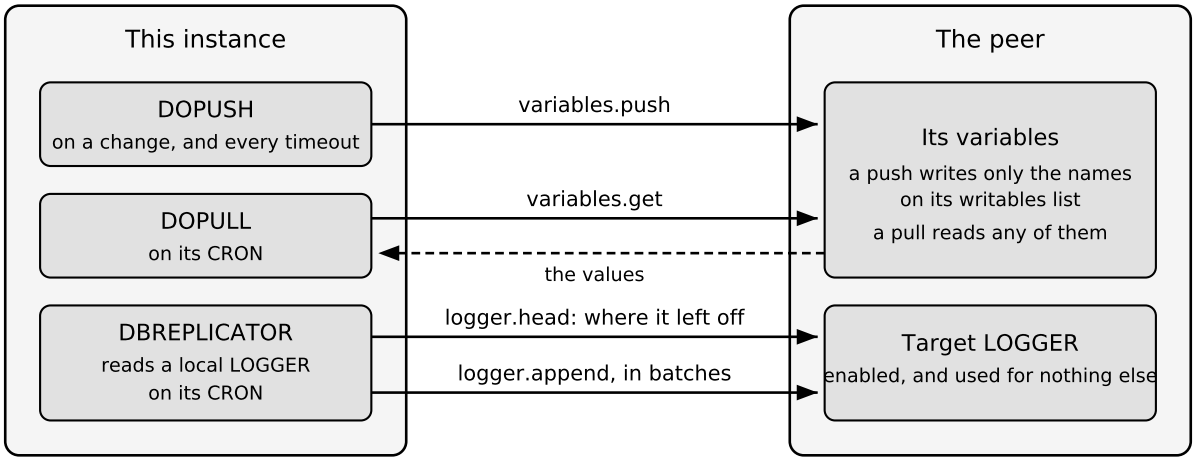
Amtiri

REFERENCE MANUAL

Orchester links

Generated on November 25, 2026



Three modules carry data between two Data Orchester instances over the signed peer link: live values in either direction, and logged history towards the instance that collects it. The peer link itself, with its peer entities, secrets, permissions and the pipe that reaches an instance behind NAT, is described in Connectivity; this page describes the modules that use it.

Module	What it does	Typical use
Orchester Push (`DOPUSH`)	Sends local variables to a peer when they change, and again every few minutes	A station reporting its readings to a control room
Orchester Pull (`DOPULL`)	Asks a peer for variables on a schedule, and writes them locally	A control room sampling stations that are not set up to push
Logger Replicator (`DBREPLICATOR`)	Copies a local LOGGER's rows to a LOGGER on a peer	Gathering every station's history in one place

Every type also has the settings all modules share: whether it is enabled, its name, its status and message variables and their thresholds. They are described once, in Settings every module shares.

What every link needs

- **A peer entity on each side.** Here, one for the far instance, with the shared secret and, when this side dials, its address. There, one for this instance, with the same secret. Both instances' licences must belong to the same organization.
- **One module, one peer.** Each module talks to exactly one peer: the one whose code is in the module's **DataOrchester** setting. Any number of modules may use the same peer.
- **A running far side.** An instance that is stopped answers every call as unavailable.
- **Time.** The peer entity's `timeout`, 10 000 ms unless set, bounds each call, direct or over a pipe. Connecting gives up after 5 seconds.

- **Size.** A far side takes at most 500 values in one push request, at most 500 names in one pull request, and 8 MiB in any request. A push or pull module with more names splits them into several requests of at most 500.

Orchester Push

Sends the values of local variables to a peer, which writes them into its own variables of the same name, or of the same name with a prefix.

Settings

Key	Setting	Type	Default	Meaning
`peer`	DataOrchester	peer code	none	The peer to send to
`delay`	Delay [ms]	int	`250`	How long changes are gathered before they are sent together
`timeout`	Timeout [s]	int	`300`	How long a variable that has not changed waits before it is sent again
`prefix`	Prefix	text	none	Put in front of each code as it is sent
`variables`	Variables	variable codes, separated by commas	none	The variables to send. Empty sends nothing

A **Delay** or **Timeout** left empty takes its default.

How values are sent

- **Once at the start.** Every listed variable is sent shortly after the module opens.
- **On a change.** Changes are gathered for `delay` and sent together, so a change leaves within about twice `delay`, in one request with every other change of that moment.
- **Again every `timeout`.** A variable that has not changed is sent again once `timeout` seconds have passed since it was last sent, so a far side that restarted, or missed a request, catches up without waiting for the value to move.
- **Retrying.** When the peer could not be reached or could not answer, the same variables go again at the next pass, `delay` later. When the peer refused the request, they are not resent until they change or their `timeout` comes round.
- **Long lists.** More than 500 variables due at once, as at the start of a module that lists more, go in several requests of at most 500, one after the other.
- **The prefix.** With `prefix` `ST2_`, `LIT_301` is sent as `ST2_LIT_301`.
- **Codes in any case.** A code in `variables` may be written in any case, and is sent once, in capitals. Before engine 6.18.8 a code written in lower case was sent twice.

> **Warning**
>
> Keep `delay` above 0: at 0 the module checks for changes without pausing, and keeps a processor core busy. A `timeout` of 0 sends every variable at every pass, four times a second at the default `delay`.

What the far side accepts

The far side writes a value only when the peer entity it holds for this instance lists the name, as received and so with the prefix, in its `writables`. The names it refuses are named in its answer, and the module shows `{n} of {total} variable(s) delivered`. A refused name is not resent at once; it goes again with the next change or the next `timeout`, and is refused again until the far side lists it. Each name should be a variable defined on the far side, where the value is converted to that variable's declared type.

> **Important**
>
> A module may list more than 500 variables, but the far side may run its formulas between two of its requests: a formula there that reads values from two requests may, for a moment, see one new and one old. Before engine 6.18.8 a push module sent everything due in one request, and one listing more than 500 was refused whole, as `TOO_LARGE`, at every start.

Messages

A counted failure is a `WARNING` that reads `… (1 of 3)` and `… (2 of 3)`, and becomes an `ERROR` reading `… after 3 attempts`, with the thresholds set in Settings every module shares. The complete list is in Module messages.

When	Level	Text
No loaded peer has the code in **DataOrchester**	`ERROR`	`Peer '{peer}' is not defined`
The same, while other peers are loaded: `{peers}` lists their codes	`ERROR`	`Peer '{peer}' is not defined (known: {peers})`
The peer is disabled, or its secret does not resolve	`ERROR`	`Peer '{peer}' is disabled or has no secret`
The far side refused the request	`WARNING`, then `ERROR`	`Peer '{peer}' refused the request: {reason}`
The far side could not be reached, is not running, or failed to answer	`WARNING`, then `ERROR`	`Peer '{peer}' unavailable: {reason}`
The far side refused some of the names, or some of a pass's requests failed	`WARNING`	`{n} of {total} variable(s) delivered`

- **Not defined** covers two cases: no peer entity has that code, or the licence admits no more peers. The text then lists the peers that did load, as `(known: …)`.
- **Has no secret** covers a peer whose secret does not resolve: an `env:` name the service's environment does not set. Such a peer is loaded, but it can neither call nor be called, and it takes no place among the peers the licence admits. Before engine 6.18.8 it was left out, and modules naming it read `Peer '{peer}' is not defined`.
- **The reason** is the answer's status, followed by its message where there is one: `UNAUTHORIZED` for a wrong secret, a clock more than 5 minutes apart or an instance the far side does not know; `FORBIDDEN` for another organization; `TOO_LARGE` for more than 500 values in one request; `UNAVAILABLE` for a far side that is stopped or cannot be reached. `ERROR - Unsigned or unverifiable response.` means the far side answered without a valid signature, as it does to a request over 8 MiB and while it holds off a sender after repeated failed authentications.
- **One report per pass.** A pass of several requests reports once for all of them: when every request failed, the last failure, counted once; when some failed, `{n} of {total} variable(s) delivered`. A pass that delivers everything clears the failures.

Example

A station, `STATION_2`, reports three readings to the control room, whose peer entity for it lists `ST2_LIT_301`, `ST2_FIT_501` and `ST2_P101_RUN` in its `writables`:

```
DataOrchester   CONTROL_ROOM
Delay [ms]      250
Timeout [s]     300
Prefix          ST2_
Variables       LIT_301, FIT_501, P101_RUN
```

A change of `LIT_301` reaches the control room as `ST2_LIT_301` within about half a second, and all three are sent again every five minutes whether they changed or not. Had the control room listed only the first two, the first send after the start would leave the module warning `2 of 3 variable(s) delivered`.

Orchester Pull

Asks a peer for the current values of its variables on a schedule, and writes them into local variables. The far side needs nothing but a peer entity for this instance: any peer that authenticates may read its variables.

Settings

Key	Setting	Type	Default	Meaning
`peer`	DataOrchester	peer code	none	The peer to ask
`cron`	CRON	Quartz cron	`0 0/5 * * * ?`	When to ask. The default is every 5 minutes; the syntax is in Settings every module shares
`prefix`	Prefix	text	none	Put in front of each code as the value is written here
`variables`	Variables	variable codes, separated by commas	none	The far side's variables to ask for. Empty asks for nothing

How values arrive

- **One request per tick**, asking for every listed variable. A list of more than 500 is asked for in several requests of at most 500, and the answers are written together once the last one is in. The first request is made at the first tick after the module opens, not when it opens.
- **Written locally** under the far side's code, with the prefix in front: with `prefix` `ST2_`, the far side's `LIT_301` is written into `ST2_LIT_301`. The value goes through the engine's write path, converted to the local variable's declared type; a value equal to the one held changes nothing.
- **Missing names.** A variable that the far side does not have, or that holds no value there, is left out of the answer. The local variable keeps what it held, and the module warns.
- **A slow peer.** A tick that falls while the previous request is still waiting for its answer is skipped.

> **Important**
>
> A module may list more than 500 variables, but the far side reads each request's values when that request arrives, so values asked for in different requests may be a moment apart. Before engine 6.18.8 a pull module asked for its whole list in one request, and one listing more than 500 was refused whole, as `TOO_LARGE`, at every tick.

Messages

When	Level	Text
Opening: the CRON is empty. The module does not open	`ERROR`	`No schedule given`
Opening: the CRON cannot be read. The module does not open	`ERROR`	`Invalid schedule '{cron}': {reason}`
No loaded peer has the code in **DataOrchester**	`ERROR`	`Peer '{peer}' is not defined`
The same, while other peers are loaded: `{peers}` lists their codes	`ERROR`	`Peer '{peer}' is not defined (known: {peers})`
The peer is disabled	`ERROR`	`Peer '{peer}' is disabled or has no secret`
The far side refused the request	`WARNING`, then `ERROR`	`Peer '{peer}' refused the request: {reason}`
The far side could not be reached, is not running, or failed to answer	`WARNING`, then `ERROR`	`Peer '{peer}' unavailable: {reason}`
Some names have no value on the far side	`WARNING`	`{n} variable(s) not defined on the peer`
Some of a tick's requests failed	`WARNING`	`{n} of {total} variable(s) delivered`

The peer messages mean what they mean for Orchester Push, and a tick of several requests reports once, as a push pass does. A tick whose every name arrives clears the warning.

Example

The control room samples the same station every minute:

```
DataOrchester   STATION_2
CRON            0 * * * * ?
Prefix          ST2_
Variables       LIT_301, FIT_501, P101_RUN
```

Every minute the control room writes `ST2_LIT_301`, `ST2_FIT_501` and `ST2_P101_RUN`. If the station's `FIT_501` has never held a value, the module warns `1 variable(s) not defined on the peer` and the other two still arrive. When the station sits behind NAT and dials the control room, the requests travel down the station's pipe, and the control room's peer entity for it needs no address.

Logger Replicator

Copies the rows a local LOGGER has written to a LOGGER on a peer, so the peer holds the history too. It sends what the far side does not have yet, so a link that was down for a day is caught up once it is back, with nothing lost.

Settings

Key	Setting	Type	Default	Meaning
`logger`	Logger	LOGGER code	none	The local logger to read
`peer`	DataOrchester	peer code	none	The peer to copy to
`target`	Target	LOGGER code on the peer	none	The logger on the peer that receives the rows
`prefix`	Prefix	text	none	Put in front of each code on the far side: it names the series there
`cron`	CRON	Quartz cron	`0 0/5 * * * ?`	When to replicate. The default is every 5 minutes
`batch`	Batch	int	`500`	How many rows one request carries
`variables`	Variables	variable codes, separated by commas	none	The series to copy. Empty copies nothing

A new record also holds a `code` key, which the module does not use.

How rows are copied

At each tick the module:

1. Asks the target logger, in one request, for the latest timestamp it holds of each series, under the far side's name, prefix included.
2. Reads the local rows of each series from that timestamp on, the boundary included, and sends them in batches of `batch` rows. A series the target has never seen is sent from its first row.
3. Moves on to the next series when a series is caught up, or after 20 batches.

- **Nothing is lost or doubled.** The far side drops rows it already holds, so the rows at the boundary, or a batch sent again after a lost answer, are written once. A batch that fails is sent again from the same point at the next tick.
- **A backlog is worked off.** Rows beyond 20 batches of one series wait for the next tick, and the module warns until it has caught up, then writes `Replication caught up` into its info message.
- **A slow peer.** A tick that falls while the previous one is still running is skipped. The peer's `timeout` has to cover one batch: the far side writes a whole batch before it answers.
- **Rows keep their timestamps**, in UTC as the local logger wrote them.
- **Write the prefix in capitals.** The local logger stores each code in capital letters, and the replicator looks its codes up the same way, so `variables` may be written in any case. The prefix is sent exactly as written: a prefix in lower case stores rows that the far side's history functions, which look a variable up in capitals, never find. Before engine 6.18.8 a code in lower case found no rows here and replicated nothing; from 6.18.8 on it replicates, and its whole history is sent once, since the far side holds none of it.
- **The prefix names the series.** One target can gather the same codes from several stations, each with its own prefix. Changing the prefix starts new series on the far side, and the whole history is sent again under the new names.

A dedicated target logger

The far side resumes each series from the latest row it holds of it. If the target logger also logs a variable of the same name as it runs, for example one that a DOPUSH from the same station keeps up to date, its latest row is always a live one: the replicator sends only what follows it, and the rows written while the link was down never arrive.

> **Important**
>
> Replicate into a LOGGER that is used for nothing else: enabled, with an empty `variables` list. Use a prefix when several stations send the same codes.

Replication and the logger's History section

The source logger records the head each replicator's receiver acknowledges, and its History section works around it:

- **Compaction stays behind the receiver.** A compaction of the source stops strictly before every acknowledged head, so the receiver always gets raw rows. A variable a replicator reads but has no head for yet is left out, and a disabled replicator holds compaction back too.
- **Deleting unsent rows** at the source loses them at the receiver too; the preview counts them.
- **Restoring a period the receiver already has** does not reach it: restore there too.
- **A receiver compacted ahead of its senders** gets replicated rows inside its compacted periods, and says so with `{n} replicated row(s) arrived inside a compacted period of {variable}`. Compact a receiver only well behind its replication lag.
- **While the source reclaims space**, its write lock is held and replicated batches into it are answered busy; the sender tries again at its next tick.

Messages

When	Level	Text
Opening: the CRON is empty. The module does not open	`ERROR`	`No schedule given`
Opening: the CRON cannot be read. The module does not open	`ERROR`	`Invalid schedule '{cron}': {reason}`
No local logger has the code in **Logger**	`ERROR`	`Logger '{logger}' is not defined`
No loaded peer has the code in **DataOrchester**	`ERROR`	`Peer '{peer}' is not defined`
The same, while other peers are loaded: `{peers}` lists their codes	`ERROR`	`Peer '{peer}' is not defined (known: {peers})`
The peer is disabled	`ERROR`	`Peer '{peer}' is disabled or has no secret`
The peer has no logger with the code in **Target**	`ERROR`	`Target logger '{target}' not found on the peer`
The target logger is disabled	`ERROR`	`Target logger '{target}' is disabled on the peer`
The far side refused the request	`WARNING`, then `ERROR`	`Peer '{peer}' refused the request: {reason}`
The far side could not be reached, is not running, or failed to answer	`WARNING`, then `ERROR`	`Peer '{peer}' unavailable: {reason}`
Some series could not be copied in a tick	`WARNING`	`{n} of {total} variable(s) delivered`
Rows are still waiting after a tick	`WARNING`	`Replication behind: more than {n} row(s) pending`
The backlog is worked off	`INFO`	`Replication caught up`

- The backlog message names the batch size: `Replication behind: more than 500 row(s) pending`.
- An empty **Target** is refused by the far side, with the reason `BAD_REQUEST`.
- A batch the far side refuses, or that does not reach it, is written to the log and sent again at the next tick.
- **One report per tick.** When no series got through, the module reports the last failure, counted once, so a far side that refuses every batch takes it to `WARNING` and, after `errorAfterFailures` ticks, to `ERROR`. When some got through, it warns `{n} of {total} variable(s) delivered`. Before engine 6.18.8 the failure showed only until the tick ended.
- A tick that copies everything clears the errors and warnings of the ticks before it.

Example

The station copies its level, flow and pump status to the control room every five minutes:

```
Logger          STATION_LOG
DataOrchester   CONTROL_ROOM
Target          REPLICA
Prefix          ST2_
CRON            0 0/5 * * * ?
Batch           500
Variables       LIT_301, FIT_501, P101_RUN
```

On the control room, a LOGGER with the code `REPLICA`, enabled and with no variables of its own, receives the rows as `ST2_LIT_301`, `ST2_FIT_501` and `ST2_P101_RUN`, and a formula there reads the station's level with `history(REPLICA, "ST2_LIT_301", addHours(now(), -24), now())`. A second station replicating into the same logger with the prefix `ST3_` keeps its series apart.

Next steps

- Connectivity
- Logger
- Settings every module shares
- Module messages
- Module catalogue

This page describes Data Orchester engine 6.21.0.