     ↪ infra::Function<void()>& onDone) = 0;
};
```

Observer callbacks must not allocate and must not block.

8. Register

```
CanProtocolServer server{ can, config };
MyCategoryServer myCategory{ server.Transport() };
server.RegisterCategory(myCategory);

CanProtocolClient client{ can };
MyCategoryClient myCategoryClient{ client.Transport(), client };
client.RegisterCategory(myCategoryClient);
```

`CanProtocolClient` implements `CanSequenceSource`, which is all a client category needs — categories never depend on `CanProtocolClient` itself.

Always use `client.Transport()` for a category's `CanFrameTransport&`, rather than constructing a second, independent `CanFrameTransport` over the same `hal::Can`. Each `CanFrameTransport` owns its own `sendInProgress` flag and send queue; two independent transports wrapping the same `hal::Can` would each believe they alone own the outstanding transmission, and concurrent sends from both could overlap on the HAL.

`RegisterCategory()` returns `false`, without registering, if the category's ID is already registered or if 8 categories (`canMaxRegisteredCategories`) are already registered on that server or client — check the return value if registration can plausibly fail in your composition. Registered category IDs are reported automatically by category discovery.

9. Test

Unit tests use GoogleTest with `testing::StrictMock<>` only. `can_lite.test_util` provides `hal::CanMock`, plus `CanCategoryServerStub` / `CanCategoryClientStub`, which own a CAN mock, a `CanFrameTransport` and a `CanSequenceSource` so a test category stays default-constructible.

Cover at minimum, per message type: the happy path, a short payload, and — for server categories — an out-of-sequence command.

Chapter 15

can-lite: Protocol Specification

Version: 1.0
Protocol Version Byte: 1
Status: Draft
Date: 2026

1. Abstract

This document specifies the can-lite CAN bus protocol, a lightweight client-server communication protocol operating over CAN 2.0B (29-bit extended identifiers) at up to 1 Mbit/s. The protocol provides a minimal built-in System category with heartbeat, command acknowledgement, status request, and category discovery messages. Applications extend the protocol by registering custom category handlers on the server.

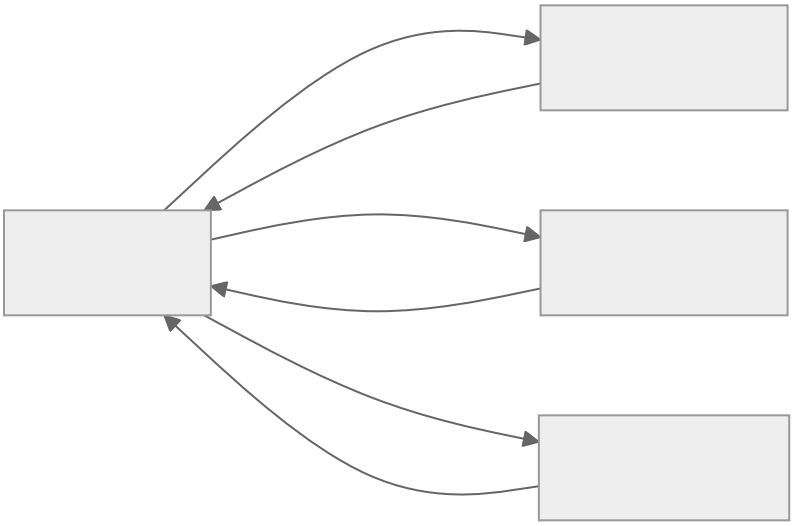
2. Terminology

Term	Definition
Server	A node on the CAN bus that listens for commands, processes them, and sends responses. Each server has a unique node ID and serves exactly one client.
Client	The initiator of all commands and queries. A single client can communicate with multiple servers.
Broadcast	A message addressed to all servers (node ID 0x000)
Category	A 4-bit field in the CAN identifier that groups related message types. The built-in System category (0x0) is always available; applications register additional categories.
Category Handler	A `CanCategoryServer` or `CanCategoryClient` implementation registered via `RegisterCategory()` that processes all messages for a specific category.
Sequence Number	An 8-bit counter in byte[0] of command frames for replay protection
Scale Factor	Integer multiplier used to convert floats to fixed-point integers

3. Architecture

The protocol follows a **client-server** model:

- The **client** is the sole initiator of commands and queries. It can address multiple servers on the same CAN bus by targeting different node addresses. The client maintains independent sequence counters per server.
- The **server** passively listens for incoming frames addressed to its node ID (or the broadcast address). It processes commands, dispatches them to the appropriate category handler, and sends acknowledgement responses.
- The topology is **one client to many servers**. A server serves exactly one client and keeps a single sequence counter; addressing one server from two clients concurrently is not supported and is rejected with a sequence error.
- Both the server and the client automatically transmit heartbeat messages starting from construction. Each side uses the presence or absence of the other's heartbeats to determine whether its peer is **online** or **offline**: the client notifies the application via `CanProtocolClientObserver` (`OnServerOnline(nodeId)` / `OnServerOffline(nodeId)`), and the server via `CanProtocolServerObserver` (`Online()` / `Offline()`).



4. Transport

- Physical layer: CAN 2.0B
- Bit rate: up to 1 Mbit/s (configurable)
- Identifier format: 29-bit extended only; 11-bit frames are silently discarded
- Maximum payload: 8 bytes per frame (CAN 2.0 standard)

4.1 ISO-TP Segmentation (ISO 15765-2)

For payloads exceeding the 8-byte CAN frame limit, can-lite provides an optional ISO-TP transport layer (`IsoTpTransportImpl`) that implements ISO 15765-2 segmentation and reassembly.

Frame Types

PCI Nibble	Frame Type	Description
`0x0N`	Single Frame (SF)	N = data length (1–7); entire PDU fits in one frame
`0x1NNN`	First Frame (FF)	NNN = total PDU length (8–4095); first 6 bytes
`0x2N`	Consecutive Frame	N = sequence number 0–F (wraps); up to 7 bytes
`0x3S`	Flow Control (FC)	S = status: 0=CTS, 1=Wait, 2=Overflow

Flow Control Fields

Byte	Field	Description
0	PCI	`0x3S` — Flow Status (0=CTS, 1=Wait, 2=Overflow)
1	BS	Block Size — number of CFs before next FC (0 = unlimited)
2	STmin	Minimum separation time: 0x00–0x7F = 0–127 ms; 0xF1–0xF9 = 100–900 μs (ISO-TP sub-ms range); 0x80–0xF0 and 0xFA–0xFF = reserved (treated as 0x7F / 127 ms per ISO 15765-2)

Timing Parameters

Parameter	Value	Description
N_Bs	1000 ms	Sender timeout waiting for a Flow Control frame
N_Cr	1000 ms	Receiver timeout waiting for a Consecutive Frame
N_WFTmax	16	Maximum consecutive Flow Control `Wait` frames before the sender aborts (implementation-defined per ISO 15765-2)

On `FS = Wait`, the sender restarts N_Bs and continues waiting rather than aborting immediately; only `N_WFTmax` consecutive `Wait` frames (or an N_Bs timeout) abort the transfer.

Integration

`IsoTpTransportImpl` is attached to `CanProtocolServer` or `CanProtocolClient` via `AttachIsoTpTransport(IsoTpTransport&)`. When attached, incoming frames are first offered to the ISO-TP layer; if no registered channel claims the frame, it falls through to normal category dispatch. PDUs reassembled by the transport layer are delivered via the `SetOnPduReceived` callback. Channels are reclaimed via `ReleaseChannel(dataId)`, called automatically on abort and available for the application to call once it is done with a given `dataId`.

The implementation uses `WithStorage` for zero-heap construction. All internal components (`IsoTpSender`, `IsoTpReceiver`, `IsoTpChannelImpl`) are non-template classes that receive their PDU buffer storage via `infra::WithStorage` aliases, following the EMIL convention:

```
IsoTpTransportImpl::WithStorage<64, 4> isoTp{ canFrameTransport };
protocolServer.AttachIsoTpTransport(isoTp);
```

```
isoTp.RegisterReceiveChannel(dataId, fcId);
```

Addressing constraint. `IsoTpTransportImpl` routes frames to channels purely by the literal CAN ID carried in `dataId`/`fcId` — the sender's identity is not otherwise encoded. `AttachIsoTpTransport()` alone does not register any channel; the application (or a category built on top of it) must explicitly call `RegisterReceiveChannel()` for each `dataId`/`fcId` pair it expects multi-frame traffic on, using IDs that unambiguously identify a single correspondent (e.g. a dedicated node-to-node exchange such as a firmware upload session). A `dataId` shared by more than one concurrent sender at a time will have their frames interleaved into the same channel; the transport layer does not detect or reject this. Because of this, can-lite does not provide automatic multi-frame support for arbitrary category command/response traffic across many nodes — only the point-to-point case with an explicitly-managed channel is supported.

5. CAN Identifier Layout

All 29 bits of the extended CAN ID are structured as follows:

```
Bit:  28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9  8  7  6  5  4  3  2  1  0
     |----  Priority  ----|-- Category --|------  Message Type  ------|-------------
---- Node ID -------------------|
    |    5 bits (0-31)  |  4 bits (0-F)|      8 bits (0-FF)      |          12 bits (0-
FFF)                  |
```

Field Encoding:

```
raw_id = (priority << 24) | (category << 20) | (message_type << 12) | node_id
```

6. Priority Levels

Value	Name	Usage
0	Emergency	Safety-critical events
4	Command	Commands, parameter writes
8	Response	Command acknowledgements
12	Telemetry	Periodic measurements
16	Heartbeat	Node liveness

Lower numerical values have higher CAN bus arbitration priority.

7. Message Categories

Value	Name	Description
0x0	System	Heartbeat, command acknowledgement, status request, category discovery
0x1	Firmware Upgrade	Block-based firmware transfer, verification, and activation
0x2 - 0xF	Application	Reserved for application-defined categories

The System category is always available. Category 0x1 is defined in a separate extension specification:

- Firmware Upgrade (Chapter 16)

Applications register additional categories (values 0x2-0xF) by providing `CanCategoryServer` and `CanCategoryClient` implementations to `CanProtocolServer::RegisterCategory()` and `CanProtocolClient::RegisterCategory()`. Both return `false`, without registering, if the category's ID is already registered or if 8 categories are already registered (the System category always occupies one of those 8 slots). See Extending can-lite with categories (Chapter 14).

7.1 Reserved Message Type

Message type `0xFE` is reserved across all categories for a **category error response**, carrying a category-specific failure detail that the universal acknowledgement status cannot express.

Byte	Field	Type	Description
0	Originating Command	uint8	Message type of the command that failed
1	Error Code	uint8	Category-defined error code

A category error response is normally followed by a command acknowledgement with status `categoryError`.

8. Message Catalog

8.1 System (Category 0x0)

8.1.1 Heartbeat (Type 0x01)

Sent at CanPriority::heartbeat. No sequence validation.

Byte	Field	Type	Description
0	Version	uint8	Protocol version (currently 1)

Both the server and the client automatically transmit heartbeat messages starting from the moment they are constructed. A heartbeat is sent **at least 1 second after the last transmitted message** of any kind, by whichever side is sending it. This ensures liveness detection without adding traffic when that side is already actively communicating. The heartbeat acts as a keep-alive: it fires only

during periods of silence. The client's heartbeat is broadcast (node ID 0x000), since the client has no node ID of its own on the bus.

The client uses received heartbeats to track server liveness. When a heartbeat (or any message) is received from a server, the client considers that server **online** and resets that server's timeout timer. If no messages are received from a server within a configurable timeout (default 3 s), the client considers it **offline** and notifies the application via `CanProtocolClientObserver::OnServerOffline(nodeId)`. Multiple servers (up to 8) can be tracked simultaneously with independent liveness timers.

Symmetrically, the server uses received traffic to track whether its client is still present. Any frame correctly addressed to the server (re)starts the server's client timeout timer (default 3 s, `Config::clientTimeout`), the same "any message counts" rule the client uses for server liveness; a heartbeat specifically also notifies the application via `CanProtocolServerObserver::Online()`. If the timeout timer expires without further traffic from the client, the server notifies `CanProtocolServerObserver::Offline()`. A server tracks only one client, per REQ-CAN-006.1.

8.1.2 Command Acknowledgement (Type 0x02)

Sent by the server at CanPriority::response.

Byte	Field	Type	Description
0	Category	uint8	CanCategory of the acknowledged command
1	Command	uint8	CanMessageType of the acknowledged command
2	Status	uint8	See acknowledgement status table
3	Expected Sequence	uint8	Meaningful only when Status is Sequence Error (4); 0 otherwise

The category byte ensures the client can uniquely identify which command is being acknowledged, since message type values may be reused across categories.

On receiving a `sequenceError` acknowledgement, the client adopts byte 3 as its next sequence number for that server, so that a single lost frame, or the client's own sequence state having reset (for example after a restart), does not leave the two sides permanently out of sync — see §11.

If no acknowledgement of any status arrives for a sequence-validated command within a configurable timeout (default 1 s, `Config::commandAckTimeout`), the client notifies the application via `CanProtocolClientObserver::OnCommandAckTimeout(nodeId, category, messageType)`. The command itself is not retried; the application decides whether and how to retry.

8.1.3 Status Request (Type 0x03)

Sent by the client at CanPriority::command. No sequence validation. Empty payload.

When received, the server responds by transmitting a heartbeat message. This allows the client to quickly confirm the server is online without waiting for the next scheduled heartbeat.

8.1.4 Category List Request (Type 0x04)

Sent by the client at CanPriority::command. No sequence validation. Empty payload.

When received, the server responds with a Category List Response message containing the IDs of all registered category handlers.

8.1.5 Category List Response (Type 0x05)

Sent by the server at CanPriority::response.

Byte	Field	Type	Description
0	Category 0	uint8	First registered category ID
1	Category 1	uint8	Second registered category ID
…	…	uint8	Additional category IDs (up to 8 max)

Each byte contains the ID of one registered category handler. The System category (0x0) is always included. Categories are listed in registration order. The response is limited to 8 category IDs by the CAN frame payload size. `RegisterCategory()` enforces this bound at registration time: the 9th and any subsequent registration attempt returns `false` and the category is not registered, so the response can never truncate.

9. Data Encoding

All multi-byte integers are encoded **big-endian** (network byte order).

9.1 Encoding Algorithm

```
fixed_value = clamp(round(float_value × scale_factor), INT_MIN, INT_MAX)
float_value = fixed_value / scale_factor
```

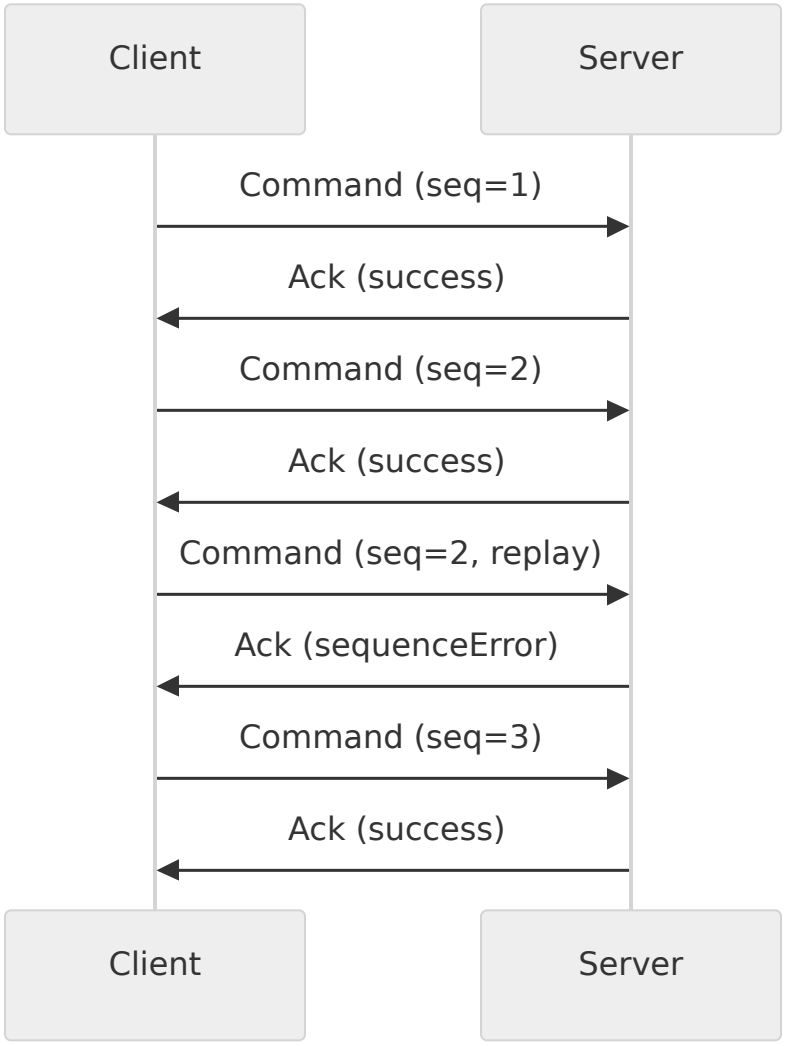
Values are saturated (clamped) to the target integer range to prevent overflow.

10. Enumeration Tables

10.1 Acknowledgement Status

Value	Status	Description
0	Success	Command accepted and processed
1	Unknown Command	Message type not recognized for category
2	Invalid Payload	Payload too short or field out of range
3	Invalid State	Command not valid in current state
4	Sequence Error	Sequence number not (previous + 1) mod 256
5	Rate Limited	Message rate limit exceeded
6	Not Implemented	Command recognized but handler not implemented
7	Category Error	Category-specific rejection; details in category 0xFE frame

11. Sequence Number Protocol

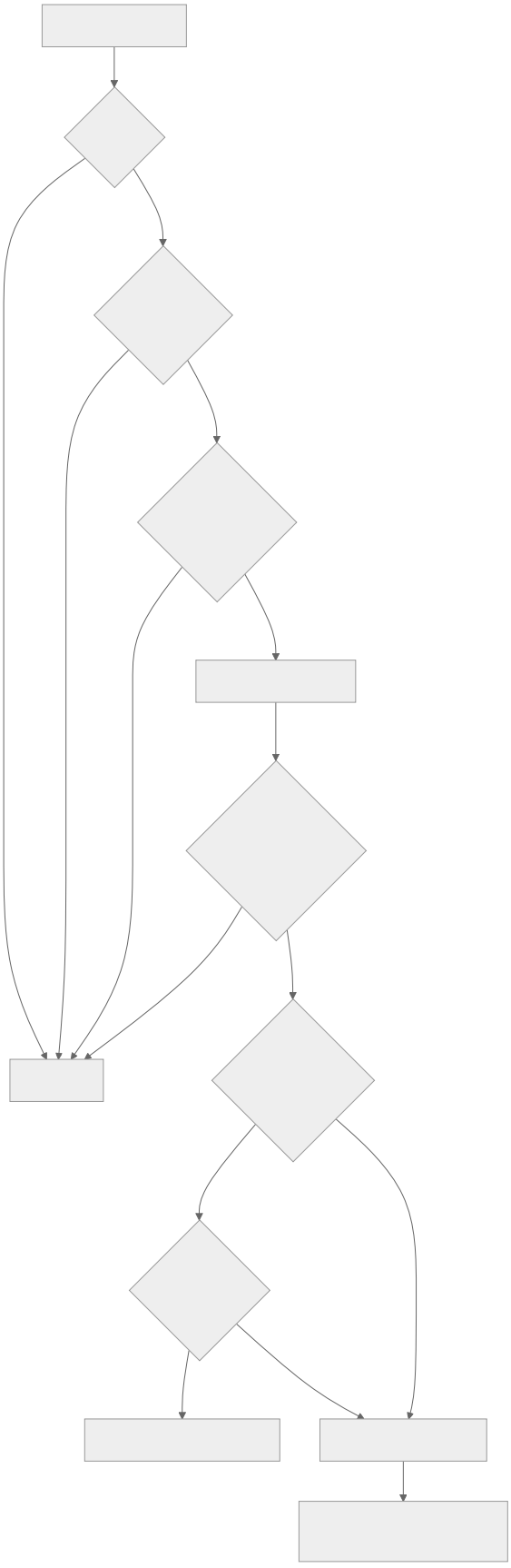


- For command frames that carry a payload, Byte[0] is an unsigned 8-bit sequence counter used for best-effort in-order processing.
- The server accepts the first sequenced command received regardless of sequence value and records it as the reference.
- Each subsequent command should have sequence = (previous + 1) mod 256.
- Out-of-order or duplicated commands are rejected with a `sequenceError` acknowledgement.
- Individual category handlers declare whether they require sequence validation via `RequiresSequenceValidation()`. Categories that opt out bypass validation entirely.
- When sequence validation is required and the payload is empty (no sequence byte present), the frame is rejected with `invalidPayload`.
- Heartbeat and status request frames do not use sequence numbers.
- Unrecognized message types within a registered category are rejected with an `unknownCommand` acknowledgement.
- A `sequenceError` acknowledgement carries the sequence number the server expected (§8.1.2); the client adopts it as its next sequence number for that server. This recovers a client whose

own sequence state has drifted from the server's (for example, after the client process restarts and its in-memory counter resets to 0 while the server, unaware of the restart, still expects its old counter to continue) without requiring the server to also restart.
- Sequence numbers are not an authentication or security mechanism and MUST NOT be relied upon to prevent malicious replay on an untrusted CAN bus.

12. Rate Limiting

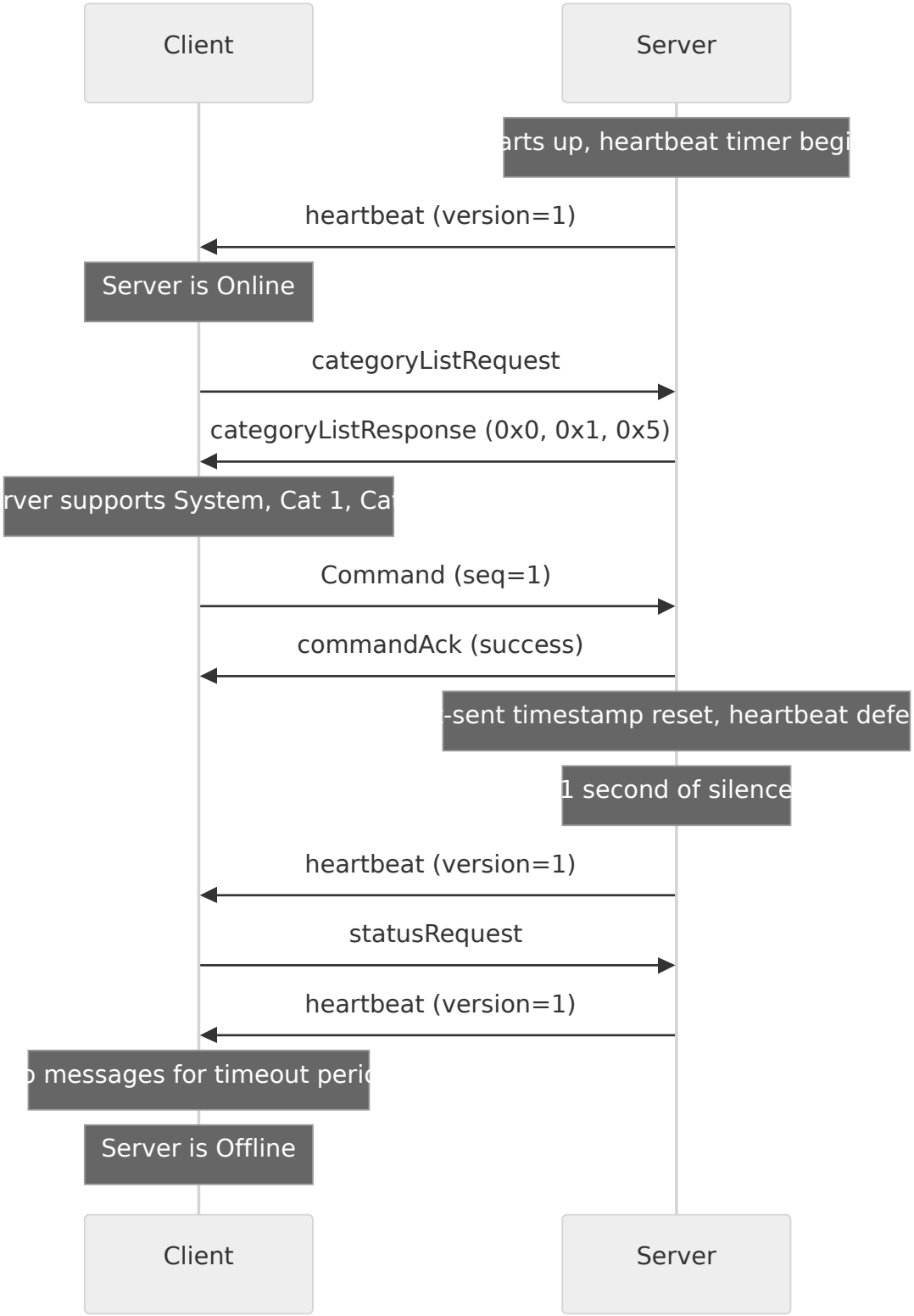
The server enforces a configurable maximum message rate (default: 500 messages per period). Messages received after the limit is reached are silently discarded. The rate counter resets automatically every second via an internal timer.



13. Node Addressing

- Each server has a unique 12-bit node ID (1–4095) set at configuration time.
- Node ID 0x000 is reserved as the broadcast address.
- Frames addressed to the broadcast ID are accepted by all servers.
- Frames addressed to a different node ID are silently discarded.

14. Typical Command Flow



15. Extensibility

Applications extend can-lite by defining additional categories:

1. **Define an enum value** for the new category (0x2–0xF; 0x1 is reserved for Firmware Upgrade).
2. **Implement `CanCategoryServer` and `CanCategoryClient`** subclasses that handle the message types within that category.
3. **Register the handler** with the server at construction time.

The server dispatches incoming frames to the matching category handler. Frames with unregistered categories are silently discarded. Frames with a registered category but an unknown message type receive an `unknownCommand` acknowledgement.

16. Security Considerations

- **In-order delivery / duplicate detection:** Sequence number validation rejects out-of-order or duplicated commands. This is a best-effort ordering check, not a security mechanism — see §11's caveat: sequence numbers are transmitted in the clear and MUST NOT be relied upon to prevent malicious replay on an untrusted CAN bus.
- **Bus flooding protection:** Configurable rate limiting discards excess messages.
- **Input validation:** All payloads are length-checked before parsing. Enum-valued fields are decoded via unchecked `static_cast` from the wire byte; an out-of-range value produces a scoped-enum value with no matching enumerator (well-defined per the C++ standard, since these enums have a fixed underlying type) rather than a rejected frame — it is not currently validated against the enum's defined range.
- **No heap allocation:** Fixed-size buffers prevent memory exhaustion.
- **Node isolation:** Strict node ID filtering prevents cross-node interference.

Chapter 16

Firmware Upgrade — Category Extension Specification

Category ID: 0x1
Extends: can-protocol.md (Chapter 15)
Version: 1.0
Status: Draft
Date: 2026

1. Overview

This document specifies the Firmware Upgrade category extension for the can-lite protocol. It defines a block-transfer protocol for sending firmware images from a client to a server over CAN, with integrity verification and activation control.

The category does **not** require protocol-level sequence validation (`RequiresSequenceValidation() = false`). Ordering is enforced by explicit block indices within the data transfer messages, and the protocol-level 8-bit sequence counter would wrap too quickly for large transfers (thousands of blocks).

1.1 Design Goals

- **Reliability:** Per-block acknowledgement and CRC32 verification before activation.
- **Resumability:** Block-indexed transfer allows the client to detect and retransmit missing blocks.
- **Safety:** New firmware is verified in-place before activation; the running firmware is never overwritten in-flight.
- **Simplicity:** No multi-frame segmentation; each CAN frame is self-contained.

1.2 Assumptions

- The server has a dual-bank (A/B) flash layout or a dedicated staging area where incoming firmware is written without overwriting the active image.
- The server's bootloader or application can swap to the newly written image upon activation.

- Maximum firmware size per transfer: **393 216 bytes** (65 536 blocks × 6 data bytes). For larger images, a page-based extension can be added in a future revision.

2. Upgrade States

Value	State	Description
0	Idle	No upgrade in progress
1	Receiving	Accepting data blocks
2	Verifying	CRC32 check in progress
3	Complete	Verification passed, ready to activate
4	Error	Transfer or verification failed

3. Error Codes

Value	Error	Description
0	Ok	No error
1	Busy	Upgrade already in progress
2	InvalidSize	Firmware size exceeds available storage
3	SequenceError	Block index out of order or duplicate
4	WriteError	Flash write/erase failure
5	CrcMismatch	CRC32 verification failed
6	NotReady	Activate requested before verify passed
7	InvalidState	Command not valid in current upgrade state
8	SessionTimeout	No activity within the session timeout

4. Message Types — Commands (Client → Server)

All commands are sent at `CanPriority::command`.

4.1 Begin Upgrade (0x00)

Initiate a firmware upgrade session. If an upgrade is already in progress, the server responds with error code `Busy`.

Byte	Field	Type	Description
0–3	FirmwareSize	uint32	Total firmware image size in bytes

Total: 4 bytes.

The server erases the staging area (if applicable), transitions to Receiving state, and responds with a Begin Upgrade Response (0x80).

4.2 Data Block (0x01)

Transfer a block of firmware data.

Byte	Field	Type	Description
0–1	BlockIndex	uint16	Zero-based block sequence number
2–7	Data	uint8[6]	Firmware data (up to 6 bytes)

Total: 2–8 bytes. The last block may contain fewer than 6 data bytes.

Blocks must be sent in ascending order starting from 0. The server responds with a Data Block Ack (0x81) for each received block.

4.3 Verify (0x02)

Request CRC32 verification of the received firmware image against the expected checksum.

Byte	Field	Type	Description
0–3	CRC32	uint32	Expected CRC32 of the complete firmware

Total: 4 bytes.

The server transitions to Verifying state, computes the CRC32 over the received data, and responds with a Verify Response (0x82).

4.4 Activate (0x03)

Instruct the server to switch to the newly uploaded firmware. This command is only valid after a successful Verify (state = Complete).

Empty payload.

The server responds with an Activate Response (0x83), then performs the bank swap or reboot. The server may go offline briefly during this process.

4.5 Abort (0x04)

Cancel the current upgrade session and discard received data. Valid in any state except Idle.

Empty payload.

The server transitions to Idle state and responds with a command acknowledgement (System category, success).

4.6 Query Progress (0x05)

Request the current state of the upgrade process.

Empty payload.

The server responds with a Progress Response (0x85).

4.7 Session Timeout

If the server does not receive a DataBlock, Verify, Activate, or Abort command within a configurable inactivity period (default 30 s) after a BeginUpgrade or the last DataBlock, the server automatically aborts the upgrade session and transitions to Idle state. The application is notified via the `OnSessionTimeout()` observer callback.

QueryProgress does not reset the session timer.

The session timeout is configured per-category-instance via `FirmwareUpgradeCategoryServer::Config::sessionTimeout`.

5. Message Types — Responses (Server → Client)

Response message type IDs follow the `0x80 + command_id` convention. All responses are sent at `CanPriority::response`.

5.1 Begin Upgrade Response (0x80)

Byte	Field	Type	Description
0	Status	uint8	Error code (see Section 3)
1–2	PageSize	uint16	Flash erase page size in bytes

Total: 3 bytes.

`PageSize` informs the client of the server's flash granularity. This can be used for transfer optimization but is not required.

5.2 Data Block Ack (0x81)

Byte	Field	Type	Description
0	Status	uint8	Error code (see Section 3)
1–2	BlockIndex	uint16	Index of the acknowledged block

Total: 3 bytes.

5.3 Verify Response (0x82)

Byte	Field	Type	Description
0	Status	uint8	0 = CRC match, 5 = CRC mismatch

Total: 1 byte.

On success, the server transitions to Complete state.

5.4 Activate Response (0x83)

Byte	Field	Type	Description
0	Status	uint8	0 = activating, 6 = not ready

Total: 1 byte.

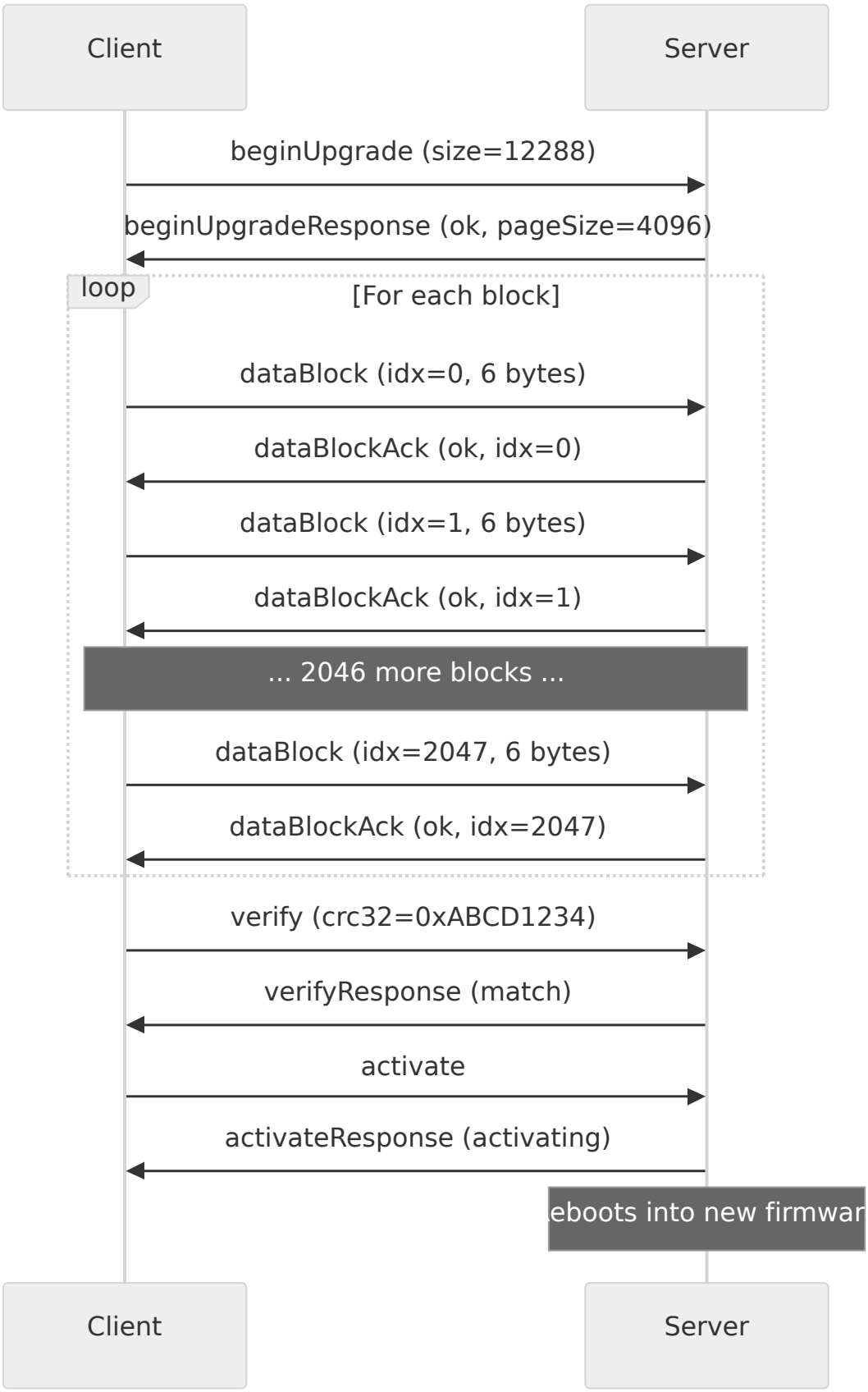
5.5 Progress Response (0x85)

Byte	Field	Type	Description
0	State	uint8	Upgrade state (see Section 2)
1–2	BlocksReceived	uint16	Number of blocks received so far
3–4	TotalBlocks	uint16	Total blocks expected

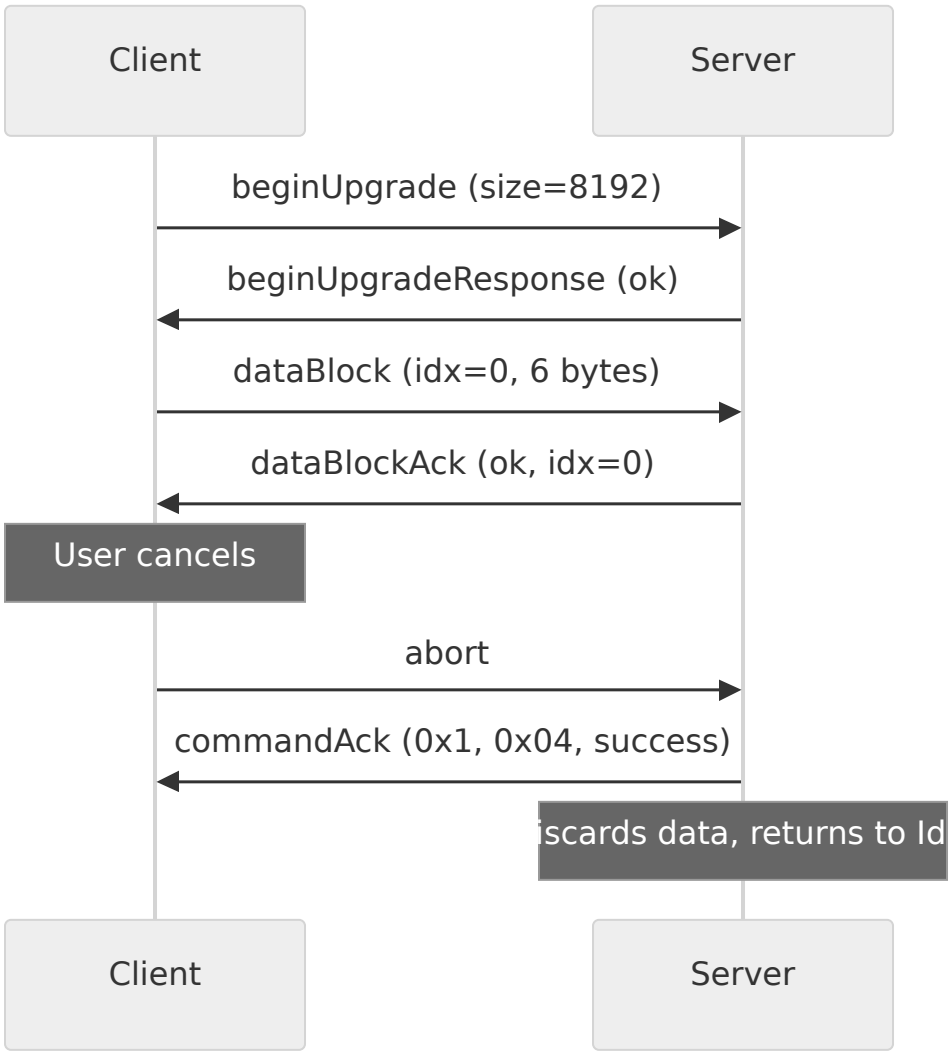
Total: 5 bytes.

6. Typical Flow

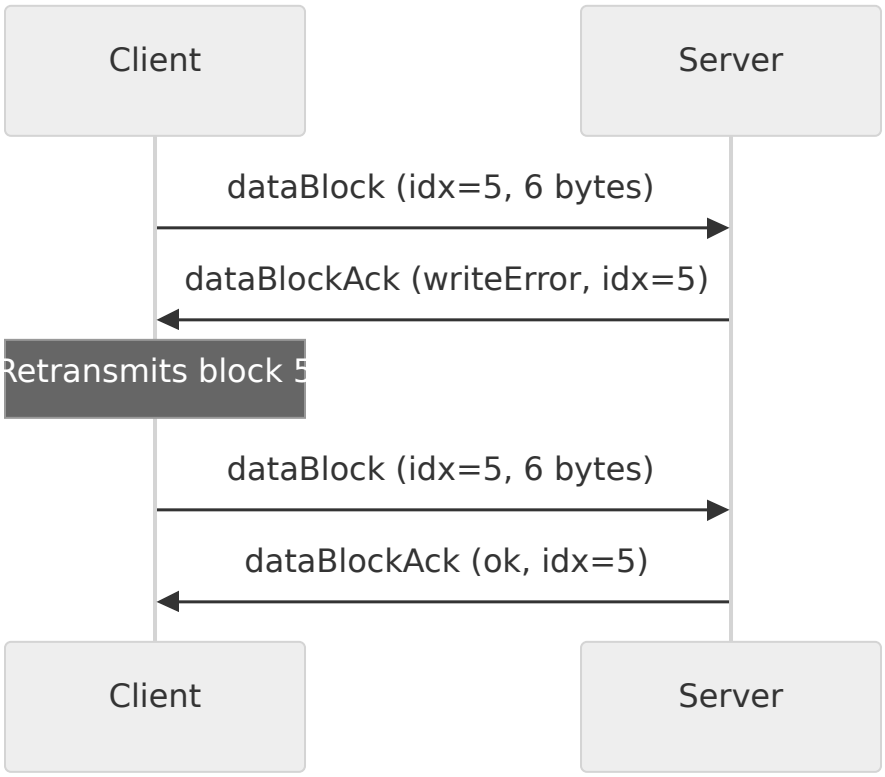
6.1 Successful Upgrade



6.2 Aborted Upgrade



6.3 Failed Block Write



7. Implementation Considerations

7.1 Flash Layout

A dual-bank (A/B) layout is recommended:

```
+------------------+------------------+
|  Bank A (Active)  |  Bank B (Staging) |
|  Running firmware |  Incoming image   |
+------------------+------------------+
```

The incoming firmware is written to the inactive bank. On activation, the bootloader swaps the active bank pointer and reboots. If the new firmware fails to start (watchdog timeout), the bootloader reverts to the previous bank.

7.2 Transfer Rate

At 1 Mbit/s with 6 data bytes per CAN frame (~120 bits per frame including framing overhead), the theoretical maximum throughput is approximately **6 KB/s** (accounting for ack frames). Faster rates can be achieved by batching multiple data blocks before requiring acks (window-based acknowledgement), which is a possible future extension.

7.3 Security

For production deployments, consider:

- **Signature verification:** Validate a cryptographic signature (e.g., ECDSA-P256) appended to the firmware image before activation. The signature can be sent in a separate message type or included in the last data blocks.
- **Encryption:** If firmware confidentiality is required, encrypt the image before transfer and decrypt on the server after verification.
- **Rollback prevention:** Maintain a monotonic version counter in non-volatile storage; reject firmware with a lower version number.

These security extensions are out of scope for this initial specification but should be implemented for any deployment where the CAN bus is accessible to untrusted parties.

7.4 Large Firmware Images

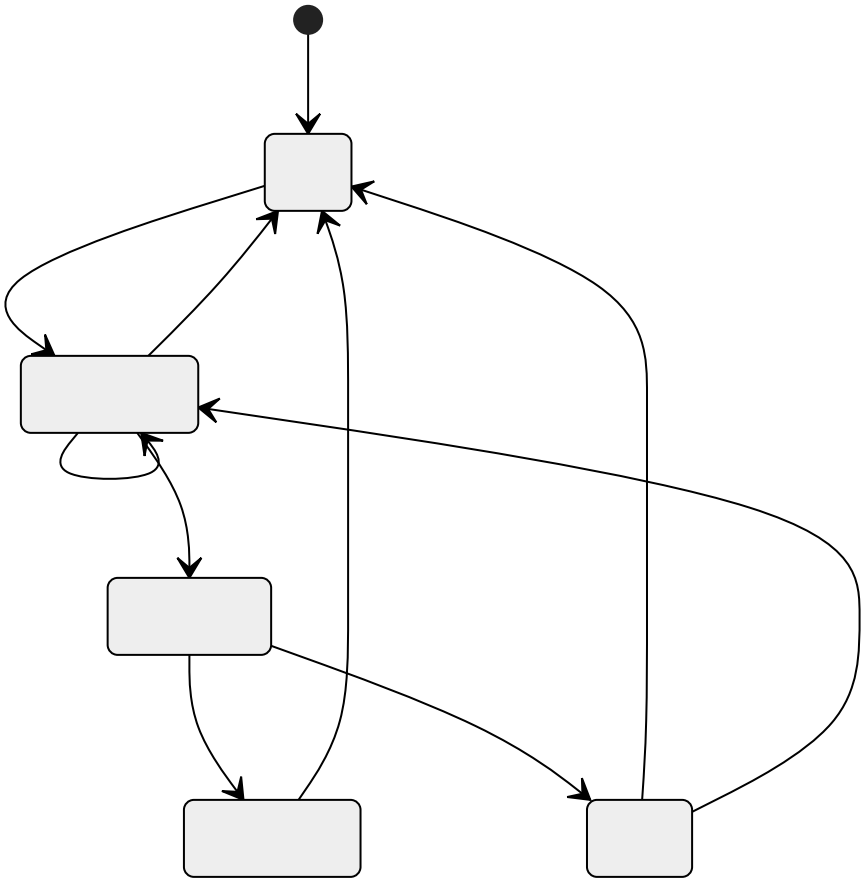
The uint16 block index limits transfers to ~384 KB. For larger images, two extension strategies are available:

1. **Set Page (new message type):** A "set page" command selects the upper bits of the address, and block indices are offsets within the current page. This allows addressing up to 4 GB with negligible overhead.
2. **Extended Data Block:** A new message type using uint32 block indices, reducing data payload to 4 bytes per frame (256 MB max).

7.5 Error Recovery

- If a block ack indicates `SequenceError`, the client should retransmit the expected block.
- If a block ack indicates `WriteError`, the client may retry the same block a limited number of times before aborting.
- If the server stops responding, the client should send a Query Progress to determine the current state and resume from the last acknowledged block.

7.6 State Machine



Part V

Appendices

Chapter 17

Glossary

Protocol vocabulary — node, category, command, response, priority, sequence number, PDU and the rest — is defined in the Protocol Specification (Chapter 15) §2, and the constants and enumerations are tabulated in §5, §6 and §10 of the same document. Segmentation frame types and timing parameters are in §4.1.

This glossary covers only the vocabulary this booklet adds: the implementation patterns the design is built from, and the terms used to classify behaviour.

Design and implementation vocabulary

Term	Meaning
Binding	The association of one message identifier with one member function of the category that handles it. The unit of dispatch inside a category (Chapter 5)
Bounded container	A container with a compile-time maximum, used everywhere a growing container would be. Reaching the maximum is a defined outcome, never an allocation
Channel	A pair of identifiers — one for data, one for the flow control that answers it — with one sender and one receiver (Chapter 8)
Completion	A function handed to an observer so it can report the outcome later, without the notifying component holding state across the gap (Chapter 9)
Design limit	A deliberate simplification whose cost is stated rather than hidden. Used as a marker throughout Chapter 10
EMIL	embedded-infra-lib, the infrastructure library supplying the bounded containers, timers, callbacks and observer pattern
Eviction	Reclaiming the least recently claimed slot of a full fixed-size table, in rotation, rather than refusing the newcomer (Chapter 7)
Latent	Behaviour that is correct today but depends on an assumption a driver or an integrator could break. Used as a marker throughout Chapter 10
Peek and commit	Taking the next sequence number and advancing the counter as two steps, so a refused send does not consume a number (Chapter 7)
Reference, not ownership	The composition rule: components borrow their collaborators, and the application owns everything (Chapter 2)

Term	Meaning
Silence guard	A timer restarted by activity, so the action it guards happens only after real inactivity. How both heartbeats work (Chapter 6)
Sticky validity	Payload composition and consumption poison themselves on the first out-of-range access, so a caller checks once rather than per field (Chapter 4)
Tumbling window	A counting window that resets wholesale on a timer, as opposed to a sliding one. Why a burst can cross a boundary at twice the limit (Chapter 10, §7.2)
`WithStorage`	The pattern in which an implementation is non-template and borrows its storage, while a nested alias owns that storage and carries the size. Described in Architecture and Design Decisions (Chapter 13) §10.1

Roles

Term	Meaning
Acknowledger	The role, played by the server protocol object, through which a category has an acknowledgement sent on its behalf — because an acknowledgement belongs to the system category, and no category speaks for another (Chapter 5)
Frame transport	The single outbound queue per node, owned by the protocol object and lent to every category (Chapter 4)
Sequence source	The role, played by the client protocol object, through which a client category obtains sequence numbers without depending on the client library (Chapter 5)
Observer	The application-side implementation of a category's or protocol object's event interface. Exactly one per subject, bound for its own lifetime

Chapter 18

Appendix A — Requirements Catalogue

Generated from `documents/requirements/*.yaml` at build time. These are the normative requirements the design in this booklet implements; each section corresponds to one requirement file.

Can Protocol

ID	Title	Shall
`REQ-CAN-001`	CAN bus transport	The protocol shall use the CAN bus as the communication channel between clients and servers. All protocol interactions follow a client-server model where the client initiates requests and the server processes them.
`REQ-CAN-002`	Extended CAN identifiers	The protocol shall use exclusively 29-bit extended CAN identifiers for all transmitted and received frames. Both client and server shall silently discard any frame using an 11-bit standard identifier.
`REQ-CAN-003`	CAN identifier structure	The 29-bit CAN identifier shall encode four fields: a 5-bit priority field (bits 28-24), a 4-bit category field (bits 23-20), an 8-bit message type field (bits 19-12), and a 12-bit node address field (bits 11-0).

ID	Title	Shall
REQ-CAN-004	Client-server model	The protocol shall follow a client-server architecture where the client is the sole initiator of commands and queries, and the server listens for incoming frames, processes them, and sends responses. The server shall not initiate unsolicited commands to the client.
REQ-CAN-005	Server node identity	Each server shall be configured with a unique 12-bit node address (1-4095) that identifies it on the CAN bus. The server shall process only frames addressed to its own node address or to the broadcast address (0x000).
REQ-CAN-006	Multi-server client support	The client shall support communication with multiple servers on the same CAN bus by addressing frames to different server node addresses. The client shall maintain independent sequence counters per server.
REQ-CAN-006.1	Single-client server	Each server shall serve exactly one client. The server shall maintain a single sequence counter shared by all sequence-validated categories, so concurrent command streams originating from more than one client are not supported and shall be rejected with a sequence error. The relationship is therefore one client to many servers, never many clients to one server.
REQ-CAN-007	Broadcast message support	The protocol shall reserve node address 0x000 as the broadcast address. All servers shall accept frames addressed to 0x000 regardless of their configured node address, enabling simultaneous control of multiple servers.

ID	Title	Shall
REQ-CAN-008	Default system category	The protocol shall define a built-in System category (0x0) that provides heartbeat, command acknowledgement, status request, and category discovery messages. This category shall be available without any application-specific extensions.
REQ-CAN-009	Server heartbeat mechanism	The server shall automatically begin transmitting heartbeat frames upon startup. A heartbeat frame shall contain a 1-byte protocol version identifier and shall be sent at the lowest CAN priority level. The server shall transmit a heartbeat at least 1 second after the last message of any kind was sent, acting as a keep-alive during periods of silence. The heartbeat shall not be sent while the server is actively transmitting other messages within the 1-second window.
REQ-CAN-009.1	Client online/offline detection	The client shall use received heartbeat frames (and other messages) to determine whether a server is online or offline. When a heartbeat is received from a server, the client shall consider that server online and notify the application layer. When no messages are received from a server within a configurable timeout, the client shall consider that server offline and notify the application layer.
REQ-CAN-010	Protocol version identification	The heartbeat frame shall carry a protocol version byte that allows the client to verify wire-format compatibility with the server.

ID	Title	Shall
REQ-CAN-011	Command acknowledgement	The server shall transmit a 4-byte acknowledgement frame for every processed command, containing the original command's category, message type, a status code indicating success or the specific error condition (unknown command, invalid payload, invalid state, or sequence error), and an expected-sequence byte (REQ-CAN-037). Messages rejected by rate limiting are the exception: per REQ-CAN-022 they are silently discarded and receive no acknowledgement, since sending one would itself add traffic during the overload condition rate limiting exists to shed.
REQ-CAN-012	Status request	The client shall be able to send a status request message to a server. Upon receiving this message, the server shall respond by transmitting a heartbeat message, allowing the client to quickly confirm the server is online without waiting for the next scheduled heartbeat.
REQ-CAN-013	Category-based message dispatch	The protocol shall dispatch received frames to handlers based on the category field in the CAN identifier. Each category handler shall be independently registered and shall process only its own message types.

ID	Title	Shall
REQ-CAN-014	Application-defined categories	The protocol shall allow applications to define additional message categories beyond the built-in System and Firmware Upgrade categories by registering custom category handlers on the server and client. Category identifiers 0x2 to 0xF shall be available to applications. RegisterCategory() shall reject, by returning false without registering, an attempt to register a category whose ID is already registered, or that would exceed canMaxRegisteredCategories (8) categories registered at once on that server or client. This bound exists because the category discovery response (REQ-CAN-008) reports every registered category ID in a single 8-byte frame; it must never truncate.
REQ-CAN-015	Fixed-point data encoding	All floating-point values transmitted over CAN shall be encoded as big-endian signed fixed-point integers with defined per-quantity scale factors, with saturation clamping to prevent integer overflow.
REQ-CAN-016	Short payload rejection	The server shall reject any command frame whose payload is shorter than the minimum required length for the given message type, and shall respond with an invalid-payload error without applying any state changes.

ID	Title	Shall
REQ-CAN-017	Sequence number validation	The server shall validate command frames using an 8-bit sequence counter in the first payload byte. The first received command shall be accepted regardless of its sequence value. Subsequent commands shall be accepted only if their sequence number equals the previous sequence number plus one, modulo 256. Out-of-sequence commands shall be rejected with a sequence error response.
REQ-CAN-018	Sequence number wrap-around	The sequence counter shall wrap from 255 to 0 without error, allowing continuous operation without sequence resets.
REQ-CAN-019	Sequence bypass per category	Individual category handlers shall be able to declare whether they require sequence number validation. Categories or specific message types that do not require sequencing shall bypass validation entirely.
REQ-CAN-020	Unknown message category handling	The server shall silently discard any received frame whose category field does not correspond to a registered category handler, without generating any response.
REQ-CAN-021	Unknown message type handling	The server shall respond with an unknown-command error when a frame is received with a registered category but an unrecognized message type.
REQ-CAN-022	Message rate limiting	The server shall enforce a configurable maximum number of received messages per second. Messages exceeding this limit shall be silently discarded. The rate counter shall be reset automatically every second by an internal timer.
REQ-CAN-023	Node address filtering	The server shall process only CAN frames whose node address matches its configured local address or the broadcast address (0x000), and shall silently discard all other frames.

ID	Title	Shall
REQ-CAN-024	No dynamic memory allocation	The CAN protocol library shall not perform any dynamic memory allocation and shall use only statically-sized data structures.
REQ-CAN-025	Category discovery	The client shall be able to send a category list request message to a server. Upon receiving this message, the server shall respond with a category list response containing the IDs of all registered category handlers. The response payload shall list one category ID per byte, limited to the maximum CAN frame payload size (8 bytes). If more than 8 categories are registered, the server shall include only the first 8 in registration order and shall drop the remainder without an error indication; a server needing to expose more than 8 categories is not currently supported.
REQ-CAN-026	Per-server sequence counter on client	The client shall maintain independent 8-bit sequence counters for each server node it communicates with. The sequence counter for a node shall start at 0 on the first command and increment by 1 for each subsequent command to that same node. Counters for different nodes shall be independent. Up to 8 server nodes may be tracked simultaneously.
REQ-CAN-027	Server liveness detection	The client shall track server liveness independently for each server node. When any frame is received from a source node ID (non-zero), the client shall call OnServerOnline(nodeId) the first time and reset a per-server timeout timer (default 3 s). If no frame is received from a server within the timeout period, the client shall call OnServerOffline(nodeId). Up to 8 server liveness slots may be tracked simultaneously.

ID	Title	Shall
REQ-CAN-028	Send queue overflow indication	The CanFrameTransport::SendFrame() method shall return true if the frame was successfully sent or enqueued, and false if the internal send queue was full and the frame was dropped. The onDone callback shall not be called when false is returned.
REQ-CAN-029	ISO-TP segmentation support	The library shall provide an optional ISO 15765-2 (ISO-TP) transport layer that segments PDUs exceeding 8 bytes into multiple CAN frames and reassembles them at the receiver. The transport layer shall be attachable to CanProtocolServer and CanProtocolClient via AttachIsoTpTransport().
REQ-CAN-030	ISO-TP frame types	The ISO-TP implementation shall support all four ISO 15765-2 frame types: Single Frame (SF, PCI nibble 0x0), First Frame (FF, PCI nibble 0x1), Consecutive Frame (CF, PCI nibble 0x2), and Flow Control (FC, PCI nibble 0x3).
REQ-CAN-031	ISO-TP flow control	The sender shall wait for a Flow Control (ContinueToSend) frame from the receiver before transmitting Consecutive Frames. The receiver shall emit a Flow Control frame after receiving the First Frame. Block size (BS=0) and separation time (STmin) fields shall be honoured. On Flow Status = Wait, the sender shall restart the N_Bs timer and continue waiting rather than aborting, up to N_WFTmax (16) consecutive Wait frames, after which it shall abort. On Flow Status = Overflow, the sender shall abort immediately.

ID	Title	Shall
REQ-CAN-032	ISO-TP N_Bs timeout	The sender shall abort an in-progress multi-frame transmission and notify the abort callback with reason nBsTimeout if no Flow Control frame is received within 1000 ms of sending the First Frame or the last block of Consecutive Frames.
REQ-CAN-033	ISO-TP N_Cr timeout	The receiver shall abort reassembly and notify the abort callback with reason nCrTimeout if no Consecutive Frame is received within 1000 ms of the previous frame.
REQ-CAN-034	ISO-TP no dynamic memory allocation	The ISO-TP transport layer shall not perform any dynamic memory allocation. Channel state and PDU buffers shall use statically-sized storage provided at construction time via IsoTpTransportImpl::WithStorage.
REQ-CAN-035	ISO-TP channel management	IsoTpTransportImpl shall support registering multiple independent receive channels identified by a (dataId, fcId) pair. The maximum number of simultaneous channels shall be bounded at compile time via the MaxChannels template parameter (default 4). Registering a channel with a dataId already in use shall return false without modifying state. A channel shall be reclaimable via ReleaseChannel(dataId), allowing its slot to be reused once the application (or the transport itself, on abort) is done with that dataId.

ID	Title	Shall
REQ-CAN-036	Server-side client-liveness detection	The client shall automatically begin transmitting heartbeat frames upon construction, broadcast to node 0x000, following the same quiet-period keep-alive rule as the server's own heartbeat (REQ-CAN-009). The server shall restart its client-liveness timeout (default 3 s) on any frame correctly addressed to it, not only heartbeats, since a client sending commands continuously may defer its own heartbeat frame indefinitely; a received heartbeat shall additionally notify the application via CanProtocolServerObserver::Online(). If the timeout elapses without further traffic from the client, the server shall notify CanProtocolServerObserver::Offline(). A server tracks liveness for one client only, consistent with REQ-CAN-006.1.
REQ-CAN-037	Sequence resynchronization on sequence error	The sequenceError acknowledgement (REQ-CAN-011) shall carry, in its fourth byte, the sequence number the server expected. On receiving a sequenceError acknowledgement, the client shall adopt that value as its next sequence number for the acknowledging server, so that a single lost frame, or the client's own sequence state having reset independently of the server, does not leave the two sides unable to recover.

ID	Title	Shall
REQ-CAN-038	Command acknowledgement timeout	When the client sends a sequence-validated command, it shall start a timer (default 1 s, configurable). If no acknowledgement of any status arrives for that command before the timer elapses, the client shall notify the application via CanProtocolClientObserver::OnCommandAckTimeout(nodeId, category, messageType). Receiving a matching acknowledgement before the timer elapses shall cancel it. The client shall not automatically retry the timed-out command.

Firmware Upgrade

ID	Title	Shall
REQ-FWU-001	Upgrade session initiation	The client shall be able to initiate a firmware upgrade session by sending a Begin Upgrade command containing the total firmware size in bytes (uint32). The server shall reject the request if an upgrade is already in progress (Busy) or the size exceeds available storage (InvalidSize).
REQ-FWU-002	Upgrade session abort	The client shall be able to abort a firmware upgrade session at any time by sending an Abort command. The server shall discard any received data and return to the Idle state.
REQ-FWU-003	Block-based data transfer	Firmware data shall be transferred in blocks of up to 6 bytes per CAN frame, with each block identified by a uint16 block index (bytes 0–1). Blocks shall be transmitted in ascending order starting from index 0.

ID	Title	Shall
REQ-FWU-004	Per-block acknowledgement	The server shall acknowledge each received data block with a Data Block Ack containing the block index and a status code. The client shall not consider a block successfully transferred until the corresponding ack with Ok status is received.
REQ-FWU-005	Block write failure handling	If the server fails to write a block to flash storage, it shall respond with a WriteError status in the Data Block Ack. The client may retransmit the failed block.
REQ-FWU-006	CRC32 integrity verification	After all data blocks have been transferred, the client shall send a Verify command containing the expected CRC32 of the complete firmware image. The server shall compute the CRC32 over the received data and respond with match or mismatch status.
REQ-FWU-007	Firmware activation	The client shall be able to command the server to activate the newly uploaded firmware by sending an Activate command. This command shall only be accepted after a successful CRC32 verification (Complete state). The server shall perform the firmware swap and reboot.
REQ-FWU-008	Activation safety	The server shall reject Activate commands if the firmware has not been successfully verified (NotReady error), preventing activation of corrupt or incomplete images.
REQ-FWU-009	Upgrade progress query	The client shall be able to query the current upgrade state at any time. The server shall respond with the current state, the number of blocks received, and the total number of blocks expected.

ID	Title	Shall
REQ-FWU-010	Active firmware preservation	The firmware upgrade process shall not overwrite the currently running firmware. Incoming firmware shall be written to a staging area or inactive flash bank, preserving the active image until activation.
REQ-FWU-011	Upgrade state machine	The firmware upgrade process shall follow a defined state machine with states: Idle, Receiving, Verifying, Complete, and Error. Commands that are not valid in the current state shall be rejected with an InvalidState error.
REQ-FWU-012	Sequence validation bypass	The firmware upgrade category shall not require protocol-level sequence validation (RequiresSequenceValidation = false). Transfer ordering shall be enforced by the explicit block index in each Data Block message, since the protocol's 8-bit sequence counter is insufficient for the number of blocks in a typical firmware transfer.
REQ-FWU-013	Maximum firmware size	The firmware upgrade protocol shall support firmware images up to 393210 bytes (65536 blocks × 6 bytes per block), limited by the uint16 block index field. Transfers exceeding this limit shall be rejected during Begin Upgrade.
REQ-FWU-014	Response message type IDs	Response message types within the firmware upgrade category shall use IDs formed by adding 0x80 to the corresponding command message type ID, reserving 0x00–0x7F for commands and 0x80–0xFF for responses.

ID	Title	Shall
REQ-FWU-015	Session inactivity timeout	The server shall abort a firmware upgrade session if no DataBlock, Verify, Activate, or Abort command is received within a configurable inactivity timeout period (default 30 s) after BeginUpgrade or the last DataBlock. On timeout the server shall transition to Idle state and notify the application via the OnSessionTimeout() observer callback. QueryProgress commands shall not reset the inactivity timer.

can-lite

A zero-heap CAN 2.0B client-server protocol library

Built from revision *version* on *date*

https://github.com/embedded-pro/can-lite