Process Control

Romain De Groote

16 October 2026

One small program, rebuilt step by step from direct method calls to an Event-Driven Architecture over MQTT: what moves at each step, and who ends up knowing whom.

Table of contents

About these notes **5**
How to read the diagrams . . . 5

1 Step 0 — What this is about **6**
1.1 The three questions . . . 6

2 Part I — Start with what you already know **7**
2.1 Step 1 — Rock bottom: just call it . . . 7
2.2 Step 2 — Three egoistic capabilities . . . 8
2.3 Step 3 — Give every capability a mailbox . . . 8
2.4 Steps 4–6 — The three capabilities become egoistic . . . 8
2.5 Step 7 — Enter the mediator . . . 9
2.6 Step 8 — Ask again: who knows whom? . . . 10
2.7 Step 9 — Why virtual threads? . . . 11
2.8 Step 10 — Keep the services egoistic . . . 11
2.9 What part I moved . . . 11

3 Part II — Cross the process boundary **12**
3.1 Step 11 — Separate the components . . . 12
3.2 Steps 12–13 — A directory becomes a mailbox . . . 12
3.3 Step 14 — The first distributed-systems problem: publication . . . 13
3.4 Step 15 — Observe a mailbox with `WatchService` . . . 13
3.5 Steps 16–17 — Services and mediator become processes . . . 13
3.6 Step 18 — Run the fully external local version . . . 14
3.7 What part II moved . . . 15

4 Part III — Make it actually distributed **17**
4.1 Syncthing replicates; it does not deliver . . . 17
4.2 Test one layer at a time . . . 17
4.3 What one keystroke costs . . . 17
4.4 Break a mediator . . . 18
4.5 The mediator becomes an ordinary service . . . 18
4.6 What part III moved . . . 19

5 Part IV — Rewind **20**
5.1 The obvious shortcut, and its trap . . . 20
5.2 Capability is not topology . . . 20
5.3 What changes in the code . . . 20
5.4 `ROUTE` and `TOPIC` are Intent . . . 21

6 Part V — Replay the dynamic idea across processes **22**
6.1 One blocking watch is not enough . . . 22
6.2 The helper: `FileSubscriptions` . . . 22
6.3 Two publications instead of four . . . 23

7 Part VI — Replay it over Syncthing, and name the messages **24**
7.1 The distributed dynamic run . . . 24
7.2 What "direct" means, exactly . . . 24
7.3 Intent, Status, Event . . . 24
7.4 The one-writer rule . . . 25

8 Part VII — Now merge the two architectural ideas **26**
8.1 A transformer earns its existence . . . 26
8.2 The transformer is egoistic too . . . 26
8.3 The complete filesystem architecture . . . 27

9 Part IX — MQTT **28**
9.1 What the filesystem taught, and what it is not for . . . 28
9.2 MQTT should look familiar . . . 28
9.3 The library, the client id and the broker address . . . 29
9.4 Structured Intent, and two object mappers . . . 29
9.5 DUISE, and rewiring a running system . . . 30
9.6 What a new subscriber learns, and what a dead one admits . . . 30
9.7 What survived the transport, and what coupling remains . . . 31
9.8 The whole journey . . . 31
9.9 The sentences to remember . . . 32

Appendices **33**

Glossary **33**

About these notes

These notes follow one small program through a long series of rebuilds, from direct method calls to an event-driven architecture over MQTT. Every rebuild leaves the program doing exactly what it did before, and changes who knows about whom.

The program under construction never changes: you type a line, it comes back rotated by 13 letters.

`keyboard` → `ROT13` → `console`

What changes is everything around it. The same system is rebuilt again and again, and each rebuild asks three questions, in this order:

> **Who knows whom?**
>
> **Who owns what?**
>
> **Who decides how the system is wired together?**

One summary is added here each time a step is finished, so the book grows at the pace of the work. Each summary says what the step did, what was observed when it ran, and what moved architecturally — usually a piece of knowledge that changed hands.

Based on material by Reto Konig, University of Fribourg.

How to read the diagrams

Every diagram exists once as a drawing and three times as a language. In the web version, a diagram that carries a message flow has a small play bar under it: **Play** walks one message through the whole chain, **One hop** advances a single step, and **Reset** puts it back. Boxes light up as the message reaches them, so you can see which component is waiting and which one is working.

In the PDF, and in a browser with JavaScript switched off, the same picture is there — just still. Nothing in the notes depends on the animation.

1 Step 0 — What this is about

Step 0 contains no code. It states the subject: not ROT13, but the architecture around it. The functionality stays almost unchanged for 81 steps, which is what makes the changes around it visible.

Two rules are set here and hold all the way through:

Meet each solution only after you have experienced the problem it solves.

That is why the build-up does not start with MQTT, and not with dynamic subscriptions either. A system with deliberately awkward mediators is built first, put into operation, and only then asked whether those mediators have earned their place.

The second rule is about vocabulary: *Intent*, *Status/State* and *Event* are named only after they have been used. They appear around step 42, not before.

The planned route, from step 0:

```
direct method calls → egoistic services → mediators → separate processes
→ filesystem as transport → Syncthing → "why all these mediators?" → rewind
→ services subscribe themselves → dynamic wiring → across processes
→ across Syncthing → when a mediator IS useful → MQTT
```

1.1 The three questions

The wording of the three questions matters, and it is easy to replace it with something weaker:

As asked here	A weaker version
Who **knows** whom?	who *communicates* with whom
Who **owns** what?	which *data* each part holds
Who **decides** how the system is wired?	which part *connects* everything

"Knowing" is not "communicating": two components can exchange messages all day without either one naming the other. "Deciding the wiring" is not "connecting": the part that forwards messages need not be the part that chose the route. Both distinctions become concrete later — the first already in step 1, the second in part III.

2 Part I — Start with what you already know

Steps 1 to 10 stay inside one JVM. The system is built three times: as direct calls, as three capabilities that refuse to know each other, and as those same capabilities glued together by mediators. Nothing is distributed yet, and that is the point: the only thing that changes is who knows whom.

2.1 Step 1 — Rock bottom: just call it

`ProceduralRot13.java` is the whole system in one compact source file: a loop that reads a line, and one statement that prints the rotated text.

```
void main() {
    while (true) {
        var text = IO.readln();

        if (text == null)
            return;

        IO.println(rot13(text));
    }
}
```

`IO.readln()` stands on its own line, but `rot13(text)` is not a step of its own: it is called *inside* the argument of `IO.println`. Neither `rot13` nor `IO.println` knows anything about the other. The one that knows both is `main()`, which names them and fixes their order.

What `main()` knows about `rot13`: that it exists, how to invoke it (a bare name, both live in the same implicitly generated class), what argument it takes, what it returns, that it is available *right now*, and that execution continues after it returns. It does not know how `rot13` is implemented — that is the one thing it can ignore.

The availability guarantee is worth naming, because it disappears later: `main()` and `rot13` are two methods of one already-loaded class, called on the same thread, in the same process. Nothing has to be started first, no message travels anywhere, no listener has to be subscribed.

Observed: typing a line prints it rotated; `Ctrl+D` ends the program silently, because `IO.readln()` returns `null` and `main()` returns.



Figure 2.1: The procedural version. The wiring is written into the code: `main()` names all three parts and fixes their order.

2.2 Step 2 — Three egoistic capabilities

An **egoistic** component says: I know what I can do, what data I accept and what data I produce — and I would rather not know the business process around me. Concretely, `ConsoleIn` must not contain `rot13(...)`, and `ROT13` must not contain `consoleOut(...)`.

Removing those calls immediately creates a problem, best put as a question: if nobody knows anybody, **who makes this into a system?** Note what the question does *not* say: the knowledge is not gone, it has to go somewhere else. Two connections are missing, not one — `ConsoleIn` ↔ `ROT13` and `ROT13` ↔ `ConsoleOut`.

2.3 Step 3 — Give every capability a mailbox

The smallest mechanism for the missing connections: a message and a named queue.

```
record Message(String type, String value) {}

Map<String, BlockingQueue<Message>> mailboxes = new ConcurrentHashMap<>();

BlockingQueue<Message> mailbox(String name) {
    return mailboxes.computeIfAbsent(name, _ -> new LinkedBlockingQueue<>());
}
```

These are **mailboxes**, not topics, and the distinction is worth insisting on: if two readers call `take()` on the same mailbox, they compete — each message goes to exactly *one* of them and is not copied to both. A topic, in Publish/Subscribe, would hand every subscriber its own copy. One reader per message is enough for now, and the limitation is kept on purpose: it is what makes it clear later what Publish/Subscribe actually adds.

2.4 Steps 4–6 — The three capabilities become egoistic

Each capability now touches only its own mailbox names:

Capability	reads	writes
ConsoleIn	the keyboard	`consoleIn.outbox`
ROT13	`rot13.inbox`	`rot13.outbox`
ConsoleOut	`consoleOut.inbox`	the console

No capability contains the name of another, and that is exactly the result: three nicely isolated capabilities that do absolutely nothing together. Nothing moves a message from `consoleIn.outbox` to `rot13.inbox`, or from `rot13.outbox` to `consoleOut.inbox`.

2.5 Step 7 — Enter the mediator

The smallest possible glue: a loop that takes from one mailbox and puts into another, knowing nothing about what travels through it.

```
void mediate(String from, String to) {
    while (true)
        mailbox(to).put(mailbox(from).take());
}
```

`main()` starts five virtual threads: the three capabilities, plus two mediators whose routes are given as arguments — `consoleIn.outbox` → `rot13.inbox` and `rot13.outbox` → `consoleOut.inbox`.

Observed: `Hello World` in, `Uryyb Jbeyq` out, exactly as in step 1. But `Ctrl+D` no longer ends the program. ConsoleIn's thread *returns* (it does not loop), and the other four threads sit blocked in `take()` on empty mailboxes, using no CPU. What actually keeps the process alive is the last line of `main()`, `Thread.currentThread().join()`: the main thread waits for itself, which can never finish. Virtual threads are daemon threads, so those four waiting threads would not have kept the JVM running on their own.

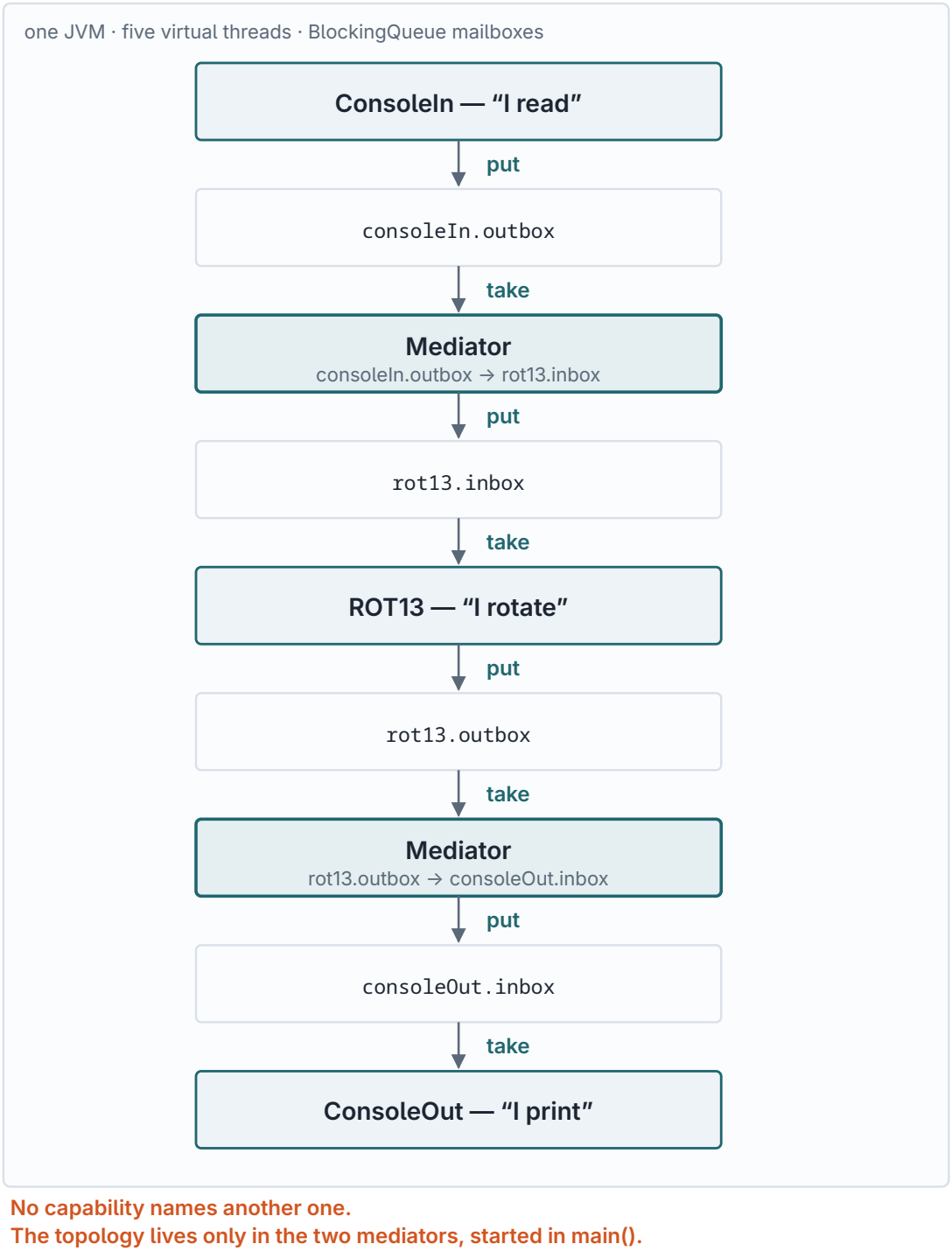


Figure 2.2: The mediated version inside one JVM. In the web version, **Play** walks one message through all eight hops.

2.6 Step 8 — Ask again: who knows whom?

Does ConsoleIn know ROT13? No. Does ROT13 know ConsoleIn or ConsoleOut? No. Has the topology disappeared? No — it sits in the two `mediate(...)` calls inside `main()`.

This is the first major result:

Decoupling often does not make knowledge disappear. It moves knowledge to where it belongs.

The capability "ROT13 transforms text" belongs to ROT13. The topology `ConsoleIn` → `ROT13` → `ConsoleOut` belongs to system composition. What changed since step 1 is not that the wiring left `main()` — it is still there — but that it is no longer written *inside* a capability. It is now data: two pairs of mailbox names. Rewiring touches those two lines and no capability at all.

2.7 Step 9 — Why virtual threads?

Five loops now wait indefinitely: the keyboard, `rot13.inbox`, `consoleOut.inbox` and the two mediators' sources. Virtual threads are simply the cheap Java way to let five waits happen at once inside one JVM.

They are an implementation convenience, not an architecture. The architecture — who knows whom, and where the wiring lives — is in the mailbox names, not in the threads. The evidence for the separation: the same architecture will run as separate processes in step 11, with no threads shared between the parts.

2.8 Step 10 — Keep the services egoistic

The obvious shortcut: the first mediator only copies messages unchanged, so why not delete it and let ROT13 `take()` from `consoleIn.outbox` directly?

Because ROT13 would then contain the name `consoleIn.outbox`, which means it would know that ConsoleIn exists and where it publishes. The topology would move *into* ROT13, and ROT13 would stop being an egoistic capability. The test that makes this concrete: suppose the input comes from a new `FileIn` capability publishing to `fileIn.outbox`.

	with the mediator	with the shortcut
what changes	one argument of the mediator	`runRot13` itself
who is touched	no capability	ROT13, whose job did not change

The price is worth stating plainly: this design may look indirect — even a little odd. There is another way to keep services egoistic *without* a mediator in every message's path, and it comes later.

2.9 What part I moved

stage	where the topology lives	who could rewire it
step 1	inside the capability code (`IO.println(rot13(text))`)	whoever edits that code
steps 7–8	in `main()`, as pairs of mailbox names	whoever edits `main()`

Functionality: unchanged. Lines of code: more. Parts that know each other: none. That is the trade everything that follows keeps testing.

3 Part II — Cross the process boundary

Steps 11 to 18 take the architecture of part I and split it across five operating-system processes on one computer. The capabilities keep their mailbox names; only the transport underneath changes, from Java objects to files in directories. The first genuinely distributed problem shows up here, and it has nothing to do with ROT13.

3.1 Step 11 — Separate the components

Three processes cannot share a `BlockingQueue`: every `java` command starts its own JVM with its own memory, so the `mailboxes` map would exist three times and each copy would see only its own messages. An external transport is needed.

TCP would do, MQTT would do. The most primitive option is picked deliberately, the **filesystem**, for a one-line reason: *because you can see it.* Messages become files you can list with `ls` and read with `cat` while the system runs. Primitive here does not mean inferior; part VIII comes back to what the filesystem is genuinely good and bad at.

3.2 Steps 12–13 — A directory becomes a mailbox

The mapping is one line long:

```
directory = mailbox        file = message
publish   = create file    receive = watch directory
```

```
messageBoard/
├── consoleIn/outbox/
├── rot13/inbox/
├── rot13/outbox/
└── consoleOut/inbox/
```

The `Message` record gains an optional timestamp, and the file format stays two lines plus an optional third:

```
TEXT
Hello World
TIMESTAMP: 2026-09-22T10:42:12.123Z
```

A message written by hand may leave the timestamp out. The `TIMESTAMP:` marker keeps a payload line that happens to look like a date from being mistaken for metadata; in this tiny format, a final line starting with `TIMESTAMP:` is reserved.

Jackson could be introduced here and deliberately is not. The point of a ridiculously simple format is that almost nothing hides the architecture — the whole message is readable in any text editor, which is what keeps the attention on who talks to whom.

3.3 Step 14 — The first distributed-systems problem: publication

A reader watching a directory can discover a file *before the writer has finished writing it.* Hence a publication protocol:

1. create a private temporary file;
2. write the complete message;
3. close it;
4. atomically rename it to its final `.msg` name;
5. consumers react only to final files.

```
Files.writeString(temporary, content);
Files.move(temporary, published, StandardCopyOption.ATOMIC_MOVE);
```

Both halves are needed. The rename is atomic — a reader sees either no `.msg` file or the complete one — but it protects nothing if readers also pick up `.tmp` files. `.tmp` and `.msg` stay in the same directory so the move cannot cross filesystems, where atomicity would be lost.

> **Writing and publishing are not the same operation.**

3.4 Step 15 — Observe a mailbox with `WatchService`

Java's own filesystem observer supplies the consumer half: register a directory for `ENTRY_CREATE`, block in `watcher.take()`, and ignore every filename that does not end in `.msg`.

That also gives a low-tech injection tool. With `ActuatorService` running as the consumer:

```
printf 'TEXT\nHello from a file' > messageBoard/consoleOut/inbox/manual.tmp
mv messageBoard/consoleOut/inbox/manual.tmp messageBoard/consoleOut/inbox/manual.msg
```

Observed: nothing happens when the `.tmp` file appears; the line is printed the moment the rename creates the `.msg` name. Both accepted formats work, with and without the trailing timestamp.

The architectural point reads `producer` → `filesystem` → `consumer`: the consumer could not tell a human typing `mv` from a real service. It knows its own inbox, not who writes into it. Egoism survived the process boundary.

3.5 Steps 16–17 — Services and mediator become processes

The three capabilities become three programs, and the mediator a fourth kind of program:

program	plays	reads	writes
`SensorService.java`	ConsoleIn	the keyboard	`consoleIn/outbox`
`ProcessorService.java`	ROT13	`rot13/inbox`	`rot13/outbox`
`ActuatorService.java`	ConsoleOut	`consoleOut/inbox`	the console
`Mediator.java <from> <to>`	the glue	`<from>`	`<to>`

The mediator is generic: its route is not compiled in, it arrives as two startup arguments. But there is an asymmetry worth keeping in mind: the three capabilities receive their **work** as messages in a mailbox, while the mediator receives its **route** as startup configuration, and it has no name and no control mailbox of its own that anyone could write to.

The practical consequence: arguments are read once, so changing the route of a running mediator means stopping it and starting it again. A configuration file would be no different — it is still read at startup. This stays deliberately unfixed until the cost of the distributed mediators has been felt.

3.6 Step 18 — Run the fully external local version

Five terminals, five processes, one local filesystem between them:

```
java SensorService.java                                  # terminal 1
java Mediator.java consoleIn/outbox rot13/inbox          # terminal 2
java ProcessorService.java                               # terminal 3
java Mediator.java rot13/outbox consoleOut/inbox         # terminal 4
java ActuatorService.java                                # terminal 5
```

Observed: `Hello World` typed in terminal 1 appeared as `Uryyb Jbeyq` in terminal 5. The three middle terminals showed nothing new — each mediator prints one "Mediating: …" line at startup, and `ProcessorService` prints nothing at all. Afterwards each of the four mailbox directories held one `.msg` file: the same line, copied along the chain.

Count it: **five processes** for what step 1 did in one statement. This is not the moment to optimise: it is not meant to be comfortable yet.

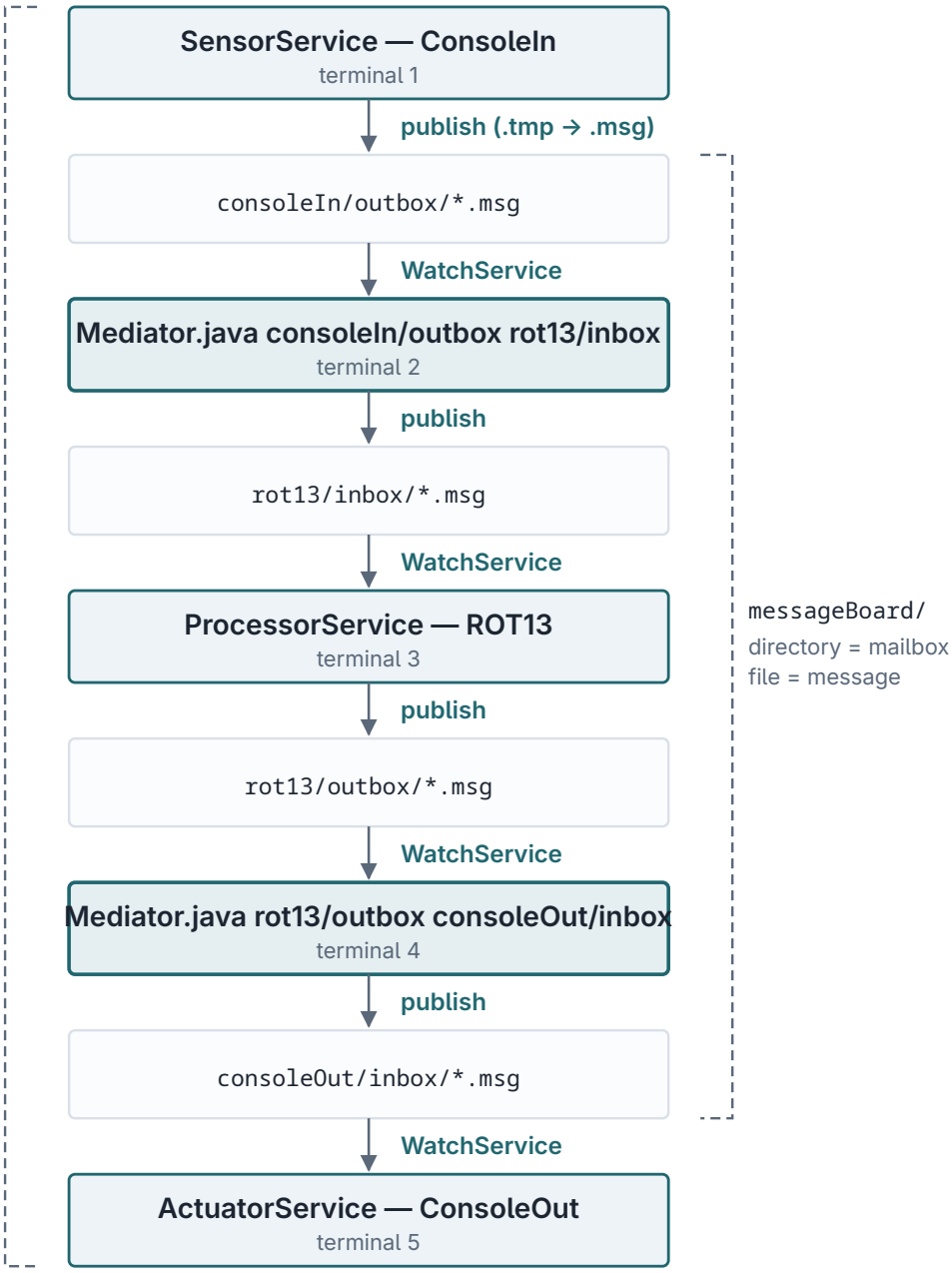


Figure 3.1: The five processes and the four mailbox directories. In the web version, **Play** follows one line of text from the keyboard to the console.

3.7 What part II moved

stage	where the topology lives	who could rewire it
step 1	inside the capability code	whoever edits that code

stage	where the topology lives	who could rewire it
steps 7–8	in `main()`, as mailbox names	whoever edits `main()`
step 18	in the two `Mediator.java` `<from> <to>` command lines	whoever starts the processes

The topology has now left the source code altogether: no `.java` file contains the route. It lives in what the operator typed, which means a person is currently the orchestrator. What the five-process version bought is real — the wiring can change without touching any program — and what it cost is equally real: five processes, four copies of every message, and a program that no longer stops when you press `Ctrl+D`.

4 Part III — Make it actually distributed

Five processes on one computer still share one filesystem. Steps 19 to 27 put them on two machines, replicate the message board between them with Syncthing, measure what that costs, and then break a mediator on purpose. By the end, the mediators' price is a number rather than an opinion.

4.1 Syncthing replicates; it does not deliver

Syncthing keeps a directory replicated between machines. That is all it is asked to do here. It knows nothing about Intent, Status, Event, ROT13 or mediators, and it has no concept of a consumer: no subscriptions, no topics, no delivery guarantees, no notion that anybody is waiting.

The trigger is still ours. Each service runs its own `WatchService` on a **local** directory; Syncthing only makes files appear there. That is why none of the three services changed a single line when the system became distributed.

Two consequences follow, and both show up later as observations rather than warnings:

- replication is **asynchronous** — the `.tmp` → `.msg` rename is atomic on each machine separately, and nothing promises ordering or atomicity between machines;
- the filesystem is a serious transport with its own strengths: files can be large, replicas survive a reboot, and every message is readable with `cat`.

4.2 Test one layer at a time

Before any Java touched the replicated folder, the dullest possible test ran first: write `hello.txt` on one machine, confirm it appears on the other, delete it, confirm the deletion travels back. Both directions worked.

That order matters. When the distributed run later failed, the question "is it replication or is it my program?" had already been answered for the transport, and the search could start one layer up.

4.3 What one keystroke costs

With the sensor on one machine and the mediators, ROT13 and the actuator on the other, one typed line produced **four** `.msg` files, one per mailbox:

mailbox	content	timestamp inside
`consoleIn/outbox`	`Hello World`	23:02:05.315
`rot13/inbox`	`Hello World`	23:02:05.315
`rot13/outbox`	`Uryyb Jbeyq`	23:02:13.471
`consoleOut/inbox`	`Uryyb Jbeyq`	23:02:13.471

The repeated timestamps are the evidence that two of those four publications are **pure copies**: the mediators republish the message unchanged, timestamp included. They compute `identity(x) = x`, as separate distributed processes.

The timing, taken from the four files' modification times on one machine so that no clock skew enters: the sensor's file appears at `01:02:05.432`, the other three at `01:02:13.460`, `.504` and `.532` — **8.10 s** later, and within **72 ms of each other**. So the chain's own work took milliseconds; the eight seconds are Syncthing's propagation. The same chain, run entirely on one machine, completes in about 24 ms.

4.4 Break a mediator

Stop the first mediator and type two lines. ConsoleIn is alive, ROT13 is alive, ConsoleOut is alive, Syncthing is alive — and the business flow is dead. The two lines pile up in `consoleIn/outbox`; nothing moves past it. The mediators are part of the availability chain.

Then restart the mediator. **Nothing is delivered.** `FileMessageBoard.watch` registers a directory for `ENTRY_CREATE` and blocks in `watcher.take()`; it never lists what is already there. The two files are still on disk, in both replicas — so no data was lost. What has no memory is the *subscription*.

That distinction is worth keeping: **the messages survived, the delivery did not.** A broker answers this with session state and retained messages; in our own code it would be one line — list the existing `.msg` files before entering the watch loop — which the material deliberately does not write yet.

4.5 The mediator becomes an ordinary service

Before removing the mediators, one asymmetry gets removed: the services receive their *work* as messages, while the mediator received its *route* as shell arguments. So the mediator is given only a name, `ConfigurableMediator mediator1`, from which it derives one permanent control mailbox, `mediators/mediator1/inbox`, and the route arrives as a message:

```
ROUTE
consoleIn/outbox
rot13/inbox
```

`from` and `to` travel in **one** message on purpose: as two, the mediator would briefly hold a new source with the old destination and forward to the wrong place. A route is one fact, and one fact belongs in one message.

A short-lived orchestrator publishes both routes and exits, so the topology is visible in one place and is **not** in the data path. Running it after the services, and typing a line before it, shows both halves at once: the early line is stranded, the later one flows.

What improved: the mediator now has an identity, takes configuration through its own inbox, keeps a generic capability, and gets its topology from an orchestrator. What did not improve: the data path. It is still `event` → `mediator` → `copied event`, still a process in the availability chain, still copying every message without adding meaning. And this version accepts exactly **one** route: `waitForRoute(...)` returns, then the mediator enters `watch(route.from(), …)` and never reads its control inbox again. Live rewiring is not promised, because one blocking `watch` cannot serve a control inbox and a source at the same time — the reason a subscription helper appears later.

4.6 What part III moved

stage	where the topology lives	in the data path?
step 18	the operator's command lines	the mediators are
step 27	an orchestrator, published as Intent	the mediators still are

The question the part ends on is therefore sharp rather than rhetorical: **why can't the service interested in the data simply listen to it itself?**

5 Part IV — Rewind

Steps 28 to 34 go back to one JVM on purpose. With no replication delay and no five-terminal setup, the only thing that can change is who knows whom — so the idea can be tried in seconds instead of in a distributed debugging session.

5.1 The obvious shortcut, and its trap

If the first mediator only copies messages, delete it and let ROT13 read the source itself:

```
listen("consoleIn.outbox");
```

The mediator is gone. But ask the recurring question: ROT13 now contains the name of ConsoleIn's mailbox. The topology moved out of the mediator and **into the service**. As the material puts it: "We removed the mediator… and put topology back into the service. *Oops.*"

5.2 Capability is not topology

ROT13 may know two things: *I can transform text*, and *I am capable of listening for compatible text messages*. It may **not** know *my input always comes from consoleIn.outbox* — that is deployment topology.

So the name is not compiled in; it arrives at runtime:

```
TOPIC consoleIn.outbox
```

ROT13 learns its wiring from data. Note who sends it: in `DynamicQueueRot13.java` it is `main()` that publishes both `TOPIC` messages — a configurator, outside the data path, in the role the orchestrator played before. It cannot be ConsoleIn: if ConsoleIn told ROT13 what to subscribe to, ConsoleIn would have to know ROT13 exists, and the topology would simply have moved into a different capability.

5.3 What changes in the code

`listen(...)` is nothing exotic: a virtual thread that repeatedly calls `take()` on one of the same queues. `subscribe(...)` starts one such thread per subscribed name. The consequence is visible in the mailboxes:

- ROT13 takes **straight from `consoleIn.outbox`** — the name it was told;
- `rot13.inbox` is no longer a payload mailbox. It carries control messages, and ROT13's loop reads it to decide whether a message is a `TOPIC` to act on.

Nothing writes into `rot13.inbox` to move data anymore. The mailbox did not disappear; it changed job, from carrying payload to carrying Intent.

One observation from running it: after `Ctrl+D`, output stops but the program stays up. Nothing was interrupted — `runConsoleIn` returns on end of input, so the **producer** is gone, while both subscriber threads sit blocked in `take()` on sources that will never publish again. The JVM stays alive because `main` ends with `Thread.currentThread().join()`. "The flow stopped" always has to be traced to *which* participant stopped.

5.4 `ROUTE` and `TOPIC` are Intent

Apply two tests to `TOPIC consoleIn.outbox`. Does it describe something that happened? No. Something currently true? No. It says *please begin listening here* — desired behaviour, which is **Intent**. The mediator's `ROUTE` is the same kind of message: *please forward from here to there.*

One trap worth naming, because it is tempting in a defence: *publishing* an Intent is naturally something that happened. Classification follows what the message says, not the fact that it exists.

And one condition, which returns later: the dynamic version is **not** automatically superior. It makes sense only if the consumer actually understands the producer's message contract.

```
STATIC MEDIATED                  DYNAMIC DIRECT
Producer → Mediator → Consumer   Intent → (Producer ⟶ Consumer)
```

6 Part V — Replay the dynamic idea across processes

Steps 35 to 38 take the subscription idea out of the single JVM. The services become separate programs again, but instead of mediators copying files, each service watches the mailbox it was *told* to watch. One assumption of the first filesystem helper does not survive the move, and fixing it is what this part is really about.

6.1 One blocking watch is not enough

`FileMessageBoard.watch(topic, receiver)` registers **one** directory and blocks forever. That was enough while every process watched one fixed mailbox. A dynamically wired service needs more:

```
always watch its own control inbox
AND
add a watched directory at runtime
AND
possibly remove it later
```

The old signature cannot express that, and the consequence was already visible earlier: the configurable mediator accepted exactly one `ROUTE`, because the moment it began watching its source it went deaf to its own control inbox. The limitation was never *whose* directory — it was *one, fixed, forever*.

6.2 The helper: `FileSubscriptions`

One `WatchService` can serve many directories. `register(...)` returns a `WatchKey` identifying one of them, `watcher.take()` returns whichever key has events, and `key.cancel()` deregisters it. The helper keeps both directions, `name` → `key` and `key` → `name`, so an event can be traced back to the mailbox it came from.

Events do not reach a callback directly. They go onto a queue, read with `take()`, carrying

```
record IncomingMessage(String mailbox, Message message) {}
```

Two reasons, and order of arrival is only the third. First, the observing thread must stay responsive for *every* subscribed directory: a callback that transformed, published or blocked would stall the observation of all the others. Second, the mailbox name travels with the message, so **one** loop in the service can decide what a message means — control inbox, so this is Intent to act on; subscribed source, so this is data to transform — and that same loop is free to add or remove subscriptions, because it is not sitting inside a callback.

6.3 Two publications instead of four

Run `SensorService`, `DynamicProcessorService` and `DynamicActuatorService` — and no mediators — then let `Configure.java` publish the two `TOPIC` messages. A line typed afterwards leaves exactly two application publications in the board:

mailbox	content	what it is
`consoleIn/outbox`	`TEXT hello`	the input
`rot13/outbox`	`TEXT uryyb`	the result
`rot13/inbox`	`TOPIC consoleIn/outbox`	Intent, published once
`consoleOut/inbox`	`TOPIC rot13/outbox`	Intent, published once

`consoleOut/inbox` holds **no payload at all**: ConsoleOut takes straight from `rot13/outbox`. Its inbox changed job, from carrying messages to carrying wiring.

A line typed *before* `Configure.java` ran stays in `consoleIn/outbox` forever. A subscription that does not exist yet misses what came before it — exactly what the restarted mediator showed when it delivered nothing.

version	application publications	processes in the data path
mediated (step 18)	4	5
configurable mediator (step 27)	4	5
dynamic (step 38)	**2**	**3**

7 Part VI — Replay it over Syncthing, and name the messages

Steps 39 to 47 distribute the dynamic version across two machines, say precisely what "direct" means, and then — only now, after the ideas have been used — give the three kinds of message their names: Intent, Status, Event.

7.1 The distributed dynamic run

Sensor on one machine, ROT13 and the actuator on the other, no mediator anywhere. After the two `TOPIC` Intents had replicated, a typed line produced the result on the other machine — and the board shows what that cost:

- `consoleIn/outbox`: the input;
- `rot13/outbox`: the result, written by the service that produced it;
- `consoleOut/inbox`: **Intent only**, where the mediated version held a full copy of every message.

The timing was the interesting part. The input was published at 02:35:12 and the results appeared on the first machine at 04:11:28.797 and .893 — 96 ms apart, long after the input. Syncthing had been saturated overnight, so the batch arrived when the queue drained. The chain was correct throughout; the transport's latency is simply not something the architecture controls. Compare 24 ms for the same chain on one machine.

7.2 What "direct" means, exactly

Two hops that look alike are architecturally opposite:

- a **transport hop** — Syncthing on each side, possibly a relay in between — adds no participant that knows the topology and nothing reacts to it. If the packets detoured through another continent, no claim here would be contradicted.
- an **architectural hop** — a mediator copying `consoleIn/outbox` → `rot13/inbox` — adds a component that must know both names, sits in the availability chain, and republishes the message.

So the defensible claim is: *ROT13 reacts to ConsoleIn's publication with no component of ours in between, and without containing ConsoleIn's identity.* Not: *the data goes straight from A to B.* The architecture controls which messages a service reacts to, not the transport's routing.

7.3 Intent, Status, Event

Three kinds of message, named only after being used:

	says	our examples
E — Event	something **happened**	`TEXT hello` as `ConsoleIn/E/text`; `TEXT uryyb` as `ROT13/E/crypted`
S — Status/State	something is **currently true**	`ROT13/S/subscriptions` — not published by our system yet
I — Intent	what **should** happen	`TOPIC consoleIn/outbox`, `ROUTE … …`

Past tense is a useful hint for Events. For Intent the test is two questions: does the message report something that happened? No. Something currently true? No. Then it requests behaviour.

One trap: *publishing* an Intent is naturally something that happened. Classify by what the message says, not by the fact that it exists — otherwise every message becomes an Event and the vocabulary stops being useful.

Status deserves its own note, because the instinct is to **ask** for it: "what are your subscriptions?" But asking is a call, and a call needs the asker to know the component, to know it is up, and to wait. Status is **published**: ROT13 publishes `ROT13/S/subscriptions` whenever its wiring changes, and anyone interested reads it. The reader then needs a topic, not a component, and ROT13 never learns who audits it. Each kind answers a different question:

- an Event is only useful if you were listening — our stranded messages, three times over;
- a Status is what a newcomer needs: the current value, without replaying history;
- an Intent asks for a change.

And commands are not forbidden. Sending `ROT13/I` with `ACTION Hello` is command-like, and the material's verdict is "Fine." The component performs its capability and publishes its Events and its Status. The flow it endorses is `Intent` → `component` → `State + Events`.

7.4 The one-writer rule

> Every writable resource has exactly one writer authority.

Do not confuse that with the UUID in a message filename. The UUID prevents two publications from choosing the same physical name; writer authority says which component **owns** a logical Event or Status resource. Two writers never collide on a filename, and that is exactly why the problem is easy to miss.

What a second writer costs: the mailbox name was the only provenance marker, so `rot13/outbox` stops meaning "ROT13 transformed this" and a subscriber can no longer trust the contract it built on. For Status it is worse — two components each claiming to publish *the current value* of one fact simply contradict each other, with no ordering guarantee across machines to break the tie. For append-only Events many publications coexist by design; the question is semantic ownership, not collision. When two authorities appear, ask why there are two rather than reaching for a lock.

8 Part VII — Now merge the two architectural ideas

Two mediators were removed and the gain was measured. Steps 48 to 52 answer the obvious next question: does that make mediation wrong? No — and the distinction that decides it is sharp enough to apply to a design you have never seen.

8.1 A transformer earns its existence

What was removed was a mediator computing `A` → `A`: an expensive identity function, in the availability chain, copying every message without adding meaning.

Now take a producer publishing `{"value":"Hello"}` and a consumer expecting a display structure. A `TextToDisplayTransformer` between them contributes schema transformation, semantic adaptation, possibly unit conversion, possibly aggregation. That is work neither side should own: ROT13 is not meant to decide terminal colours, and the console is not meant to understand ROT13's payload shape.

The line is drawn by **contract compatibility**, not by whether something is added. A mediator converting Celsius to Fahrenheit changes rather than adds and has earned its place; one appending a timestamp to a message the consumer already understands has added something and is still suspicious. The question is always: could the consumer read the producer's message as it stands?

> If a consumer semantically understands a producer's publication, direct subscription is often sufficient. If the contracts are incompatible, introduce a transformer that performs an actual adaptation.
>
> A mediator that merely forwards compatible data is suspicious. A mediator that adds necessary semantics has earned its existence.

8.2 The transformer is egoistic too

Inserted into the chain, the transformer subscribes to `rot13/outbox` and publishes to **its own** outbox; ConsoleOut then subscribes to the transformer's outbox instead of ROT13's. Note what that costs in code: **nothing**. ConsoleOut receives a different Intent — `TOPIC display/outbox` rather than `TOPIC rot13/outbox` — and the chain is rewired. ROT13 never learns the transformer exists; the transformer never learns ConsoleOut exists.

Why not let it publish straight into `consoleOut/inbox`, saving a mailbox? Four reasons:

- **knowledge** — it would contain ConsoleOut's name, which is topology inside a service, the relapse of step 28;
- **ownership** — that inbox is ConsoleOut's control channel, written by the configurator; payload from the transformer would give it two authorities;
- **reach** — its own outbox can be read by a logger, a web view or a second display at no cost, while one inbox serves exactly one consumer;
- **mutability** — an aggregator can later be inserted after it by sending one different Intent, with no component edited.

8.3 The complete filesystem architecture

Every mailbox with exactly one writer:

mailbox	single writer
`consoleIn/outbox`	SensorService
`rot13/outbox`	ProcessorService
`display/outbox`	the transformer
every control inbox	the configurator — the one designated Intent authority
the terminal	ActuatorService, which writes no mailbox at all

Egoistic capabilities, Intent from one authority, a transformer only where contracts differ, one writer per resource, and the topology nowhere in the capabilities. That is the design the filesystem was good enough to teach — and the next part asks what the filesystem itself is no longer good enough for.

9 Part IX — MQTT

9.1 What the filesystem taught, and what it is not for

The filesystem taught publication boundaries, directories as addresses, observation, independent processes, independent languages, synchronisation latency, runtime wiring, State, Intent, Events, mediator costs, service ownership and transformer services. It was a good teacher and it is not an elegant message broker: by the end the architecture was clean and the transport was a pile of `.tmp` files, atomic moves, rescanning, shared paths and replication delay.

The conclusion is **not** "files bad, MQTT good". The filesystem has an advantage MQTT cannot match — it routinely stores gigabytes or terabytes, while brokers and clients are built for small messages, and chunking a huge payload over MQTT means inventing a file-transfer protocol inside a message protocol. So:

> Use messages for coordination and events. Use an appropriate file or object store for large data. A message can carry the **location, identity and metadata** of that data instead of the terabytes themselves.

A replicated filesystem also gives every machine a local copy, which is a genuine strength — though replication is not a backup: deletion and corruption replicate too, unless versioning preserves older copies.

9.2 MQTT should look familiar

filesystem / Syncthing	MQTT
synchronised message-board root	broker / namespace
directory path	topic
`.msg` file	publication
file contents	payload
`WatchService`	subscription
dynamic directory	topic / filter
control directory	`I`
state directory	`S/…`
event directory	`E/…`

The reaction the material is after: *"Oh. We already know this architecture."* MQTT is a transport change, not an architectural claim.

Two things a directory could not do, both visible in a minute at the command line. A publication made **before** a subscriber exists never reaches it — the stranded-message problem for the third time, now with an answer the filesystem never had. And a filter like `test/#` matches `test/a` and `test/a/b/c` but not `testing/x`, because `#` works on **levels**, not string prefixes; `test/+` would match only one level down.

The deeper correction: **a topic is not a container.** There is no object called `test/a` on the broker. A topic is a routing key carried by each publication and matched against the filters of whoever is subscribed at that instant. Directories *hold* messages; topics *match* them. One subscription therefore covers topics that do not exist yet — where a message board needed a new directory and a new registration.

9.3 The library, the client id and the broker address

Java has no MQTT client in its standard library, so Paho arrives now — after the need, like the mediator in step 7 and Jackson in step 64. Four things are needed from it: connect, publish, subscribe, callback. Connecting is deliberately anticlimactic: *"Nothing architectural happened here. We merely attached our process to the transport."*

Two details of connecting carry weight:

- **the client id** is not merely a uniqueness token. The broker recognises a returning client by it and keeps session state for it: subscriptions, messages queued at QoS > 0, and the association with its Last Will. A directory never knew who was watching, which is exactly why a restarted watcher's backlog was never replayed.
- **the broker address** is deployment configuration: every runnable reads `System.getProperty("mqtt.broker", "tcp://localhost:1883")`. It is also the one piece of configuration that cannot arrive as Intent, because Intent travels *through* the broker. Topology can be published; the address of the thing that publishes it cannot.

Subscribing costs one packet, and **the broker keeps the subscription list**. There is no standard way to ask it what another client subscribed to — which is why `S/subscriptions` exists: a component's own declaration of its wiring, published so the architecture can know what the transport keeps private. Only its owner may publish it, by the one-writer rule, because only its owner knows what it accepted.

9.4 Structured Intent, and two object mappers

`TOPIC some/path` has one field and one meaning. It cannot say which *kind* of Intent this is, carry a second parameter, nest, or be extended; and there is nothing for a receiver to validate against. Richer Intent wants structure:

```
subscribe:
  topic: Demo/ConsoleIn/U/alice/E/text
```

Only then does a serialisation library have a clear job. `JsonTools` keeps **two** mappers, and the asymmetry is deliberate:

- reading uses a **YAML** factory with `FAIL_ON_UNKNOWN_PROPERTIES` off — JSON is a subset of YAML, so hand-typed Intent is accepted, and a newer configurator can send fields an older service ignores;
- writing uses plain **JSON** with `INDENT_OUTPUT` on, so published payloads are strict and readable.

Tolerant in, strict out. A single shared mapper would have to give up one of those.

9.5 DUISE, and rewiring a running system

The naming scheme finally gets its name: **D**escription (what contracts this service understands and publishes), **U**nit (which instance), **I**ntent (requested behaviour), **S**tatus (what is true now), **E**vent (what happened).

```
Demo/ROT13/U/alice/{I, S/online, S/subscriptions/text, E/crypted}
```

`U/<unit>` is what the filesystem never needed: each machine had its own replica of the same relative path, so instance identity was implicit. MQTT has one shared namespace, so a topic must say *which* copy. Hard-coding `U/alice` in a capability is therefore worse than hard-coding `consoleIn.outbox` ever was: it binds the build to one deployment. Configured by Intent, **one build serves everyone** — which is why `MqttConfigure` takes three unit names as arguments. The unit names *are* the topology.

Rewiring, observed on a running system: publish a new Intent naming a different source topic, and the service unsubscribes from the old one, subscribes to the new one, and republishes `S/subscriptions` to match. No restart, no edit. Send an invalid filter instead — `Demo/#/E/text`, where `#` is not the last level — and `MqttTools.isValidFilter` rejects it; the old subscription stands and the Status still tells the truth.

That rejection exposes a real gap, worth naming rather than hiding: it is logged locally only. The configurator cannot tell whether its Intent was applied or refused, and a monitor cannot distinguish "refused" from "not yet processed". In this design a sender learns the outcome by reading Status and comparing it with what it asked for. Publishing refusals as an Event, or a last-error Status, would close that gap; neither is in the material, so it is a design decision to justify.

9.6 What a new subscriber learns, and what a dead one admits

Subscribing to `Demo/#` for three seconds, with nothing being published, returns this:

```
Demo/ROT13/U/nitro5/S/online              {"value":true}
Demo/ROT13/U/nitro5/S/subscriptions/text  "Demo/ConsoleIn/U/nitro5/E/text"
Demo/ConsoleOut/U/nitro5/S/online         {"value":true}
Demo/ConsoleOut/U/nitro5/S/subscriptions/text  "Demo/ROT13/U/nitro5/E/crypted"
```

A late arrival learns the entire current wiring of the system, instantly, without asking anyone and without any component knowing a monitor exists. That is retained **Status**. No Events arrive, because Events are published non-retained — "retained State, non-retained Events". It is also the proper answer to the instinct of *asking* a service what it is doing: nobody asks; the service states it, once, and the broker keeps it for whoever turns up.

Retained Status alone would lie, though. Kill a service with `Ctrl+C` and its `S/online = true` stays on the broker forever. Hence the **Last Will**, registered at connect time:

```
var will = new MqttMessage("{\"value\":false}".getBytes(UTF_8), 1, true, new MqttProperties());
options.setWill(lastWillTopic, will);
```

The will is **retained**, so when the broker publishes it on the client's behalf it replaces the stale `true`. The component that fails is not the component that reports the failure — which is the only arrangement that can work. One caveat: a clean DISCONNECT does **not** trigger a will, so a well-behaved service publishes `online=false` itself on the way out, and the will covers the ugly deaths.

9.7 What survived the transport, and what coupling remains

The lower half of the filesystem→MQTT table is the point: `I`, `S`, `E`, egoistic services, runtime topology and "transformer if needed" are **unchanged**. The architecture survived the transport replacement, and that is the strongest available evidence that the architecture was never about the transport.

Loose coupling does not mean nothing knows anything. What remains, necessarily, is agreement on: message syntax, message semantics, topic contracts, Intent, ownership, the meaning of State and the meaning of Events. The goal is not to eliminate coupling but to **eliminate unnecessary knowledge**. Concretely, in this system each service must understand exactly one foreign Event contract — ROT13 understands ConsoleIn's `E/text`, ConsoleOut understands ROT13's `E/crypted` — plus the Intent schema it accepts on its own `I`. That list is the irreducible minimum.

Separately, the broker is a shared infrastructure dependency: everyone needs its address and nothing works without it. Real, but a different kind of thing from one component knowing another.

The step-78 checklist is the tool to keep. Twelve questions, of which these six carry the argument for this system:

question	answer here
does a capability contain a hard-coded producer identity?	no — it arrives as Intent
which foreign contracts must it understand?	one Event contract each, plus its own Intent schema
where is topology defined?	in the configurator, outside the data path
can topology change without recompiling?	yes — demonstrated on a running system
could the transport be replaced?	yes — demonstrated, filesystem then MQTT
does every mediator do something that needs mediation?	yes, vacuously: `TEXT` → `TEXT` needs none, and none exists

The material's own caution applies: the answers need not all be "no". They need to be **intentional**.

9.8 The whole journey

The program never changed: a line goes in, the same line comes out rotated by 13 letters. What changed, eleven times, is how the parts are joined and who holds the knowledge of how they are joined.

stage	how the parts are joined	where the topology lives	in the data path besides the capabilities
1 procedural	direct calls	inside the capability code	—
2 egoistic services	not joined at all	nowhere — that is the problem	—
3 static mediation, one JVM	mailboxes, copied by mediators	the mediators, started in `main()`	two mediators
4 static mediation, filesystem	directories, copied by mediator processes	the operator's command lines	two mediator processes

stage	how the parts are joined	where the topology lives	in the data path besides the capabilities
5 static mediation, Syncthing	replicated directories	the operator's command lines	two mediator processes, plus replication delay
6 named, configurable mediators	`ROUTE` Intent from an orchestrator	the orchestrator, outside the path	the mediators, still
7 dynamic, one JVM	services listen, told by `TOPIC` Intent	the configurator	nothing
8 dynamic, filesystem	the same, over directories	the configurator	nothing
9 dynamic, Syncthing	the same, over replicated directories	the configurator	nothing — transport hops only
10 contracts	compatible: direct; incompatible: a transformer	the configurator	a transformer, only where contracts differ
11 MQTT	topics on a broker	the configurator	nothing — the broker is transport

Read the third column from top to bottom: the topology moves outward, one stage at a time, until no capability holds any of it. Read the last one: the price of that move falls to zero at stage 7 and stays there while the transport changes underneath three times. Same architecture, better transport.

9.9 The sentences to remember

- **Event-driven describes how information and control flow.**
- **Loose coupling describes how much participants need to know about one another.**
- **Publish what happened.**
- **Declare what you want.**
- **Publish your own State.**
- **Keep capabilities egoistic.**
- **Keep topology out of capabilities.**
- **Do not pay for a mediator unless it actually mediates something.**

And whenever an architecture becomes confusing, go back to the first question of this book: **who knows whom?**

Glossary

Every term below is defined where it is first used, and linked from every later mention. The arrow on an entry leads back to that page.