Imbra Connect White Paper

Imbra Connect — White Paper

A Unified SDK for Industrial Protocol Connectivity

Author: Branimir Georgiev, Imbra.Soft **Version:** 0.1 — Draft **Date:** March 2026

Abstract

Industrial devices speak dozens of incompatible protocols. Engineers building connectivity layers spend weeks assembling fragmented, undermaintained libraries — and still end up with gaps. DeviceNet has no open Python library. EtherNet/IP lost its only maintained project. HART and Profibus are effectively unserved. The result is that every industrial connectivity project starts from scratch, reinventing the same plumbing with different tools.

Imbra Connect is a unified industrial protocol SDK that provides a single abstract interface across MQTT, Modbus, CAN, CANopen, DeviceNet, EtherNet/IP, OPC-UA, HART, and Profibus. One API, consistent across all protocols. Packet crafting and test tooling are first-class — engineers can validate implementations without physical hardware. Python and Go are fully open source (MIT). A community registry allows engineers to publish and discover protocol agents, growing coverage without Imbra.Soft writing every driver.

---

1. The Connectivity Problem

Modern industrial facilities are not built on a single protocol. A typical plant floor combines:

- PLCs communicating over Modbus TCP or EtherNet/IP
- Field sensors on HART or Profibus
- Motion controllers on CANopen
- SCADA systems on OPC-UA or legacy OPC DA
- Cloud and edge systems on MQTT

Each of these protocols has its own frame structure, addressing model, error handling, and timing constraints. An engineer building a data collection agent — to feed a historian, an MES, or an analytics platform — must implement or integrate all of them. There is no standard abstraction. There is no unified interface. Every project starts by assembling a stack from scratch.

This is not a new problem. It has persisted for decades because industrial protocols evolved independently, driven by competing vendor ecosystems rather than interoperability standards. OPC-UA was designed to solve it at the application layer, but it requires devices to support it — and the installed base of legacy Modbus, DeviceNet, and Profibus devices runs into the tens of millions globally and will not be replaced for decades.

The engineering cost is real. A conservative estimate for assembling a multi-protocol connectivity stack — selecting libraries, resolving version conflicts, writing abstraction layers, handling edge cases — is two to four weeks per project. Multiplied across the thousands of integration projects happening in industrial automation every year, the waste is enormous.

---

2. The State of the Python Ecosystem

Python is the dominant language for industrial integration scripts, historian agents, and connectivity prototypes. It has a broad ecosystem, fast iteration, and enough library coverage to get most protocols working — eventually. The problem is that the coverage is uneven, the maintenance is inconsistent, and the gaps are in exactly the protocols that matter most in legacy installations.

What exists

Protocol	Library	Stars	Status
MQTT	`paho-mqtt`	2.4k	□ Maintained — Eclipse Foundation
Modbus	`pymodbus`	2.7k	□ Maintained — very active, async-native
CAN	`python-can`	1.5k	□ Maintained — broad hardware support
CANopen	`canopen`	538	□ Maintained — built on python-can
OPC-UA	`asyncua`	1.4k	□ Maintained — still beta versioning
Siemens S7	`python-snap7`	778	□ Maintained — v3.0 pure Python rewrite

What is broken or missing

Protocol	Library	Status	Impact
EtherNet/IP	`pycomm3`	⚠ Abandoned — maintainer declared no new development	EtherNet/IP is the dominant protocol in Allen-Bradley and Rockwell installations — the most widely deployed PLC protocol in North America
DeviceNet	—	✗ No library exists	DeviceNet is installed in millions of devices globally — no open Python implementation has ever existed
HART	`hart-protocol`	⚠⚠ Stale — quiet since 2023	HART is the standard for smart field instruments — pressure, temperature, flow transmitters — in virtually every process plant
Profibus	`pyprofibus`	⚠⚠ Stale — last active June 2023	Profibus is the dominant fieldbus in European process automation, particularly in Siemens installations
DNP3	`pydnp3`	⚠ Abandoned	DNP3 is standard in utilities and water treatment

The maintenance cliff

The pattern is consistent: a library reaches good-enough coverage, the maintainer moves on, and the project quietly stops. Bug reports accumulate. Python version support falls behind. Engineers fork the repo, apply patches, and carry private versions that never get merged back. The ecosystem fragments further.

This is not a criticism of the engineers who wrote these libraries — most did so on their own time, for their own projects. The problem is structural: there is no funded, coordinated effort to maintain industrial protocol coverage in Python. Every company that needs it builds their own private solution.

The gap is confirmed, not estimated

The protocols with the worst coverage — DeviceNet, EtherNet/IP, HART, Profibus — are not obscure. They are the installed base of industrial automation built over the last thirty years. An engineer working on a brownfield plant integration is more likely to encounter these protocols than OPC-UA. The ecosystem has simply failed to serve them.

---

3. Why Fragmentation Is Expensive

The cost of fragmentation is not just the time spent finding and evaluating libraries. It compounds across the entire project lifecycle.

Assembly cost

A multi-protocol connectivity stack — Modbus, MQTT, OPC-UA, and one fieldbus — requires selecting a library for each, resolving dependency conflicts, writing an abstraction layer so the rest of the application does not care which protocol it is talking, and testing each implementation against real or simulated devices. Two to four weeks is a realistic estimate. For a team that does this repeatedly, the cost is borne on every new project.

Inconsistent APIs

Each library has its own conventions. `pymodbus` uses async/await. `python-can` uses callbacks and message filters. `paho-mqtt` uses a different callback model again. `asyncua` has its own node and subscription model. An engineer who knows Modbus must relearn the mental model for every additional protocol. There is no transferable skill — only protocol-specific knowledge.

No packet crafting without hardware

Testing a protocol implementation normally requires a physical device. Without the device on the bench, there is no way to validate that the implementation handles edge cases — malformed frames, out-of-spec responses, timing violations. This means:

- Integration bugs are found late, on-site, during commissioning
- Regression testing requires access to hardware that may be in a plant, not a lab
- New protocol implementations cannot be validated before deployment

A framework that supports packet crafting — constructing arbitrary frames, injecting malformed data, simulating device responses — removes the hardware dependency from testing. This is standard practice in network security tooling (Scapy, boofuzz) but almost entirely absent from industrial protocol libraries.

Duplication across the industry

Every industrial software company, every systems integrator, every engineer building historian agents has solved this problem independently. The solutions are not shared. The work is repeated. The bugs are rediscovered. The ecosystem does not improve.

---

4. The Imbra Connect Architecture

Imbra Connect is built on three abstractions that separate the three concerns in every protocol implementation:

```
Driver     — abstracts the hardware transport (TCP socket, serial port,
CAN interface, USB adapter)
Adapter    — handles protocol encoding and decoding (bytes ↔ Message)
Connection — ties Driver and Adapter together and exposes send/receive to
the application
```

Why this separation matters

Hardware and protocols are independent variables. A Modbus implementation should work over TCP and over serial without duplicating the protocol logic. A CAN adapter should work with a SocketCAN interface, a USB-CAN dongle, or a NetX hardware card without the application knowing the difference.

The Driver/Adapter/Connection split enforces this:

```
Application code
      ↓
  Connection  (send / receive)
   ↙       ↘
Driver    Adapter
(hardware) (protocol)
```

Adding a new hardware target means writing a Driver. Adding a new protocol means writing an Adapter. The Connection and the application code are unchanged.

The abstract interface

```python
class Connection(ABC):
    def connect(self) -> None: ...
    def disconnect(self) -> None: ...
    def send(self, message: Message) -> None: ...
    def receive(self) -> Message: ...

class Driver(ABC):
    def open(self) -> None: ...
    def close(self) -> None: ...
    def read(self) -> bytes: ...
    def write(self, data: bytes) -> None: ...

class Adapter(ABC):
    def encode(self, message: Message) -> bytes: ...
    def decode(self, data: bytes) -> Message: ...
```

Application code programs against `Connection`. Hardware and protocol are injected at construction time. Swapping from Modbus TCP to Modbus RTU is a one-line change. Swapping from a TCP socket to a serial port requires no changes to protocol or application code.

Packet crafting as a first-class concern

Imbra Connect treats packet crafting as a core capability, not an afterthought. Every protocol implementation exposes the ability to:

- Construct standard frames from structured inputs
- Construct non-standard or malformed frames for testing
- Parse raw bytes into structured messages
- Inspect frame fields individually

This is the same philosophy as Scapy for network protocols — but applied consistently across industrial protocols, including those Scapy does not cover (DeviceNet, Profibus, MQTT non-standard frames).

The result is that an engineer can write a complete integration test suite — covering normal operation, edge cases, and fault conditions — without physical hardware. Tests run in CI. Regressions are caught before deployment.

---

5. The Open Source Model

Python and Go — both free, both MIT

Imbra Connect is fully open source in Python and Go. This covers protocol implementations, packet crafting, agent scaffolding, and test tooling. There are no paid tiers within Python or Go. There are no feature gates. The complete SDK is available to anyone.

MIT is the most permissive open source licence. It places no obligation on users to open source their own code. Industrial companies can use Imbra Connect in commercial products, closed-source agents, and proprietary systems without restriction. The only requirement is to keep the copyright notice.

This is a deliberate choice. Friction is the enemy of adoption. Industrial engineers do not need to negotiate licence terms with their legal department to use an MIT library. They install it and get to work.

Prototype in Python. Ship in Go.

Python and Go serve distinct roles in the development lifecycle.

Python is the prototyping language. New protocol implementations are developed first in Python — fast iteration, interactive testing, access to a broad ecosystem. The Python implementation defines the correct behaviour, the frame structure, and the API contract.

Go is the production language. Once a Python prototype is validated, it is ported to Go for release. Go produces a single self-contained binary with no runtime dependencies — the right fit for air-gapped industrial plants, cross-platform deployment, and long-term maintenance. A Go binary from three years ago still runs without modification. A Python environment from three years ago requires active maintenance.

Go has a strong backwards compatibility guarantee. It cross-compiles trivially — one command produces a Linux ARM binary for an industrial Raspberry Pi or a Windows x64 binary for a plant PC. Goroutines are a natural fit for multi-protocol agents that poll devices, buffer writes, and flush to a historian concurrently.

Imbra.Soft maintains Go. The Python codebase is community-maintained after the Go port ships.

Rust and TypeScript — paid ports

Customers who need Rust (embedded, safety-critical, bare-metal) or TypeScript (browser, Node.js, cloud functions) can license commercial ports. These are language-specific optimisations and porting work — not feature restrictions on the open source versions.

---

6. The Community Registry

The model

Every industrial protocol and hardware vendor in the world eventually covered — not by Imbra.Soft alone, but by a community of engineers who have the hardware on the bench and want it to work.

The model: an engineer writes an agent for their Yokogawa DCS, Mitsubishi MELSEC, or custom serial device. They submit it to the Imbra registry. Every other ImBrain user installs it with one command. Imbra.Soft reviews the best ones and promotes them to official status.

This is how Home Assistant grew from 50 device integrations to 3,000+. The same dynamic applies here: the more protocols Imbra Connect covers, the more valuable it becomes for every user, which attracts more contributors, which increases coverage further.

Structure

Agents and the registry are separate things:

Component	What it is	Where it lives
Imbra Connect SDK	Protocol library	`github.com/imbra-ltd/imbra-connect`
Agent	Binary using the SDK	Its own repository
Registry	Index of known agents	`agents.imbra.io`

The registry is a catalogue. Code lives in agent repositories. Installation resolves back to the

source repository.

Tiers

Tier	How it gets there	Guarantee
Community	Submitted by anyone, passes automated checks	Reviewed for safety, not quality
Verified	Imbra-reviewed, tested on real hardware	Works as documented
Official	Maintained by Imbra.Soft, included in installer	Fully supported

Submission rules

Open source under MIT is mandatory for registry submission. No exceptions. This ensures that every agent in the registry can be inspected, audited, and forked. No malware, no telemetry, no phone-home — enforced by Imbra review.

---

7. Protocol Coverage and Roadmap

Current implementations

Protocol	Language	Status
MQTT 3.x / 5.x	Python	Available
CAN	Python	Available
CANopen	Python	Available
DeviceNet	Python	Available

Planned implementations

Protocol	Priority	Notes
Modbus TCP/RTU	High	Go port of existing Python implementation
EtherNet/IP	High	No maintained open library exists — fills confirmed gap
OPC-UA	High	Go: evaluate `gopcua` maturity
HART	Medium	No maintained open library exists
Profibus	Medium	Only stale library exists

Community target protocols

Protocol	Notes
Siemens S7	`python-snap7` exists — agent wrapper straightforward
Yokogawa YDAS	Vendor-specific — community with hardware access needed
Mitsubishi MELSEC	Vendor-specific — community with hardware access needed
DNP3	Utilities and water treatment — no maintained library
Foundation Fieldbus	Controlled by FieldComm Group — original implementation needed

8. Use Cases

Historian agent

The primary use case for Imbra Connect is powering ImBrain data collection agents. An agent reads tag values from a PLC, sensor, or field instrument and forwards them to ImBrain Core over gRPC. Imbra Connect handles the protocol conversation. The agent handles the collection logic — what to read, how often, how to map tags to the historian schema.

With Imbra Connect, the same agent scaffolding works regardless of the underlying protocol. A Modbus agent and an OPC-UA agent share the same structure — only the Connection implementation differs.

Integration scripts for MES and ERP

Engineers writing one-off integration scripts — pulling production counts from a PLC into an ERP, pushing recipe parameters from an MES to a controller — use Imbra Connect for the protocol layer. The consistent API means the script is readable and maintainable by anyone familiar with the framework, not just the original author.

Protocol testing without hardware

Commissioning engineers use Imbra Connect's packet crafting capabilities to validate device behaviour before on-site deployment. Test suites constructed against the frame-level API run in CI and catch regressions before they reach the plant floor. Edge cases — malformed responses, timeout handling, rollover conditions — are tested in software, not discovered during commissioning.

9. Conclusion

The industrial protocol ecosystem has a structural problem. Coverage is fragmented. Maintenance is inconsistent. The protocols with the largest installed base — DeviceNet, EtherNet/IP, HART, Profibus — are the worst served. Engineers pay this cost on every project, in every company, independently.

Imbra Connect addresses the problem at the right layer. A unified abstract interface means application code does not depend on protocol specifics. Packet crafting as a first-class concern removes hardware from the testing dependency chain. Open source under MIT removes licence friction. A community registry means coverage grows with the community, not just with Imbra.Soft's engineering capacity.

The industrial world will not converge on a single protocol. The installed base of legacy devices will persist for decades. The right response is not to wait for convergence — it is to build an abstraction layer that makes the heterogeneity manageable.

That is what Imbra Connect does.

References

- pymodbus — https://github.com/pymodbus-dev/pymodbus
- python-can — https://github.com/hardbyte/python-can
- asyncua — https://github.com/FreeOpcUa/opcua-asyncio
- pycomm3 (archived) — https://github.com/ottowayi/pycomm3
- paho-mqtt — https://github.com/eclipse/paho.mqtt.python
- Scapy — https://scapy.net
- Home Assistant integration count — https://www.home-assistant.io/integrations

---