Amtiri

TECHNICAL INFORMATION

Connectivity

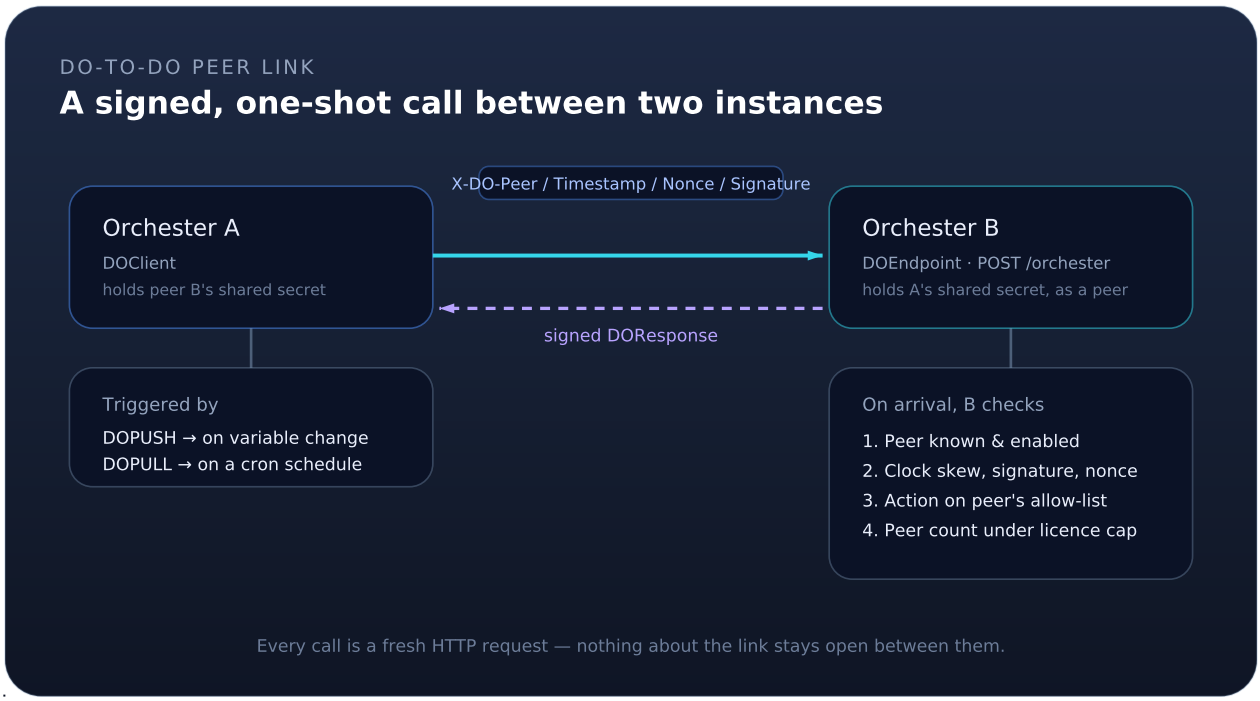
Generated on November 25, 2026



An Orchester instance rarely runs alone. It replicates to and from other instances, accepts data from third-party systems, polls or answers industrial field devices, and joins the buses of vessels and machines. Thirteen mechanisms cover all of it, and this guide walks through each: what it's for, and what to watch out for. The settings of each module are on its own page of the reference manual, listed in the Module catalogue.

Method	Who calls whom	Typical use
DO-to-DO Peer Link	Either side, signed, one-shot	Live values, commands and history between two Orchester instances
External REST API	External system calls in	A third-party system pushing data into Orchester
Modbus Master	Orchester polls out	Reading from and writing to a PLC, meter or drive
Modbus Slave	A remote master calls in	Exposing Orchester's own variables to a SCADA/HMI
OPC UA Client	Orchester dials a server	Reading and writing tags on a PLC or gateway that speaks OPC UA
OPC UA Server	A remote client calls in	Exposing Orchester's own variables to a SCADA, HMI or reporting tool
Siemens S7	Orchester polls out	Reading and writing PLC memory on a Siemens controller with no OPC UA
EtherNet/IP	Orchester polls out	Reading and writing Logix controller tags by name on a Rockwell PLC
IEC 104 Client	Orchester dials an RTU	Interrogating and commanding a telecontrol station over IEC 60870-5-104
IEC 104 Server	A utility master dials in	Exposing Orchester's own variables to a utility SCADA as telecontrol points
NMEA 0183	Orchester listens to a talker; plotters dial in to Orchester	A GNSS receiver, a boat's instruments and AIS in; a chart plotter's feed out
NMEA 2000	Orchester joins the backbone as a node	A vessel's position, depth, wind and engines, and values sent back onto the backbone
CAN bus	Orchester joins the bus	A vehicle's or a machine's CAN bus, by frame or through its DBC file

DO-to-DO Peer Link

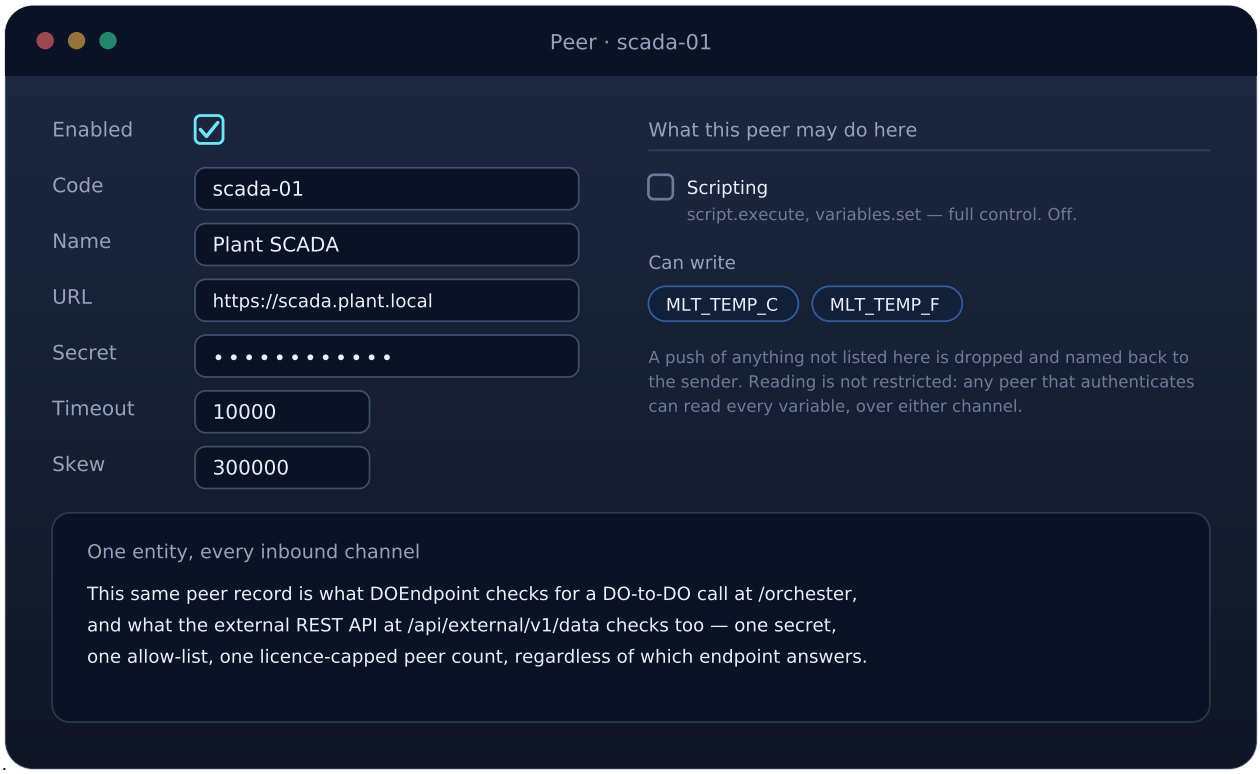


Two Orchester instances talk to each other through a single, generic endpoint: `POST /orchester`. Every call carries one action — `variables.push`, `variables.get`, `variables.set`, `script.execute`, `logger.head`, `logger.append`, or a `ping` — in a signed envelope, and gets a signed reply. There is no persistent socket: each call is an ordinary HTTP request, closed once the response arrives — even a pipe, below, is only a request the dialling side keeps open.

Both instances must hold licences issued to the same organization in the portal: replication stays inside one organization. A peer whose licence names another organization, or none at all, is refused with `FORBIDDEN`.

The request is authenticated with four headers — `X-DO-Peer`, `X-DO-Timestamp`, `X-DO-Nonce`, `X-DO-Signature` — built from an HMAC-SHA256 signature over the method, path, peer code, timestamp, nonce and a hash of the body, using a secret both sides already share. A nonce cache rejects any request replayed inside the clock-skew window, and repeated authentication failures from the same source trip a rate limit.

You don't configure this exchange directly — you configure the **peer** it runs against, and the modules that use it:



Field	Meaning
`enabled`	Whether this peer is usable at all
`code`	The identity the far side signs with
`url`	Where to reach the peer — only needed if this instance calls out to it
`secret`	The shared HMAC key, or `env:VAR_NAME` to read it from an environment variable instead
`timeout`	How long an outbound call waits before giving up (ms)
`skew`	Maximum allowed clock drift between the two sides (ms), default 5 minutes
`writables`	The variables this peer may write here with `variables.push`. Empty means none: a peer saved without a list is a misconfiguration, not a superuser
`scripting`	Whether this peer may run AScript here (`script.execute`, `variables.set`). That is full control of the instance — off by default, and granted only to a peer you trust completely

Two modules ride on top of a peer: `DOPUSH` sends `variables.push` whenever a watched variable changes; `DOPULL` calls `variables.get` on a cron schedule. Both point at the same peer entity, so the URL, secret and permissions only need to be set once no matter how many modules use that link. Their settings, and those of the logger replicator, are in Orchester links.

> **Important**
>
> A peer's permissions cover **writing and scripting only**. Every other action is open to any peer that authenticates: it can read *any* variable with `variables.get`, and append history rows for any variable with `logger.append`. Registering a peer is therefore a decision about who may read your plant, not only who may change it — the secret is the boundary, so treat it as one.

Naming this instance

The `code` in a peer entity names the *far* side. This instance's own name is set separately, in the field at the top of the Orchester panel's left rail, and is stored in `entities/.instance`.

It has to match the `code` of the peer entity that represents this instance on the other machine — that name travels in the `X-DO-Peer` header and is how the far side knows which secret to verify your request with. An instance that has never been named calls itself `ORCHESTER` and says so in its log at every start.

> **Note**
>
> The name does **not** travel in a configuration export. An archive taken from one plant and imported into another must not rename the instance that imported it — two installations answering to the same code collide replication cursors and replay caches. Set it once per installation, like the licence.

Reaching an instance behind NAT

A site on an outbound-only network — behind NAT, a mobile router, or a firewall that admits no incoming connection — has no address the control room can dial. It does not need one. Every enabled peer that has a `url` is dialled automatically: the instance opens a **pipe** to it, one signed `pipe.poll` request held open against the far side's ordinary endpoint. The far side answers that poll with the next request it has for the dialling instance — a `variables.push` from its `DOPUSH`, a `variables.get` from its `DOPULL`, a replicator's `logger.head` — and the result travels back on the next poll.

So the usual layout is a control room with an address, and a site whose peer entity for the control room carries its `url`. The control room's peer entity for the site needs no `url` at all: its calls to the site ride down the pipe.

- A pipe is a transport, not a second protocol. Every request inside it is the same signed envelope, held to the same `writables` and `scripting` permissions, as one sent directly.
- If the far side cannot be reached, the dialling instance backs off and keeps trying, with no restart needed, and says so once in its log.
- When both sides have a `url` for each other, each keeps its own pipe, and each pipe only carries requests addressed to the instance that dialled it. Nothing decides who dials.

Example. Instance `plant-a` pushes two variable values to instance `plant-b`, which has granted it `variables.push`:

```
POST /orchester HTTP/1.1
X-DO-Peer: plant-a
X-DO-Timestamp: 1755000000000
X-DO-Nonce: 7c9e6679-7425-40de-944b-e07fc1f90ae7
X-DO-Signature: 3q2+7wYAAAA9CGFP...

{"v":1,"action":"variables.push","code":"plant-a","id":"7c9e6679-7425-40de-944b-e07fc1f90ae7",
 "data":{"values":{"TEMP_C":21.4,"PUMP_RUNNING":true}}}
```

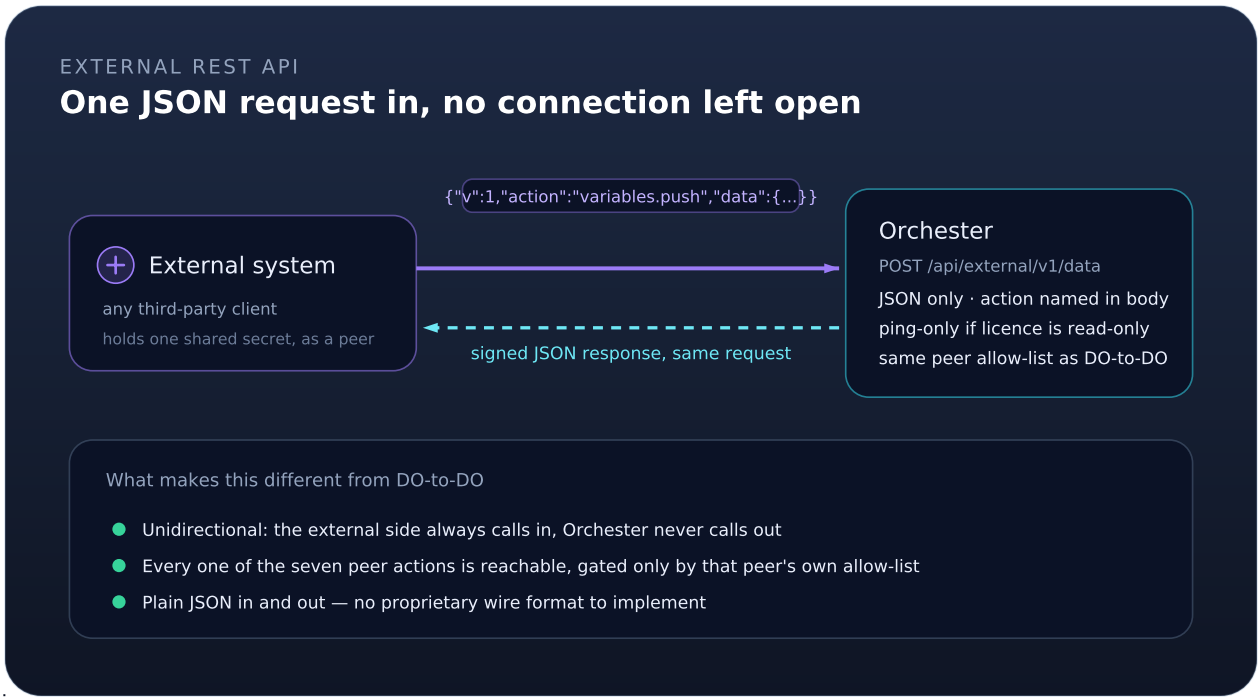
`plant-b` answers with a signed `{"status":"OK","data":{"applied":2,"rejected":0,"rejectedNames":[]}}`.

If `plant-a`'s peer entity on `plant-b` does not list one of those variables under `writables`, that value is dropped and the answer becomes `PARTIAL`, naming what was refused:

```
{"status":"PARTIAL","data":{"applied":1,"rejected":1,"rejectedNames":["PUMP_RUNNING"]}}
```

`PARTIAL` is a success, not a retry: the sender logs the refused names and does not send them again, because a variable the far side will not accept never becomes acceptable by being sent a second time.

External REST API



The DO-to-DO link assumes the far side is another Orchester instance, speaking the same internal wire format. For everything else — a customer's own backend, an integration platform, a script — there's a second endpoint, `POST /api/external/v1/data`, built on exactly the same peer, signing, permission and licence model, but JSON-only and reachable by any system that can compute an HMAC-SHA256 signature.

The request shape is the same one-action-per-call envelope: a JSON body naming the action and carrying its data, the same four `X-DO-*` signature headers, and the same peer permissions deciding what a given external system may write. It's unidirectional by design — the external system always calls in, and Orchester never calls back out to it, so there's nothing to keep open between requests.

One difference from the internal link: when this instance's licence is in a read-only state (expired, invalid, or a tampered clock), the external endpoint answers `ping` only, and refuses everything else. The internal peer-to-peer channel doesn't carry that restriction, so replication between your own instances keeps working even while a licence issue is being sorted out — only the door held open to outside systems narrows.

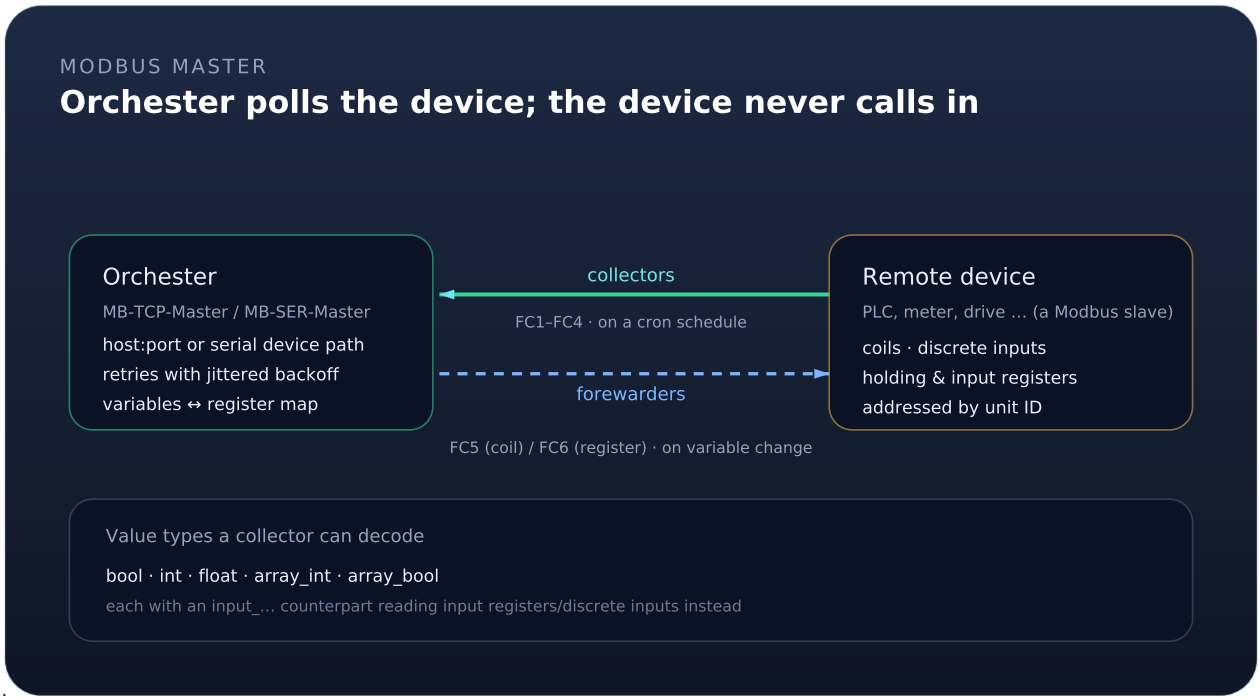
Set this up exactly like a DO-to-DO peer — the same editor, the same two permissions shown above — since it's the same entity either way. List only the variables that integration actually has to write, and leave **scripting off**: it hands the caller full control of the instance, so reserve it for peers you trust completely. Remember that reads are not scoped — an external system you register can query every variable in the plant.

Example. An ERP system, registered as peer `erp-integration` with `ORDERS_COMPLETED` as its only writable variable and no scripting, records a completed order count:

```
curl -X POST https://plant.example.com/api/external/v1/data \
  -H "Content-Type: application/json" \
  -H "X-DO-Peer: erp-integration" \
  -H "X-DO-Timestamp: 1755000000000" \
  -H "X-DO-Nonce: 3fa85f64-5717-4562-b3fc-2c963f66afa6" \
  -H "X-DO-Signature: <base64 HMAC-SHA256 over method, path, peer, timestamp, nonce and body>" \
  -d '{"v":1,"action":"variables.push","data":{"values":{"ORDER_COUNT":128}}}'
```

The signature is computed the same way on any platform: HMAC-SHA256 over `POST\n/api/external/v1/data\n<peer>\n<timestamp>\n<nonce>\n<sha256 of the body>`, using the peer's shared secret — there's no DataOrchester-specific SDK required, just a standard crypto library.

Modbus Master



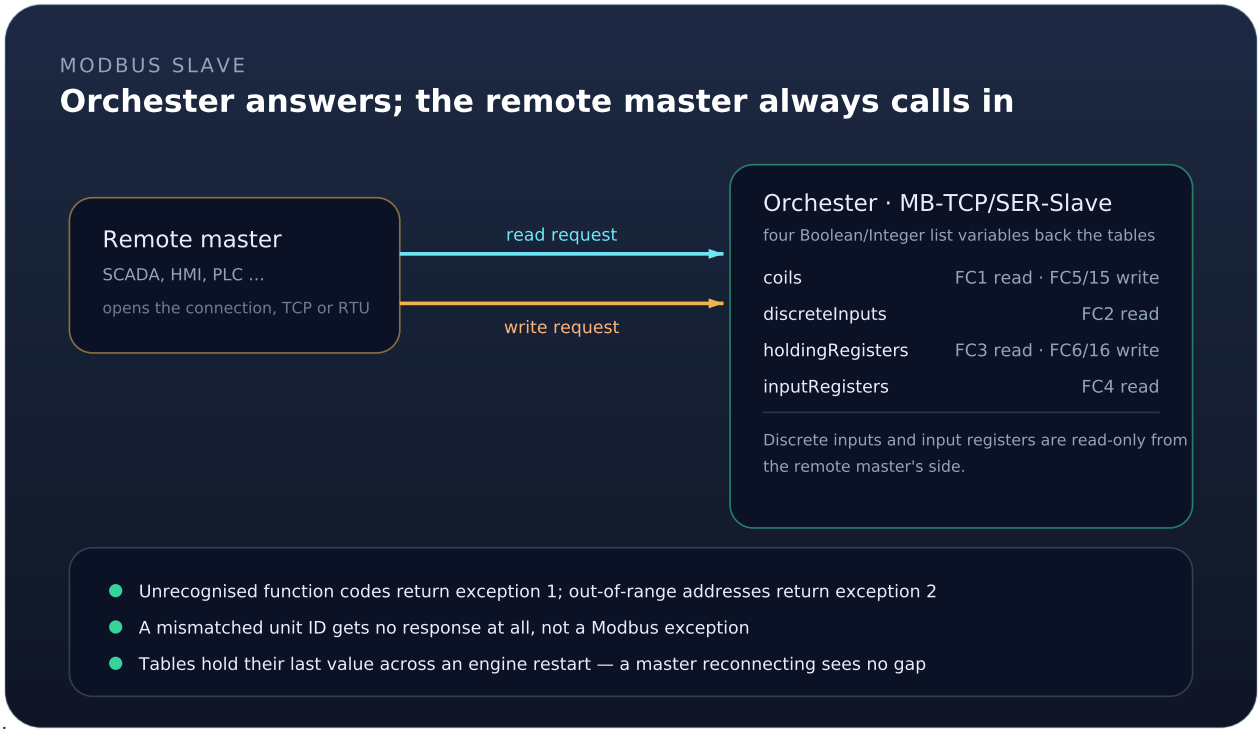
As a Modbus master, Orchester is the one opening the connection — to a PLC, a power meter, a variable-speed drive, anything that answers Modbus requests. Two module types exist:

- `MB-TCP-Master` speaks Modbus TCP. It is given a host, a port and one unit ID for every request.
- `MB-SER-Master` speaks RTU on an RS-485 or RS-232 line. It is given the serial port, baud rate, parity and stop bits, and each row names the unit ID it asks, so one line can reach several devices.

Both read and write through the same two lists: **collectors** read coils or registers into variables on a schedule, and **forwarders** write a variable to the device when it changes. Addresses are 0-based offsets, sent as they are: holding register 40010 in the 4xxxx notation is address 9.

The settings, register types and examples are in Modbus.

Modbus Slave

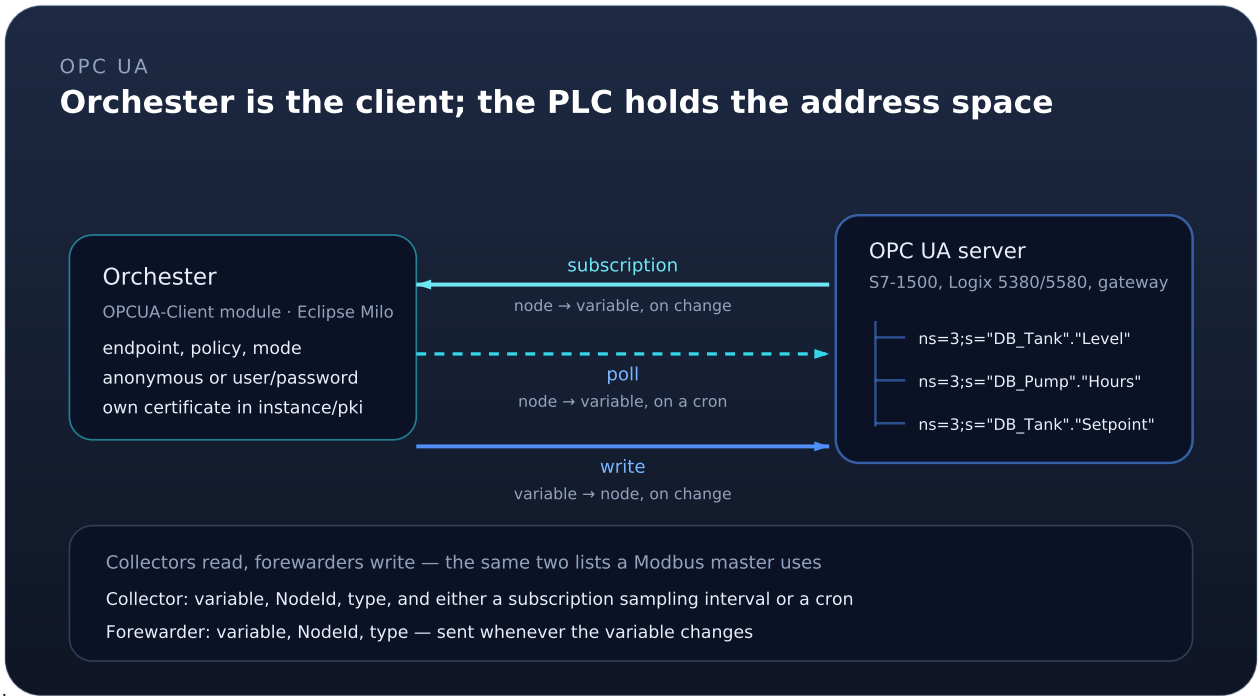


Flip the direction, and Orchester becomes the thing being polled: `MB-TCP-Slave` and `MB-SER-Slave` turn an instance into a Modbus slave that a SCADA system, an HMI or another PLC can read from and write to. Four list variables back the four Modbus tables. Coils and holding registers accept a master's writes; discrete inputs and input registers are read-only.

A request for a unit ID other than the slave's own gets no answer at all — indistinguishable, on the wire, from the slave not existing. A master's write goes through the variable's ordinary write, so only a bank variable marked persistent keeps what a master wrote across an engine restart.

The settings and the tables are in Modbus.

OPC UA Client



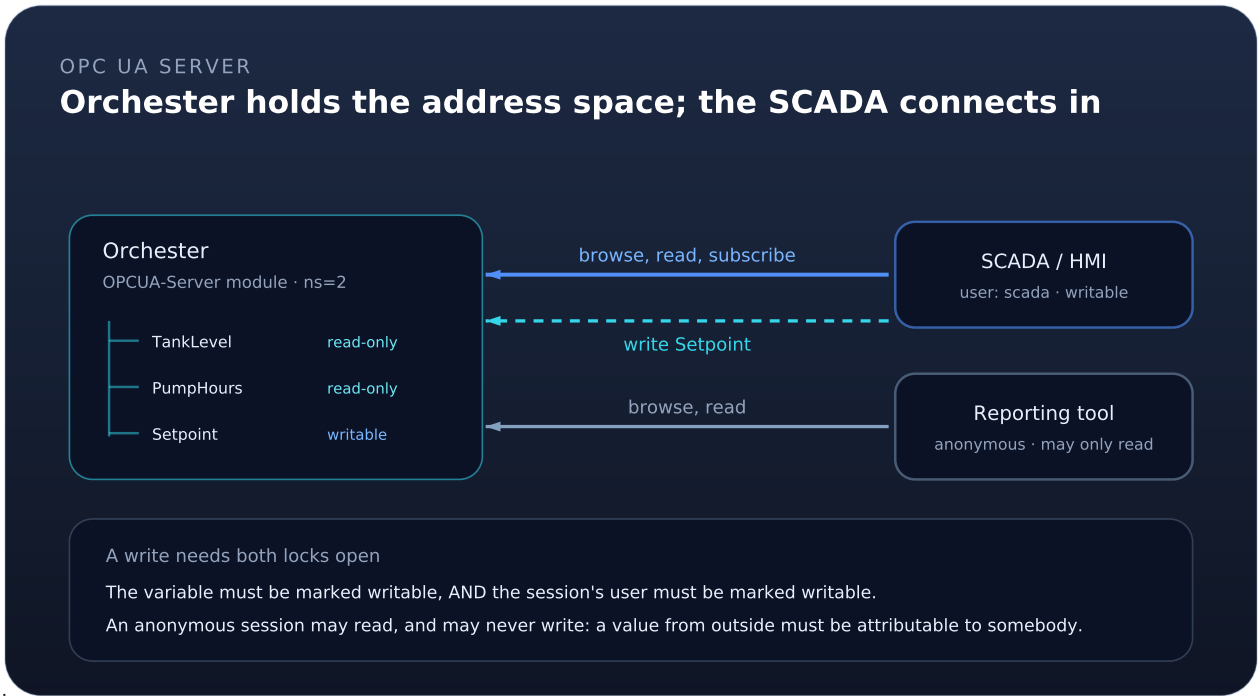
The `OPCUA-Client` module makes Orchester an OPC UA client. It opens one session to a server — a PLC through its built-in server, a dedicated OPC UA gateway, or another SCADA product — and reads and writes nodes in that server's address space. A node is either subscribed, and the server sends its value when it changes, or polled on a schedule. The session can sign and encrypt, and the user's password can be read from the environment instead of being stored.

The settings, types and examples are in OPC UA Client.

Certificates

Any security policy other than `None` means both ends authenticate by certificate. Orchester keeps its own, and the ones it has been shown, under `instance/pki/opcua`. A server it has not been told to trust is **refused**, and there is no trust-on-first-use: trusting it is a deliberate act, moving its certificate from `rejected/` to `trusted/`. The first connection to a secured server is therefore expected to fail once. The folders and the rules are in Certificates.

OPC UA Server



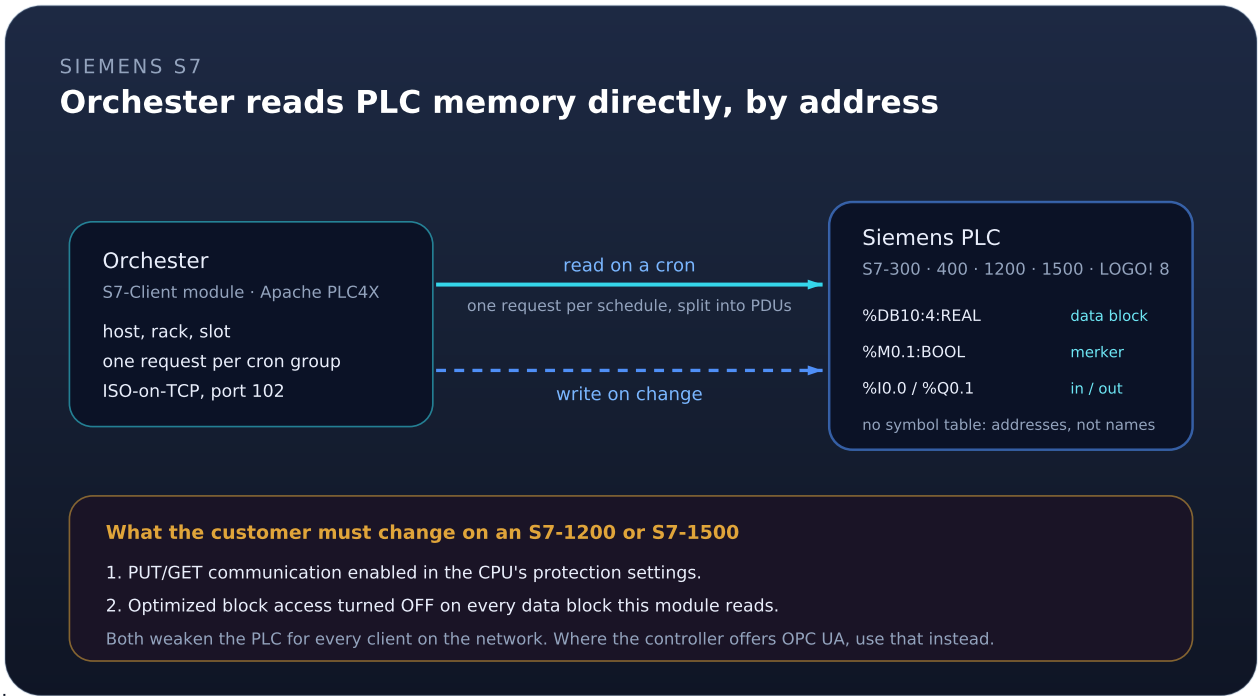
The `OPCUA-Server` module is the other direction: a SCADA, an HMI or a reporting tool connects *in*, at `opc.tcp://<host>:<port>/orchester`, and reads variables this instance holds. It is to the OPC UA client what the Modbus slave is to the Modbus master — with the difference that OPC UA can say who is connecting, and Modbus cannot. Every security policy except `None` is offered with signing and encryption only.

The settings and the address space are in OPC UA Server.

Who may write

Both locks have to be open. A write is accepted only when the variable is marked writable *and* the session's user is marked writable, and an anonymous session may read but never write. A client writing a variable that is not writable gets `Bad_NotWritable`; a user who may not write, anonymous included, gets `Bad_UserAccessDenied`.

Siemens S7



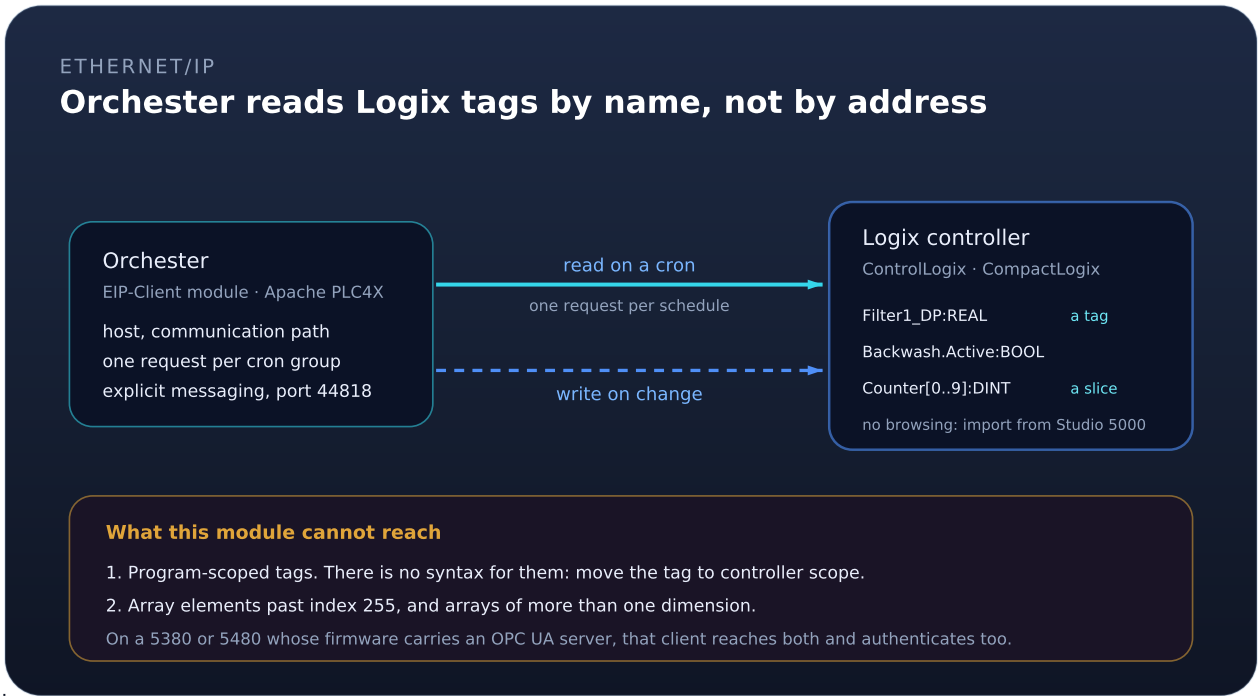
The `S7-Client` module reads and writes Siemens PLC memory directly, over the classic S7comm protocol on port 102. It covers the controllers that have no OPC UA server of their own — S7-300, S7-400, LOGO! 8 — and the S7-1200 and S7-1500 whose owners have not switched theirs on.

> **Warning**
>
> **On an S7-1200 or S7-1500 the customer has to change two settings in the CPU**, and both weaken it: **PUT/GET communication** must be enabled in the protection settings, and **optimized block access** must be turned off on every data block this module reads. Neither change is specific to Orchester — they open the PLC to every client on the network. **Where the controller offers OPC UA, use the OPC UA client instead:** it needs no such change, and it authenticates and encrypts.

An S7 address states its own type — `%DB10:4:REAL` is a 32-bit float at byte 4 of data block 10 — so there is nothing else to keep in step. Addresses sharing a schedule are read in one request. There is no symbol browsing and there are no subscriptions: addresses are typed or pasted from TIA Portal, and every one is polled.

The settings, the address grammar and the types are in Siemens S7.

EtherNet/IP



The `EIP-Client` module reads and writes Rockwell Logix controller tags **by name**, over EtherNet/IP explicit messaging on port 44818.

> **Tip**
>
> On a controller whose firmware carries an **OPC UA server**, prefer the OPC UA client: it reaches program-scoped tags, which this module cannot, and it authenticates and encrypts.

The settings, the types and the import are in EtherNet/IP.

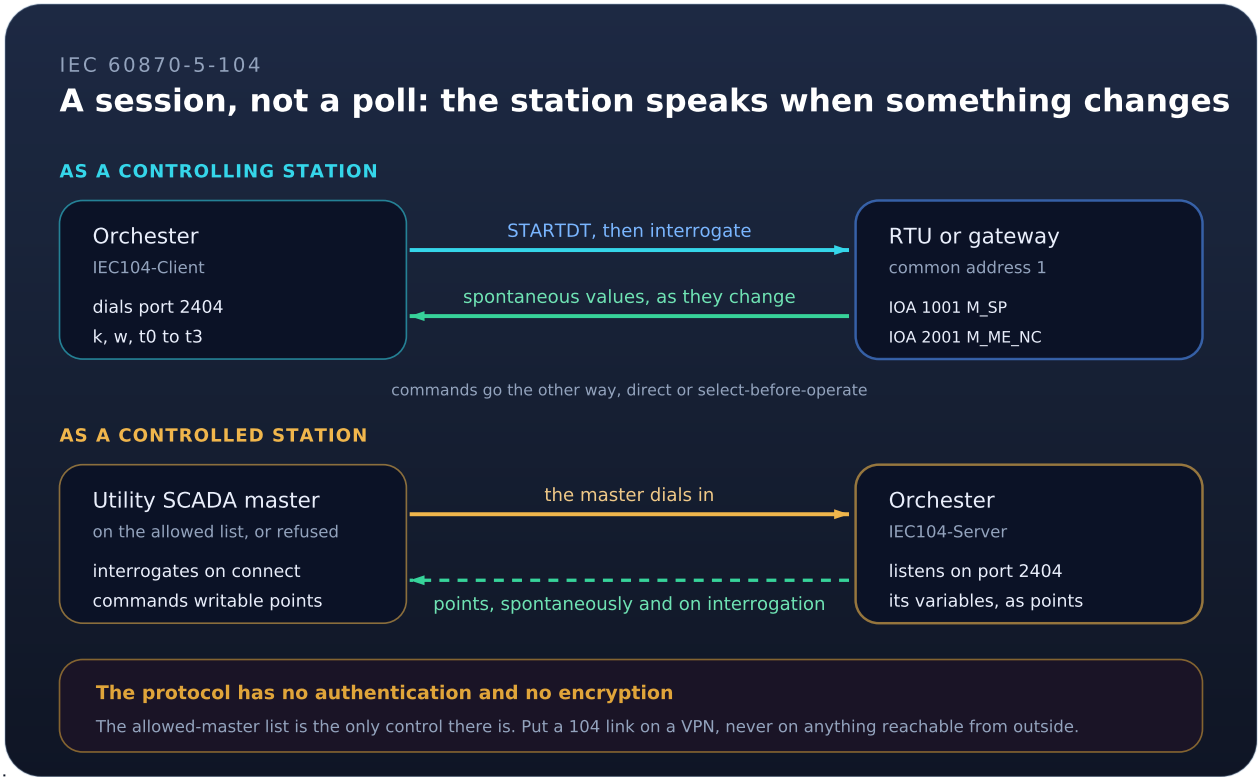
Naming a tag

A tag address is a name, then its type: `Filter1_DP:REAL`, `Backwash.Active:BOOL`, one element as `Counter[3]:DINT`, or a slice as `Counter[0..9]:DINT`, which arrives as a list. Three shapes are refused when the module opens: a program-scoped tag such as `Program:MainProgram.Tag`, an index past 255 such as `Counter[256]:DINT`, and a second dimension such as `Counter[0..9,0..3]:DINT`. The full rules are in Naming a tag.

Getting the tag list in

This driver **cannot browse a controller**, so the tag list comes from Studio 5000: export it with *Tools # Export # Tags* and use **Import tags** in the editor. What the import keeps and skips is in Importing tags.

IEC 60870-5-104



Two modules, one protocol. `IEC104-Client` is a *controlling station*: it dials an RTU or a substation gateway, interrogates it and commands it. `IEC104-Server` is a *controlled station*: it answers a utility's SCADA master and exposes Orchester's own variables as telecontrol points.

> **Caution**
>
> **This protocol has no authentication and no encryption.** Anyone who can reach port 2404 can read every point and operate every writable one. The allowed-master list is the only control the module offers, and it is not a substitute for a network: put a 104 link on a **VPN or a dedicated circuit**, never on anything reachable from the internet. IEC 62351 security is not implemented.

The settings, the types and the examples are in IEC 60870-5-104.

A session, not a poll

A 104 link is a **session**: the controlling station connects and sends STARTDT, interrogates the station for everything it has, and from then on the station sends what changes, as it changes. Nothing is polled, so a point has an address rather than a schedule. See A session, not a poll.

Naming a point

A point is an **information object address**, which the utility assigns, and a **type**, named by its family — `M_SP` accepts both `M_SP_NA_1` and the time-tagged `M_SP_TB_1` — or by its exact name. See Points and types.

The six link parameters

`k`, `w`, `t0`, `t1`, `t2` and `t3` have to match what the far end uses; a utility states all six in its interoperability document. See Link parameters.

As a controlling station

A command can go **select-before-operate**: the client selects the point, and executes only once the station has confirmed the selection. The client offers the station its clock and never takes the station's, and it never asks for counters: they arrive only when the station sends them. See IEC 60870-5-104 Client.

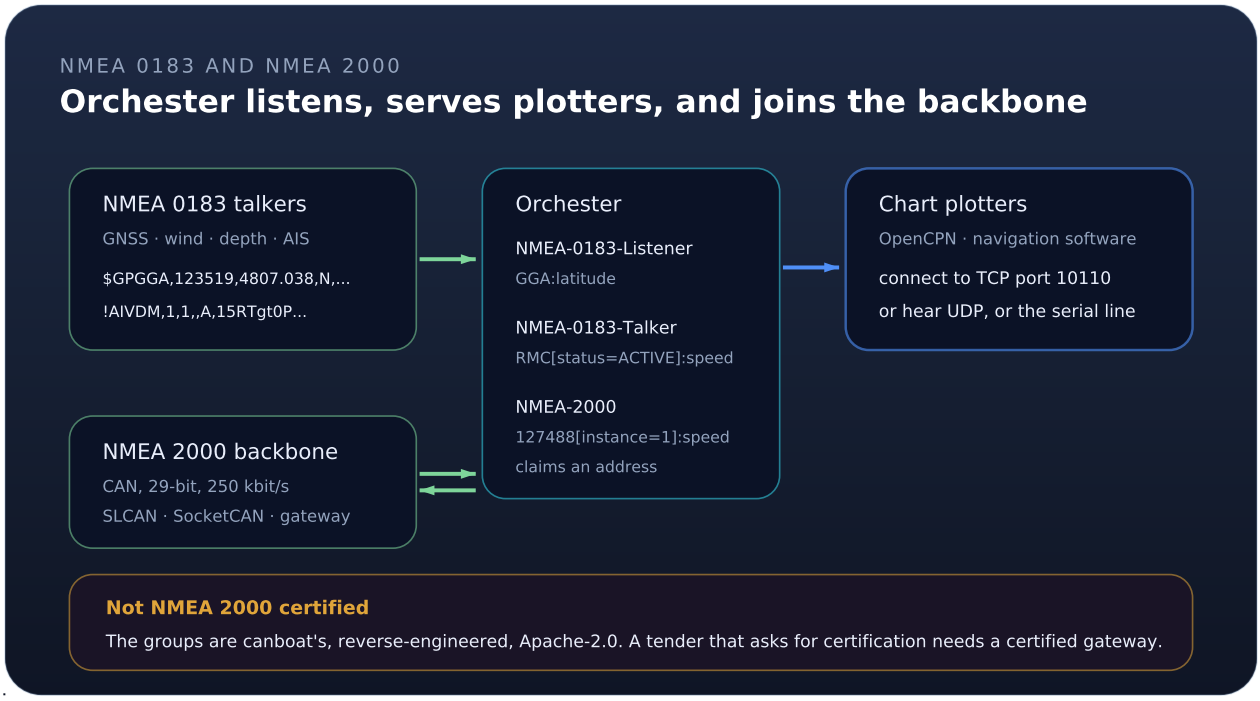
As a controlled station

`allowedClients` **is fail-closed**: a master not named in it is refused, and an **empty list refuses everyone**. Each point says whether it is reported spontaneously, with what deadband, and whether a command may write it. A clock synchronisation from a master is confirmed and not applied. See IEC 60870-5-104 Server.

The point list

Both editors import and export the point list as CSV, with the columns `IOA,TYPE,VARIABLE,OPTIONS`, because that is how a utility describes a link. See The point list as CSV.

NMEA



Three modules, two protocols. `NMEA-0183-Listener` listens to a talker - a GNSS receiver, a wind instrument, an AIS receiver, a multiplexer - on a serial port, a TCP feed or UDP. `NMEA-0183-Talker` builds sentences from variables and serves them, on TCP port 10110 by convention, to chart plotters and navigation software. `NMEA-2000` joins an NMEA 2000 backbone through an SLCAN adapter, SocketCAN or a gateway, reads its parameter groups and sends its own as a node.

> **Important**
>
> Data Orchester reads and writes NMEA 2000 parameter groups, from canboat's reverse-engineered definitions. It is not NMEA 2000 certified. Where a tender asks for certified equipment, put a certified gateway in between.

The settings, the address grammar and the examples are in NMEA.

Naming a value

An NMEA 0183 value is a sentence and a field, `GGA:latitude`, with a talker in front when only one talker counts: `GPGGA:latitude`. An NMEA 2000 value is a parameter group and a field, `128267:depth`. Selectors narrow either: `127488[instance=1]:speed` is the starboard engine, `AIS[MMSI=257123456]:speedOverGround` one vessel. See Addresses.

Joining a backbone

Before it may send anything on an NMEA 2000 backbone, the module claims an address and answers what other devices ask a node: its product information and the groups it sends and receives. Behind an Actisense gateway the gateway is the node instead. See Joining as a node.

CAN



Two modules read and write a CAN bus - a vehicle's, a mobile machine's, a J1939 engine's - through an SLCAN USB adapter on any system, or SocketCAN on Linux. **`CAN-Raw`** names each value by its frame and bits: `00FEE500/00FFFF00:uint@24:16`. **`CAN-Signals`** names it by signal, through the machine's DBC file: `EEC1` and `EngineSpeed`, in rpm. Several modules share one adapter.

The settings, the address grammar and the DBC subset are in CAN.

Nothing is polled

A bus pushes: a frame arrives when its sender sends it. A collector has no schedule; it has a minimum interval, so a value arriving a hundred times a second is written at most as often as asked, and the last one always lands. A forwarder is sent when its variable changes and, with an interval, on its own clock. See What is sent, and when.

What the status says on each link

Every module here reports through its status variable — see Settings every module shares. What each value means depends on what the module is talking to.

Module	Warning	Error
Modbus master	Some collectors are answering and some are not, or a slave returned a Modbus exception for one register	Nothing has answered for `errorAfterFailures` polling cycles in a row
Modbus TCP slave	A client that was connecting has stopped, or a request arrived that could not be read	The port could not be bound at all — usually another process already has it
Modbus serial slave	Frames are arriving with a bad checksum, which normally means wiring, termination or the baud rate	The serial port has gone, and has stayed gone for `errorAfterSeconds`
OPC UA client	The session dropped and is being re-established, the server refused some subscribed nodes, a node returned a bad status, or a write was refused	The server could not be reached at start or has been unreachable for `errorAfterSeconds`, its certificate is not trusted, or reads failed `errorAfterFailures` times in a row
OPC UA server	A published variable could not be updated on its node, or a client wrote a value its variable refused	The port could not be bound at all — usually another process already has it — or no security policy is usable
Siemens S7	The connection dropped, the PLC refused an address, or a write failed	The PLC has been unreachable for `errorAfterSeconds`, or an address is unusable
EtherNet /IP	The connection dropped, the controller refused a tag, or a write failed	The controller has been unreachable for `errorAfterSeconds`, or a tag is unusable
IEC 104 client	The link dropped or was stopped, a point was reported invalid, a command was refused, or the station refused an interrogation	The station has been unreachable, or its link stopped, for `errorAfterSeconds`
IEC 104 server	A master disconnected, until it connects again; a master not on the allowed list was refused; or a command was refused	The port could not be bound at all — usually another process already has it
NMEA 0183	The talker's port or feed dropped, sentences are being lost to bad checksums or garbling, or a client of the talker disconnected or was dropped for not keeping up	The port or feed has been gone for `errorAfterSeconds`, or the talker's port could not be bound
NMEA 2000	The adapter or gateway dropped, the controller is error passive, or a group could not be sent	The backbone has been unreachable for `errorAfterSeconds`, the controller is bus-off, or no address could be claimed
CAN bus	The adapter dropped, the controller is error passive, or a frame could not be sent	The adapter has been gone for `errorAfterSeconds`, or the controller is bus-off
Peer link (push, pull, replicator)	Part of a batch was refused or not delivered, or some variables are not defined on the far side	The peer refused or was unavailable for `errorAfterFailures` attempts, or it is not configured here at all, is disabled or has no secret

A Modbus master with one dead register among forty stays at a warning rather than going to error: the link is up and the device is there, which is a different problem from a plant that has lost its PLC.

The texts each module writes are listed on its page, and all of them in Module messages.

Choosing between them

- Talking to **another Orchester instance**: use the DO-to-DO peer link — it's already there, and `DOPUSH` / `DOPULL` cover the common push/pull patterns without any custom code.
- Talking to **a third-party system that can call you**: use the external REST API. It reuses the same peer and licence model as DO-to-DO, so a customer's integration is one more peer entity, not a new trust boundary.
- Talking to **a field device that speaks Modbus**: use Modbus Master if Orchester should poll the device, or Modbus Slave if the device (or a SCADA layer above it) needs to poll Orchester instead. Both can run at once for the same device family, if some points are read there and others are written here.
- Talking to **a Siemens PLC with no OPC UA server**: use the S7 client — but read what it asks the customer to switch off first, and prefer OPC UA on any controller that offers it.
- Talking to **a Rockwell Logix controller**: use the EtherNet/IP client, and import the tag list from Studio 5000 rather than typing it. On a controller with an OPC UA server in its firmware, prefer OPC UA — it reaches program-scoped tags, which EtherNet/IP here cannot.
- Talking to **a water, power or gas utility's telecontrol network**: use the IEC 104 client to read an RTU, the IEC 104 server to be read by the utility's SCADA, or both. It is the only protocol here that a utility will usually insist on by name — and the only one with no security of its own, so it belongs on a VPN.
- Talking to **a vessel's instruments**: use the NMEA 0183 listener for a talker or a multiplexer, the NMEA 2000 module for a backbone, and the NMEA 0183 talker to feed the chart plotter or OpenCPN with what Orchester holds.
- Talking to **a vehicle's or a machine's CAN bus**: use CAN Signals with the manufacturer's DBC file, and CAN Raw for the frames no DBC file describes.
- Exposing **Orchester's own variables to a SCADA, HMI or reporting tool that speaks OPC UA**: use the OPC UA server. It is to the OPC UA client what the Modbus slave is to the Modbus master, and unlike Modbus it can say who is allowed to write.
- Talking to **a PLC or gateway that speaks OPC UA**: use the OPC UA client. On a modern Siemens or Rockwell controller this often reaches tags by name without any Modbus mapping, and it carries authentication and encryption that Modbus has no notion of.

All thirteen share the same underlying safety property: none of them can be configured into pulling more than a licensed instance is entitled to, and none of them stops a running engine on its own. The licence can: *What stops a running engine*, in How licensing works, says when. A stopped engine writes nothing more to any device, so every device it drives needs a safe state of its own.

Next steps

- Module catalogue
- Settings every module shares
- Configuration reference
- Licence activation