Prozesssteuerung

Romain De Groote

6 Oktober 2026

Ein kleines Programm, Schritt für Schritt neu gebaut — vom direkten Methodenaufruf zu einer ereignisgetriebenen Architektur über MQTT: was sich bei jedem Schritt verschiebt und wer am Ende wen kennt.

Inhaltsverzeichnis

Über diese Notizen **4**
Wie die Abbildungen zu lesen sind . . . 4

1 Schritt 0 — Worum es geht **5**
1.1 Die drei Fragen . . . 5

2 Teil I — Mit dem beginnen, was man schon kennt **6**
2.1 Schritt 1 — Ganz unten: einfach aufrufen . . . 6
2.2 Schritt 2 — Drei egoistische Fähigkeiten . . . 7
2.3 Schritt 3 — Jeder Fähigkeit ein Postfach geben . . . 7
2.4 Schritte 4–6 — Die drei Fähigkeiten werden egoistisch . . . 7
2.5 Schritt 7 — Auftritt des Vermittlers . . . 8
2.6 Schritt 8 — Nochmals fragen: wer kennt wen? . . . 9
2.7 Schritt 9 — Warum virtuelle Threads? . . . 10
2.8 Schritt 10 — Die Dienste egoistisch halten . . . 10
2.9 Was Teil I verschoben hat . . . 10

3 Teil II — Die Prozessgrenze überschreiten **12**
3.1 Schritt 11 — Die Komponenten trennen . . . 12
3.2 Schritte 12–13 — Ein Verzeichnis wird zum Postfach . . . 12
3.3 Schritt 14 — Das erste Problem verteilter Systeme: die Veröffentlichung . . . 13
3.4 Schritt 15 — Ein Postfach mit `WatchService` beobachten . . . 13
3.5 Schritte 16–17 — Dienste und Vermittler werden Prozesse . . . 13
3.6 Schritt 18 — Die vollständig externe lokale Fassung ausführen . . . 14
3.7 Was Teil II verschoben hat . . . 15

4 Teil III — Es wirklich verteilen **17**
4.1 Syncthing repliziert; es liefert nicht aus . . . 17
4.2 Eine Schicht auf einmal prüfen . . . 17
4.3 Was ein Tastendruck kostet . . . 17
4.4 Einen Vermittler zerbrechen . . . 18
4.5 Der Vermittler wird ein gewöhnlicher Dienst . . . 18
4.6 Was Teil III verschoben hat . . . 19

5 Teil IV — Zurückspulen **20**
5.1 Die naheliegende Abkürzung und ihre Falle . . . 20
5.2 Fähigkeit ist nicht Topologie . . . 20
5.3 Was sich im Code ändert . . . 20
5.4 `ROUTE` und `TOPIC` sind Intent . . . 21

Anhang **22**

Glossar **22**

Über diese Notizen

Diese Notizen verfolgen ein kleines Programm durch eine lange Reihe von Neubauten, vom direkten Methodenaufruf bis zu einer ereignisgetriebenen Architektur über MQTT. Jeder Neubau lässt das Programm genau das tun, was es vorher tat, und ändert, wer von wem weiss.

Das gebaute Programm ändert sich nie: Sie tippen eine Zeile, sie kommt um 13 Buchstaben verschoben zurück.

`keyboard` → `ROT13` → `console`

Was sich ändert, ist alles ringsherum. Dasselbe System wird immer wieder neu gebaut, und jeder Neubau stellt drei Fragen, in dieser Reihenfolge:

Wer kennt wen?

Wer besitzt was?

Wer entscheidet, wie das System verdrahtet ist?

Jedes Mal, wenn ein Schritt fertig ist, kommt hier eine Zusammenfassung hinzu; das Buch wächst also im Tempo der Arbeit. Jede Zusammenfassung sagt, was der Schritt getan hat, was beim Ausführen beobachtet wurde und was sich architektonisch verschoben hat — meist ein Stück Wissen, das den Besitzer gewechselt hat.

Nach Material von Reto Konig, Universität Freiburg.

Wie die Abbildungen zu lesen sind

Jede Abbildung existiert einmal als Zeichnung und dreimal als Sprache. In der Webfassung hat eine Abbildung mit Nachrichtenfluss eine kleine Leiste darunter: **Abspielen** schickt eine Nachricht durch die ganze Kette, **Ein Sprung** rückt einen einzigen Schritt vor, **Zurücksetzen** stellt alles zurück. Kästen leuchten auf, wenn die Nachricht sie erreicht; so ist zu sehen, welche Komponente wartet und welche arbeitet.

Im PDF und in einem Browser ohne JavaScript ist dasselbe Bild da — nur unbewegt. Nichts in diesen Notizen hängt von der Animation ab.

1 Schritt 0 — Worum es geht

Schritt 0 enthält keinen Code. Er nennt das Thema: nicht ROT13, sondern die Architektur darum herum. Die Funktionalität bleibt über 81 Schritte fast unverändert — genau das macht die Veränderungen ringsherum sichtbar.

Zwei Regeln werden hier gesetzt und gelten bis zum Schluss:

> **Einer Lösung erst begegnen, nachdem man das Problem erlebt hat, das sie löst.**

Deshalb beginnt der Aufbau nicht mit MQTT und auch nicht mit dynamischen Abonnements. Zuerst entsteht ein System mit absichtlich unbequemen Vermittlern, wird in Betrieb genommen, und erst dann wird gefragt, ob sie ihren Platz verdient haben.

Die zweite Regel betrifft das Vokabular: *Intent*, *Status/State* und *Event* werden erst benannt, nachdem sie benutzt wurden. Sie erscheinen etwa ab Schritt 42, nicht früher.

Die geplante Route, von Schritt 0 an:

```
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 Die drei Fragen

Der Wortlaut der drei Fragen ist wichtig, und man ersetzt ihn leicht durch eine schwächere Fassung:

So gefragt	Eine schwächere Fassung
Wer **kennt** wen?	wer mit wem *kommuniziert*
Wer **besitzt** was?	welche *Daten* jeder Teil hält
Wer **entscheidet** über die Verdrahtung?	welcher Teil alles *verbindet*

„Kennen“ ist nicht „kommunizieren“: zwei Komponenten können den ganzen Tag Nachrichten austauschen, ohne dass eine die andere nennt. „Über die Verdrahtung entscheiden“ ist nicht „verbinden“: der Teil, der Nachrichten weiterleitet, muss nicht der Teil sein, der die Route gewählt hat. Beide Unterscheidungen werden später konkret — die erste schon in Schritt 1, die zweite in Teil III.

2 Teil I — Mit dem beginnen, was man schon kennt

Die Schritte 1 bis 10 bleiben in einer JVM. Das System wird dreimal gebaut: als direkte Aufrufe, als drei Fähigkeiten, die einander nicht kennen wollen, und als dieselben Fähigkeiten, von Vermittlern zusammengeklebt. Nichts ist bisher verteilt, und genau das ist der Punkt: das Einzige, was sich ändert, ist wer wen kennt.

2.1 Schritt 1 — Ganz unten: einfach aufrufen

`ProceduralRot13.java` ist das ganze System in einer kompakten Quelldatei: eine Schleife, die eine Zeile liest, und eine Anweisung, die den rotierten Text ausgibt.

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

        if (text == null)
            return;

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

`IO.readln()` steht in einer eigenen Zeile, `rot13(text)` aber ist kein eigener Schritt: es wird *innerhalb* des Arguments von `IO.println` aufgerufen. Weder `rot13` noch `IO.println` weiss etwas vom anderen. Wer beide kennt, ist `main()`: es nennt sie und legt ihre Reihenfolge fest.

Was `main()` über `rot13` weiss: dass es existiert, wie man es aufruft (ein blosser Name, beide liegen in derselben implizit erzeugten Klasse), welches Argument es nimmt, was es zurückgibt, dass es *jetzt sofort* verfügbar ist und dass die Ausführung nach der Rückkehr weiterläuft. Nicht bekannt ist, wie `rot13` implementiert ist — das Einzige, was es ignorieren darf.

Die Verfügbarkeitsgarantie lohnt eine eigene Erwähnung, denn sie verschwindet später: `main()` und `rot13` sind zwei Methoden einer bereits geladenen Klasse, aufgerufen im selben Thread, im selben Prozess. Nichts muss zuerst starten, keine Nachricht reist irgendwohin, kein Empfänger muss abonniert sein.

Beobachtet: eine getippte Zeile erscheint rotiert; `Ctrl+D` beendet das Programm still, weil `IO.readln()` `null` zurückgibt und `main()` zurückkehrt.



Abbildung 2.1: Die prozedurale Fassung. Die Verdrahtung steht im Code: `main()` nennt alle drei Teile und legt ihre Reihenfolge fest.

2.2 Schritt 2 — Drei egoistische Fähigkeiten

Eine **egoistische** Komponente sagt: ich weiss, was ich kann, welche Daten ich annehme und welche ich erzeuge — und den Geschäftsprozess um mich herum möchte ich lieber nicht kennen. Konkret: `ConsoleIn` darf kein `rot13(...)` enthalten, und `ROT13` kein `consoleOut(...)`.

Das Entfernen dieser Aufrufe schafft sofort ein Problem, das sich als Frage stellen lässt: wenn niemand niemanden kennt, **wer macht daraus ein System?** Zu beachten ist, was die Frage nicht sagt: das Wissen ist nicht verschwunden, es muss woandershin. Es fehlen zwei Verbindungen, nicht eine — `ConsoleIn` ↔ `ROT13` und `ROT13` ↔ `ConsoleOut`.

2.3 Schritt 3 — Jeder Fähigkeit ein Postfach geben

Der kleinste Mechanismus für die fehlenden Verbindungen: eine Nachricht und eine benannte Warteschlange.

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

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

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

Das sind bewusst **Postfächer** (*mailboxes*) und keine Topics, und der Unterschied ist wichtig: rufen zwei Leser `take()` auf demselben Postfach auf, konkurrieren sie — jede Nachricht geht an *genau einen* von ihnen und wird nicht an beide kopiert. Ein Topic in Publish/Subscribe gäbe jedem Abonnenten seine eigene Kopie. Ein Leser pro Nachricht genügt vorläufig, und die Einschränkung bleibt absichtlich bestehen: sie macht später sichtbar, was Publish/Subscribe wirklich hinzufügt.

2.4 Schritte 4–6 — Die drei Fähigkeiten werden egoistisch

Jede Fähigkeit berührt nur noch ihre eigenen Postfachnamen:

Fähigkeit	liest	schreibt
ConsoleIn	die Tastatur	`consoleIn.outbox`
ROT13	`rot13.inbox`	`rot13.outbox`
ConsoleOut	`consoleOut.inbox`	die Konsole

Keine Fähigkeit enthält den Namen einer anderen, und genau das ist das Ergebnis: drei schön isolierte Fähigkeiten, die zusammen absolut nichts tun. Nichts bewegt eine Nachricht von `consoleIn.outbox` nach `rot13.inbox` oder von `rot13.outbox` nach `consoleOut.inbox`.

2.5 Schritt 7 — Auftritt des Vermittlers

Der kleinstmögliche Klebstoff: eine Schleife, die aus einem Postfach nimmt und in ein anderes legt, ohne etwas über das Durchlaufende zu wissen.

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

`main()` startet fünf virtuelle Threads: die drei Fähigkeiten plus zwei Vermittler, deren Routen als Argumente übergeben werden — `consoleIn.outbox` → `rot13.inbox` und `rot13.outbox` → `consoleOut.inbox`.

Beobachtet: `Hello World` hinein, `Uryyb Jbeyq` heraus, genau wie in Schritt 1. Aber `Ctrl+D` beendet das Programm nicht mehr. Der Thread von ConsoleIn *kehrt zurück* (er schleift nicht), und die vier anderen Threads hängen blockiert in `take()` an leeren Postfächern, ohne CPU zu verbrauchen. Am Leben hält den Prozess tatsächlich die letzte Zeile von `main()`, `Thread.currentThread().join()`: der Haupt-Thread wartet auf sich selbst, was nie enden kann. Virtuelle Threads sind Daemon-Threads; die vier wartenden Threads allein hätten die JVM also nicht am Laufen gehalten.

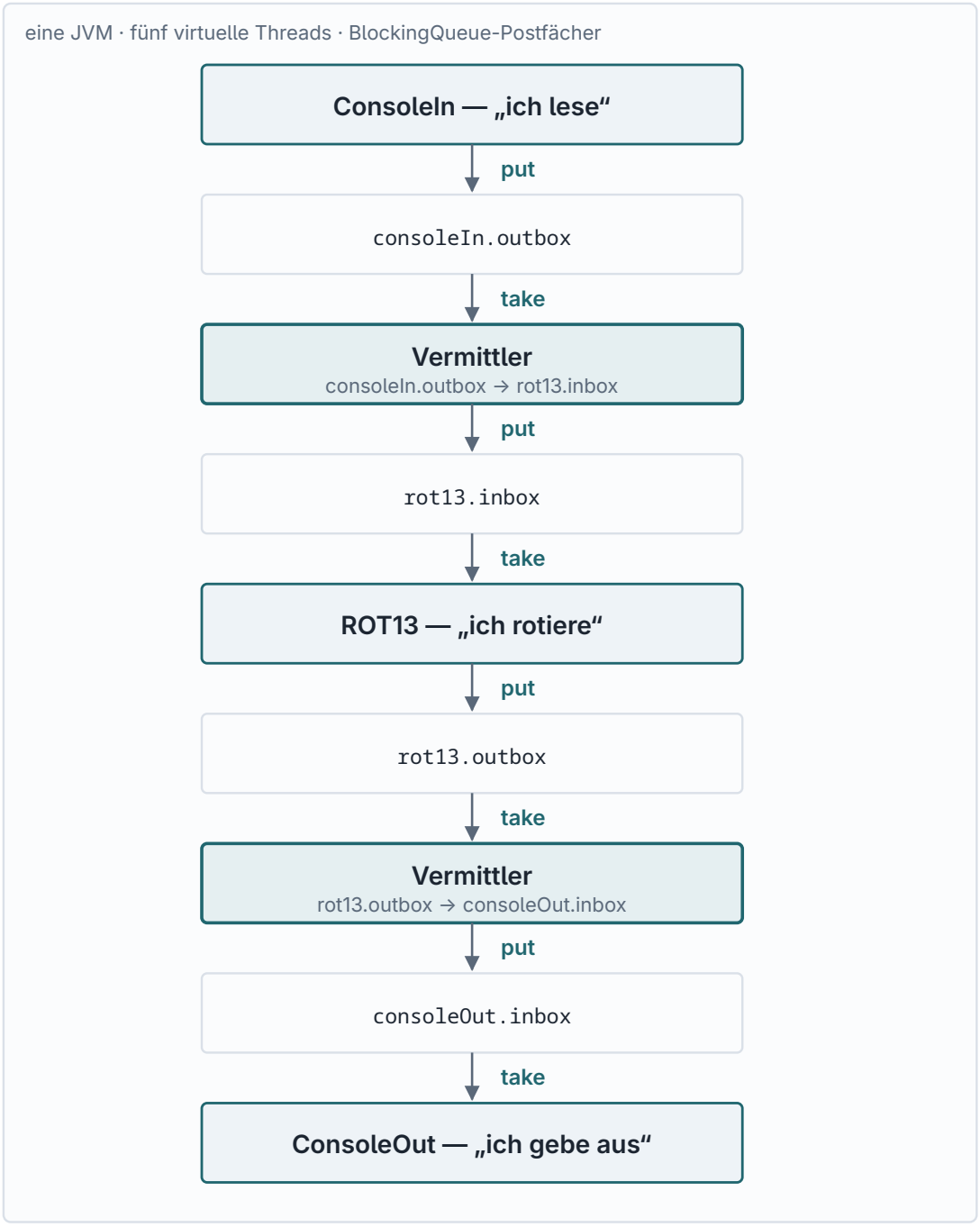


Abbildung 2.2: Die vermittelte Fassung in einer JVM. In der Webfassung schickt **Abspielen** eine Nachricht durch alle acht Sprünge.

2.6 Schritt 8 — Nochmals fragen: wer kennt wen?

Kennt ConsoleIn ROT13? Nein. Kennt ROT13 ConsoleIn oder ConsoleOut? Nein. Ist die Topologie verschwunden? Nein — sie steckt in den zwei `mediate(...)`-Aufrufen innerhalb von `main()`.

Das ist das erste grosse Ergebnis:

Entkopplung lässt Wissen oft nicht verschwinden. Sie verschiebt Wissen dorthin, wo es hingehört.

Die Fähigkeit „ROT13 transformiert Text" gehört zu ROT13. Die Topologie `ConsoleIn` → `ROT13` → `ConsoleOut` gehört zur Systemkomposition. Geändert hat sich seit Schritt 1 nicht, dass die Verdrahtung `main()` verlassen hat — sie ist noch dort —, sondern dass sie nicht mehr *innerhalb* einer Fähigkeit steht. Sie ist jetzt Daten: zwei Paare von Postfachnamen. Umverdrahten berührt diese zwei Zeilen und keine einzige Fähigkeit.

2.7 Schritt 9 — Warum virtuelle Threads?

Fünf Schleifen warten nun unbegrenzt: die Tastatur, `rot13.inbox`, `consoleOut.inbox` und die Quellen der zwei Vermittler. Virtuelle Threads sind einfach der billige Java-Weg, fünf Wartevorgänge gleichzeitig in einer JVM laufen zu lassen.

Sie sind eine Implementierungsbequemlichkeit, keine Architektur. Die Architektur — wer wen kennt und wo die Verdrahtung liegt — steckt in den Postfachnamen, nicht in den Threads. Der Beleg für die Trennung: dieselbe Architektur läuft in Schritt 11 als getrennte Prozesse, ohne geteilte Threads zwischen den Teilen.

2.8 Schritt 10 — Die Dienste egoistisch halten

Die naheliegende Abkürzung: der erste Vermittler kopiert Nachrichten nur unverändert — warum ihn also nicht löschen und ROT13 direkt `take()` auf `consoleIn.outbox` machen lassen?

Weil ROT13 dann den Namen `consoleIn.outbox` enthielte, also wüsste, dass ConsoleIn existiert und wohin es veröffentlicht. Die Topologie wanderte *in* ROT13 hinein, und ROT13 wäre keine egoistische Fähigkeit mehr. Der Test, der das konkret macht: die Eingabe komme künftig von einer neuen Fähigkeit `FileIn`, die in `fileIn.outbox` veröffentlicht.

	mit Vermittler	mit der Abkürzung
was sich ändert	ein Argument des Vermittlers	`runRot13` selbst
wer berührt wird	keine Fähigkeit	ROT13, dessen Aufgabe unverändert blieb

Der Preis gehört klar gesagt: dieses Design mag indirekt wirken — sogar etwas eigenartig. Es gibt einen anderen Weg, Dienste egoistisch zu halten, *ohne* Vermittler im Pfad jeder Nachricht; er kommt später.

2.9 Was Teil I verschoben hat

Stufe	wo die Topologie liegt	wer umverdrahten kann
Schritt 1	im Code der Fähigkeit (`IO.println(rot13(text))`)	wer diesen Code ändert
Schritte 7–8	in `main()`, als Paare von Postfachnamen	wer `main()` ändert

Funktionalität: unverändert. Codezeilen: mehr. Teile, die einander kennen: keine. Das ist der Tauschhandel, den alles Folgende immer wieder prüft.

3 Teil II — Die Prozessgrenze überschreiten

Die Schritte 11 bis 18 nehmen die Architektur aus Teil I und verteilen sie auf fünf Betriebssystemprozesse auf einem Rechner. Die Fähigkeiten behalten ihre Postfachnamen; nur der Transport darunter wechselt, von Java-Objekten zu Dateien in Verzeichnissen. Das erste wirklich verteilte Problem tritt hier auf, und es hat nichts mit ROT13 zu tun.

3.1 Schritt 11 — Die Komponenten trennen

Drei Prozesse können keine `BlockingQueue` teilen: jeder `java`-Befehl startet seine eigene JVM mit eigenem Speicher, die `mailboxes`-Map existierte also dreimal, und jede Kopie sähe nur ihre eigenen Nachrichten. Ein externer Transport ist nötig.

TCP würde gehen, MQTT würde gehen. Absichtlich fällt die Wahl auf die primitivste Möglichkeit, das **Dateisystem**, mit einer einzeiligen Begründung: *weil man es sehen kann.* Nachrichten werden Dateien, die man mit `ls` auflisten und mit `cat` lesen kann, während das System läuft. Primitiv heisst hier nicht schlechter; Teil VIII kommt darauf zurück, was das Dateisystem wirklich gut und wirklich schlecht kann.

3.2 Schritte 12–13 — Ein Verzeichnis wird zum Postfach

Die Zuordnung ist eine Zeile lang:

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

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

Der `Message`-Record erhält einen optionalen Zeitstempel, und das Dateiformat bleibt zwei Zeilen plus eine optionale dritte:

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

Eine handgeschriebene Nachricht darf den Zeitstempel weglassen. Die Marke `TIMESTAMP:` verhindert, dass eine Inhaltszeile, die zufällig wie ein Datum aussieht, für Metadaten gehalten wird; in diesem winzigen Format ist eine letzte Zeile, die mit `TIMESTAMP:` beginnt, reserviert.

Jackson könnte hier eingeführt werden und wird es absichtlich nicht. Der Sinn eines lächerlich einfachen Formats ist, dass fast nichts die Architektur verdeckt — die ganze Nachricht ist in jedem Texteditor lesbar, was die Aufmerksamkeit darauf hält, wer mit wem spricht.

3.3 Schritt 14 — Das erste Problem verteilter Systeme: die Veröffentlichung

Ein Leser, der ein Verzeichnis beobachtet, kann eine Datei entdecken, *bevor der Schreiber sie fertig geschrieben hat*. Daher ein Veröffentlichungsprotokoll:

1. eine private temporäre Datei erzeugen;
2. die vollständige Nachricht schreiben;
3. sie schliessen;
4. sie atomar auf ihren endgültigen `.msg`-Namen umbenennen;
5. Konsumenten reagieren nur auf endgültige Dateien.

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

Beide Hälften sind nötig. Das Umbenennen ist atomar — ein Leser sieht entweder keine `.msg`-Datei oder die vollständige —, aber es schützt nichts, wenn Leser auch `.tmp`-Dateien aufgreifen. `.tmp` und `.msg` bleiben im selben Verzeichnis, damit der Verschiebevorgang kein Dateisystem überschreitet, wo die Atomarität verloren ginge.

> **Schreiben und Veröffentlichen sind nicht dieselbe Operation.**

3.4 Schritt 15 — Ein Postfach mit `WatchService` beobachten

Javas eigener Dateisystem-Beobachter liefert die Konsumentenhälfte: ein Verzeichnis für `ENTRY_CREATE` registrieren, in `watcher.take()` blockieren und jeden Dateinamen ignorieren, der nicht auf `.msg` endet.

Das ergibt auch ein sehr einfaches Einspeisewerkzeug. Mit `ActuatorService` als laufendem Konsumenten:

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

Beobachtet: beim Auftauchen der `.tmp`-Datei geschieht nichts; die Zeile erscheint in dem Moment, in dem das Umbenennen den `.msg`-Namen erzeugt. Beide erlaubten Formate funktionieren, mit und ohne abschliessenden Zeitstempel.

Die architektonische Lehre lautet `producer` → `filesystem` → `consumer`: der Konsument konnte einen Menschen, der `mv` tippt, nicht von einem echten Dienst unterscheiden. Er kennt sein eigenes Eingangspostfach, nicht den, der hineinschreibt. Der Egoismus hat die Prozessgrenze überlebt.

3.5 Schritte 16–17 — Dienste und Vermittler werden Prozesse

Die drei Fähigkeiten werden drei Programme, und der Vermittler eine vierte Art Programm:

Programm	spielt	liest	schreibt
`SensorService.java`	ConsoleIn	die Tastatur	`consoleIn/outbox`
`ProcessorService.java`	ROT13	`rot13/inbox`	`rot13/outbox`
`ActuatorService.java`	ConsoleOut	`consoleOut/inbox`	die Konsole

Programm	spielt	liest	schreibt
`Mediator.java <from> <to>`	der Klebstoff	`<from>`	`<to>`

Der Vermittler ist generisch: seine Route ist nicht einkompiliert, sie kommt als zwei Startargumente. Doch eine Asymmetrie sollte man im Kopf behalten: die drei Fähigkeiten erhalten ihre **Arbeit** als Nachrichten in einem Postfach, der Vermittler dagegen seine **Route** als Startkonfiguration — und er hat keinen Namen und kein eigenes Steuerpostfach, in das jemand schreiben könnte.

Praktische Folge: Argumente werden einmal gelesen, das Ändern der Route eines laufenden Vermittlers heisst also: anhalten und neu starten. Eine Konfigurationsdatei wäre nicht anders — auch sie wird beim Start gelesen. Dieser Punkt bleibt absichtlich offen, bis die Kosten der verteilten Vermittler zu spüren waren.

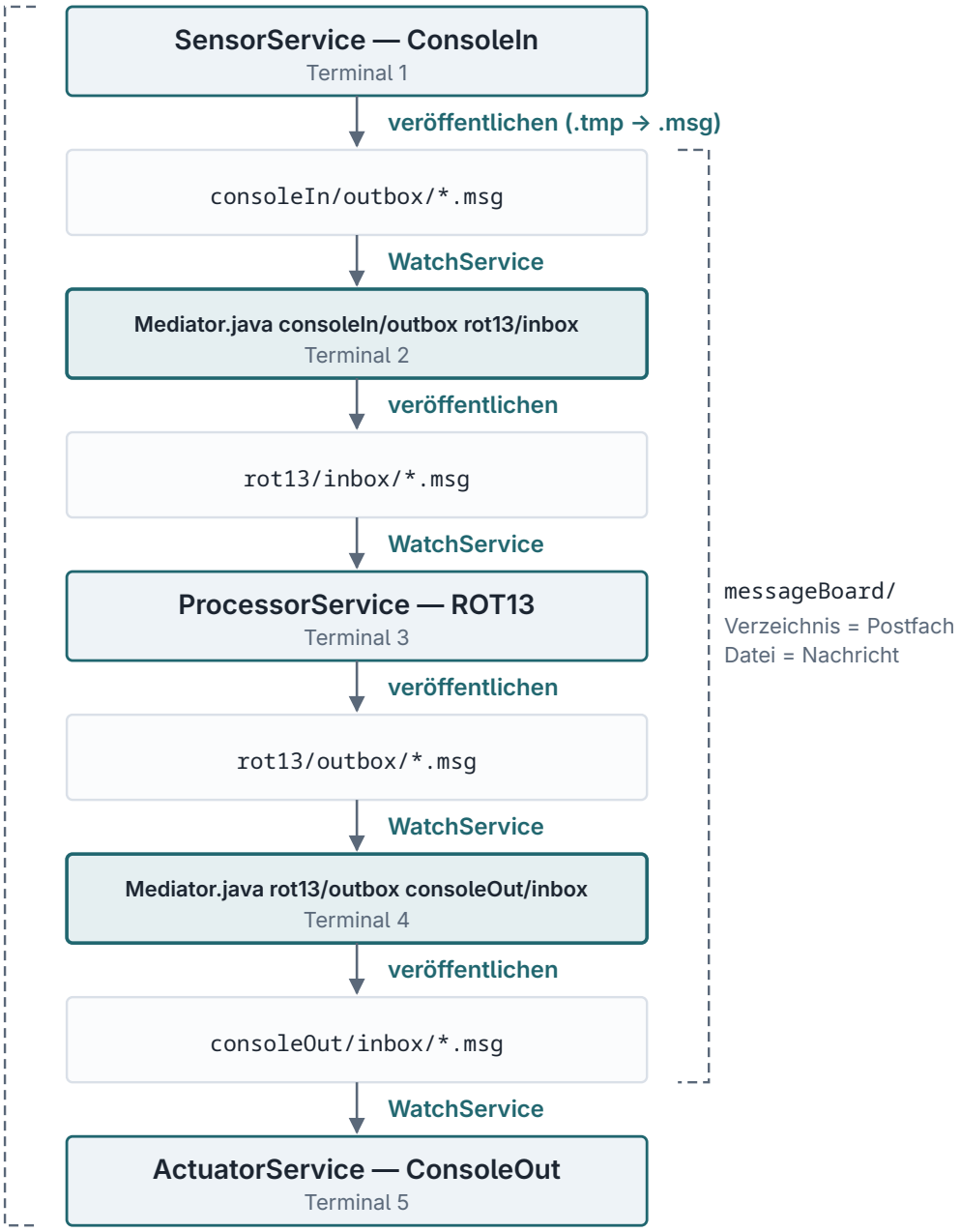
3.6 Schritt 18 — Die vollständig externe lokale Fassung ausführen

Fünf Terminals, fünf Prozesse, ein lokales Dateisystem dazwischen:

```
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
```

Beobachtet: `Hello World`, in Terminal 1 getippt, erschien als `Uryyb Jbeyq` in Terminal 5. Die drei mittleren Terminals zeigten nichts Neues — jeder Vermittler gibt beim Start eine Zeile „Mediating: …" aus, und `ProcessorService` gibt überhaupt nichts aus. Danach enthielt jedes der vier Postfachverzeichnisse eine `.msg`-Datei: dieselbe Zeile, entlang der Kette kopiert.

Nachzählen: **fünf Prozesse** für das, was Schritt 1 in einer Anweisung tat. Jetzt ist nicht der Moment zu optimieren: bequem soll es noch nicht sein.



**Fünf von Hand gestartete Prozesse.
Die Topologie steckt nun in den zwei Mediator-Befehlszeilen —
bei dem, der sie getippt hat.**

Abbildung 3.1: Die fünf Prozesse und die vier Postfachverzeichnisse. In der Webfassung folgt **Abspielen** einer Textzeile von der Tastatur zur Konsole.

3.7 Was Teil II verschoben hat

Stufe	wo die Topologie liegt	wer umverdrahten kann
Schritt 1	im Code der Fähigkeit	wer diesen Code ändert
Schritte 7–8	in `main()`, als Postfachnamen	wer `main()` ändert

Stufe	wo die Topologie liegt	wer umverdrahten kann
Schritt 18	in den zwei Befehlszeilen `Mediator.java <from> <to>`	wer die Prozesse startet

Die Topologie hat den Quellcode nun ganz verlassen: keine `.java`-Datei enthält die Route. Sie steckt in dem, was der Bediener getippt hat — damit ist derzeit ein Mensch der Orchestrator. Was die Fünf-Prozess-Fassung einbringt, ist echt: die Verdrahtung kann sich ändern, ohne ein Programm anzufassen. Was sie kostet, ist ebenso echt: fünf Prozesse, vier Kopien jeder Nachricht und ein Programm, das bei `Ctrl+D` nicht mehr stehen bleibt.

4 Teil III — Es wirklich verteilen

Fünf Prozesse auf einem Rechner teilen immer noch ein Dateisystem. Die Schritte 19 bis 27 verteilen sie auf zwei Maschinen, replizieren das Nachrichtenbrett mit Syncthing zwischen ihnen, messen die Kosten und zerbrechen dann absichtlich einen Vermittler. Am Ende ist der Preis der Vermittler eine Zahl und keine Meinung.

4.1 Syncthing repliziert; es liefert nicht aus

Syncthing hält ein Verzeichnis zwischen Maschinen repliziert. Mehr wird hier nicht verlangt. Es weiss nichts von Intent, Status, Event, ROT13 oder Vermittlern, und es kennt keinen Konsumenten: keine Abonnements, keine Topics, keine Lieferzusagen, keine Vorstellung davon, dass jemand wartet.

Der Auslöser bleibt unser eigener. Jeder Dienst betreibt seinen eigenen `WatchService` auf einem **lokalen** Verzeichnis; Syncthing lässt die Dateien dort nur erscheinen. Deshalb musste keiner der drei Dienste eine Zeile ändern, als das System verteilt wurde.

Daraus folgen zwei Dinge, die später als Beobachtung und nicht als Warnung auftreten:

- Replikation ist **asynchron** — das Umbenennen `.tmp` → `.msg` ist auf jeder Maschine einzeln atomar, und nichts verspricht Reihenfolge oder Atomarität zwischen Maschinen;
- das Dateisystem ist ein ernstzunehmender Transport mit eigenen Stärken: Dateien dürfen gross sein, Replikate überleben einen Neustart, und jede Nachricht lässt sich mit `cat` lesen.

4.2 Eine Schicht auf einmal prüfen

Bevor irgendein Java-Programm den replizierten Ordner berührte, lief der langweiligste denkbare Test: `hello.txt` auf einer Maschine schreiben, prüfen, dass es auf der anderen erscheint, es löschen, prüfen, dass die Löschung zurückkommt. Beide Richtungen funktionierten.

Diese Reihenfolge ist wichtig. Als der verteilte Lauf später scheiterte, war die Frage „Replikation oder mein Programm?" für den Transport schon beantwortet, und die Suche konnte eine Schicht höher beginnen.

4.3 Was ein Tastendruck kostet

Mit dem Sensor auf einer Maschine und den Vermittlern, ROT13 und dem Aktor auf der anderen erzeugte eine getippte Zeile **vier** `.msg`-Dateien, eine pro Postfach:

Postfach	Inhalt	Zeitstempel darin
`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

Postfach	Inhalt	Zeitstempel darin

Die wiederholten Zeitstempel belegen, dass zwei dieser vier Veröffentlichungen **reine Kopien** sind: die Vermittler veröffentlichen die Nachricht unverändert erneut, Zeitstempel eingeschlossen. Sie berechnen `identity(x) = x`, als eigene verteilte Prozesse.

Die Zeiten, gemessen an den Änderungszeiten der vier Dateien auf einer Maschine, damit keine Uhrendifferenz hineinspielt: die Datei des Sensors erscheint um `01:02:05.432`, die drei anderen um `01:02:13.460`, `.504` und `.532` — **8,10 s** später und innerhalb von **72 ms** zueinander. Die eigentliche Arbeit der Kette dauerte also Millisekunden; die acht Sekunden sind Syncthings Verbreitung. Dieselbe Kette, vollständig auf einer Maschine, braucht etwa 24 ms.

4.4 Einen Vermittler zerbrechen

Halten Sie den ersten Vermittler an und tippen Sie zwei Zeilen. ConsoleIn lebt, ROT13 lebt, ConsoleOut lebt, Syncthing lebt — und der Geschäftsfluss ist tot. Die zwei Zeilen stapeln sich in `consoleIn/outbox`; weiter kommt nichts. Die Vermittler gehören zur Verfügbarkeitskette.

Starten Sie den Vermittler neu. **Nichts wird ausgeliefert.** `FileMessageBoard.watch` registriert ein Verzeichnis für `ENTRY_CREATE` und blockiert in `watcher.take()`; es listet nie, was schon da ist. Die zwei Dateien liegen weiter auf der Platte, in beiden Replikaten — es ging also nichts verloren. Kein Gedächtnis hat das *Abonnement*.

Diese Unterscheidung lohnt sich: **die Nachrichten überlebten, die Auslieferung nicht.** Ein Broker antwortet darauf mit Sitzungszustand und retained messages; im eigenen Code wäre es eine Zeile — die vorhandenen `.msg`-Dateien auflisten, bevor die Beobachtungsschleife beginnt — die das Material bewusst noch nicht schreibt.

4.5 Der Vermittler wird ein gewöhnlicher Dienst

Bevor die Vermittler verschwinden, fällt eine Asymmetrie weg: die Dienste erhalten ihre *Arbeit* als Nachrichten, der Vermittler erhielt seine *Route* als Shell-Argumente. Er bekommt daher nur einen Namen, `ConfigurableMediator mediator1`, aus dem er ein dauerhaftes Steuerpostfach ableitet, `mediators/mediator1/inbox`, und die Route kommt als Nachricht:

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

`from` und `to` reisen absichtlich in **einer** Nachricht: als zwei hielte der Vermittler kurz eine neue Quelle mit dem alten Ziel und leitete an die falsche Stelle weiter. Eine Route ist ein Fakt, und ein Fakt gehört in eine Nachricht.

Ein kurzlebiger Orchestrator veröffentlicht beide Routen und endet, damit die Topologie an einer Stelle sichtbar ist und **nicht** im Datenpfad liegt. Startet man ihn nach den Diensten und hat vorher eine Zeile getippt, zeigen sich beide Hälften auf einmal: die frühe Zeile bleibt liegen, die späte läuft durch.

Was besser wird: der Vermittler hat eine Identität, nimmt Konfiguration über sein eigenes Postfach, behält eine generische Fähigkeit und bezieht seine Topologie von einem Orchestrator. Was nicht besser wird: der Datenpfad. Er bleibt `event` → `Vermittler` → `kopiertes event`, weiter ein

Prozess in der Verfügbarkeitskette, der jede Nachricht ohne Bedeutungsgewinn kopiert. Und diese Fassung nimmt genau **eine** Route an: `waitForRoute(...)` kehrt zurück, dann betritt der Vermittler `watch(route.from(), …)` und liest sein Steuerpostfach nie wieder. Umverdrahten im Betrieb wird nicht versprochen, weil ein blockierendes `watch` nicht gleichzeitig ein Steuerpostfach und eine Quelle bedienen kann — der Grund für die Abonnement-Hilfe später.

4.6 Was Teil III verschoben hat

Stufe	wo die Topologie liegt	im Datenpfad?
Schritt 18	die Befehlszeilen des Bedieners	die Vermittler sind es
Schritt 27	ein Orchestrator, als Intent veröffentlicht	die Vermittler sind es weiterhin

Die Frage, mit der der Teil schliesst, ist daher scharf und nicht rhetorisch: **warum kann der Dienst, der sich für die Daten interessiert, sie nicht selbst hören?**

5 Teil IV — Zurückspulen

Die Schritte 28 bis 34 kehren absichtlich zu einer JVM zurück. Ohne Replikationsverzögerung und ohne fünf Terminals kann sich nur ändern, wer wen kennt — die Idee lässt sich also in Sekunden ausprobieren statt in einer verteilten Fehlersuche.

5.1 Die naheliegende Abkürzung und ihre Falle

Wenn der erste Vermittler nur Nachrichten kopiert: löschen wir ihn und lassen ROT13 die Quelle selbst lesen:

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

Der Vermittler ist weg. Doch die wiederkehrende Frage: ROT13 enthält jetzt den Namen von ConsoleIns Postfach. Die Topologie ist aus dem Vermittler **in den Dienst** gewandert. Im Wortlaut des Materials: „Wir haben den Vermittler entfernt … und die Topologie zurück in den Dienst gelegt. *Hoppla.*"

5.2 Fähigkeit ist nicht Topologie

ROT13 darf zwei Dinge wissen: *ich kann Text transformieren* und *ich kann auf kompatible Textnachrichten hören*. Es darf **nicht** wissen: *meine Eingabe kommt immer aus consoleIn.outbox* — das ist Deployment-Topologie.

Der Name ist daher nicht einkompiliert, er kommt zur Laufzeit:

```
TOPIC consoleIn.outbox
```

ROT13 lernt seine Verdrahtung aus Daten. Wer sie sendet, ist wichtig: in `DynamicQueueRot13.java` veröffentlicht `main()` beide `TOPIC`-Nachrichten — ein Konfigurator ausserhalb des Datenpfads, in der Rolle des früheren Orchestrators. ConsoleIn kann es nicht sein: sagte ConsoleIn ROT13, was es abonnieren soll, müsste es wissen, dass ROT13 existiert — die Topologie wäre nur in eine andere Fähigkeit gewandert.

5.3 Was sich im Code ändert

`listen(...)` ist nichts Exotisches: ein virtueller Thread, der wiederholt `take()` auf einer derselben Warteschlangen aufruft. `subscribe(...)` startet einen solchen Thread pro abonniertem Namen. Die Folge ist in den Postfächern sichtbar:

- ROT13 nimmt **direkt aus `consoleIn.outbox`** — dem Namen, den es erhalten hat;
- `rot13.inbox` ist kein Nutzlast-Postfach mehr. Es trägt Steuernachrichten, und ROT13s Schleife liest es, um zu entscheiden, ob eine Nachricht ein anzuwendendes `TOPIC` ist.

Nichts schreibt mehr in `rot13.inbox`, um Daten zu bewegen. Das Postfach verschwand nicht; es wechselte die Aufgabe, von Nutzlast zu Intent.

Eine Beobachtung aus dem Lauf: nach `Ctrl+D` endet die Ausgabe, das Programm bleibt aber am Leben. Unterbrochen wurde nichts — `runConsoleIn` kehrt am Eingabeende zurück, der **Produzent** ist also weg, während beide Abonnenten-Threads in `take()` auf Quellen warten, die nie wieder veröffentlichen. Die JVM lebt weiter, weil `main` mit `Thread.currentThread().join()` endet. „Der Fluss stoppte" muss immer darauf zurückgeführt werden, *welcher* Beteiligte stoppte.

5.4 `ROUTE` und `TOPIC` sind Intent

Zwei Tests für `TOPIC consoleIn.outbox`. Beschreibt es etwas, das geschehen ist? Nein. Etwas, das gerade gilt? Nein. Es sagt *beginne hier zu hören* — gewünschtes Verhalten, also **Intent**. Das `ROUTE` des Vermittlers ist dieselbe Art Nachricht: *leite von hier nach dort weiter*.

Eine Falle, die benannt gehört, weil sie in einer Verteidigung verlockt: ein Intent zu *veröffentlichen* ist natürlich etwas, das geschehen ist. Die Einordnung folgt dem, was die Nachricht sagt, nicht ihrer Existenz.

Und eine Bedingung, die wiederkommt: die dynamische Fassung ist **nicht** automatisch besser. Sie ergibt nur Sinn, wenn der Konsument den Nachrichtenvertrag des Produzenten wirklich versteht.

```
STATISCH VERMITTELT              DYNAMISCH DIREKT
Produzent → Vermittler → Konsument   Intent → (Produzent ⟶ Konsument)
```

Glossar

Jeder Begriff unten wird dort definiert, wo er zum ersten Mal gebraucht wird, und von jeder späteren Erwähnung aus verlinkt. Der Pfeil eines Eintrags führt dorthin zurück.