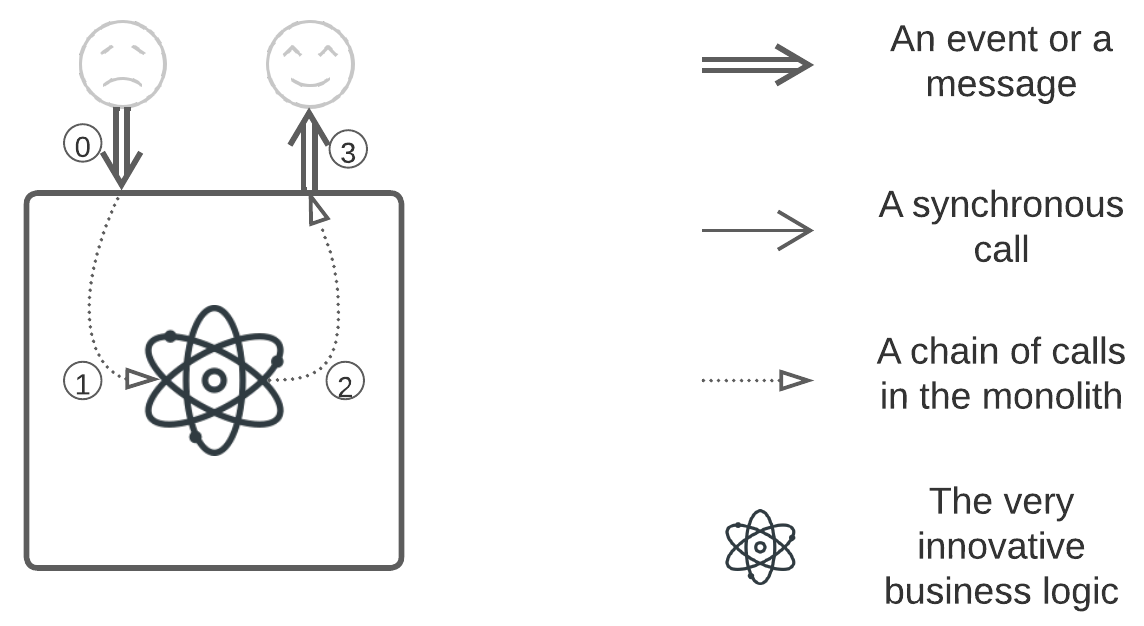
*Part* 1 **2** 3 4 5

Introduction to Software Architecture with Actors:  
Part 2 – On Handling Messages

After defining actors and traversing the *design space* [[POSA1](#wuckyka5hvdv), [POSA5](#kix.rkgtqdele94c)] for systems built of actors, it is time to look inside an actor to find out how it may work. However, first we’ll need to make a distinction between *control* *flow* and *data flow*.

## Control Flow vs Data Flow

All software processes incoming information to produce outgoing information.



If a piece of software does not produce anything, it serves no purpose. The information a piece of software works on can be either ***data*** (a sequence of signals in a hardware bus) or ***events*** (signals in a hardware bus coming at specific moments). The systems that process these two kinds of information have distinct properties.

Let’s compare a camera and a cell phone.



A camera exists to stream data from its matrix to either its display or a flash card. The process may be something like:

[Read from the matrix to the RAM] => [RGGB to RGB] => [Calculate statistics for the image] => [Apply brightness correction] => [Apply color correction] => [Apply noise reduction] => [Decrease the color depth] => (1: [Write to the LCD]) *or* (2: [Write to the viewfinder]) *or* (3: [Compress to JPEG] => [Write to the flash]).

Every frame the matrix captures takes this path, and the steps taken never depend on the frame itself. Some of the steps (*filters*) in the frame processing pipeline are optional (their application depending on the camera’s settings), but the pipeline is very stable – as the settings are seldomly changed – to the extent that hundreds if not thousands of frames pass through the same pipeline.

This was a description of **data flow**. The data passes through a static sequence of processing steps, as through a [meat grinder](https://youtu.be/HrxX9TBj2zY?t=49).

Now, let’s consider a cell phone. A user pressing the “2” key results in one of the following actions:

* If an audio connection is established, a message with the dialed digit (“2”) will be sent to the radio frequency interface, and the DTMF feedback generator will be activated.
* If the cell phone’s display shows a text input screen, the last pressed key was “2”, and the last key event’s timestamp is more recent than the specified timeout, the last dialed symbol on the screen will change.
* Otherwise, if the cell phone’s display shows a text input screen, “A”, “a”, or “2” (depending on the software state), will be appended to the text on the screen.
* If the screen shows a menu, the second option on the menu will be activated.
* If the *Snake* game is on the screen, the snake will turn upwards, unless it is already crawling upwards.
* If an alarm is active, its ringing will be delayed by 5 minutes.
* If the keypad is locked and there have been no recent key presses, animated instructions for unlocking the keypad will be shown.
* Otherwise, if the keypad is locked, nothing will happen.
* If the phone is powered off (or, rather, its CPU is running at a lower frequency, for phones never ever shut down entirely; they just wait for the power button to be pressed again), nothing will happen.

In addition to any of the events listed above, the tone generator will be activated if the “keypad tones” configuration option is enabled.

Here, a single event (pressing the “2” key) may cause various use cases or be completely ignored, based on the system’s current state. Some of the resulting actions may change the current state in such a way that the subsequent “2” key event will cause a totally different action. Even the activation of hardware modules depends on the current state of the software.

This is referred to as **control flow**. An event has no inherent meaning outside the current state of the system, which is decisive for choosing any (including void) reaction to the incoming message.

Of course, a camera has menus written in the *control flow* style. It’s also true that every phone contains an audio path engineered for *data flow*. In fact, many real-life systems involve both paradigms, but just as the requirements for *control-* and *data*-flow modules differ, so too do their architectures. While a camera may not respond to key presses while it is writing a picture to its memory card, a cell phone, as a real-time device, must allow its user to immediately stop a call in any situation (Yes, stopping a call is a complex, multi-step scenario that starts with pressing a key).

|  | **Data Flow** | **Control Flow** |
| --- | --- | --- |
| *Aim* | Processing a stream of data | Reacting to events |
| *Efficiency as* | Throughput | Responsiveness |
| *Message propagation through the system* | Similar for every message of a given type | Usually differs between instances of an event |
| *Messages change the system’s state* | Usually, no | Usually, yes |
| *Interface length (number of types of incoming messages)* | Narrow (a few to dozens) | Wide (hundreds to thousands) |
| *Length of data payload per message* | Usually long (data packet) | Usually short (event notification) |

We are now ready to start reviewing the architectures themselves.

## Monolith

*“Use case”, “scenario”, “request processing” and “task” will be used as synonyms for chains of steps that implement business logic rules.*

*These days, people refer to a set of components that need to be deployed together as a “monolith”, though the term, which was coined in the 1980s, had a distinctly different meaning prior to the proliferation of internet servers, and with them, the notion of "deployment". In spite of the fact that [*[*POSA1*](#wuckyka5hvdv)*] does not contain the term “monolith” itself, it uses “monolithic system”, “monolithic application” or “monolithic structure” as opposed to a loosely coupled (i.e. layered or distributed) system. Let’s reuse this old notion and call entities with no discernible structure* ***monoliths****. As the present cycle of articles centers on asynchronous systems, wherein components interact by sending messages, most synchronous services should be treated as monoliths for the duration of the discussion.*

Monolith is the simplest structure, and is [recommended](https://martinfowler.com/bliki/MonolithFirst.html) [[MP](#kix.hccjphx9ia0l)] for the following cases:

1. When working in a new domain that none in the team is experienced with (i.e, there is no sure way to split the project into loosely coupled parts), or
2. The project is unlikely to grow huge in any dimension (load, complexity, longevity) and
3. There are no special (real-time response, high availability) or conflicting non-functional requirements.

*Benefits:*

* The project gets a quick start (no heavy infrastructure; just go code the logic).
* It employs the simplest possible code structure (unless the project grows large).
* It’s easy to debug.
* If anything crashes, everything crashes (so there’s no need to check for incomplete tasks, half-written data or unowned locks).
* Architectural decisions are delayed till the domain is better understood – this follows the saying that *the most important decisions somehow need to be done when one has the least information available*.
* The monolithic structure allows for the best optimizations on resource (CPU, RAM, drives, network) use.

*Drawbacks:*

* The entire application’s code is coupled; thus, it is impossible to vary properties (non-functional requirements [[MP](#kix.hccjphx9ia0l)]) of parts of the system (subdomains or drivers).
* The code quality quickly deteriorates as time passes and requirements change (poor evolvability).
* If anything crashes, everything crashes (Users only have so much patience).

*Evolution*:

* If the business logic becomes unmanageable because of the project’s size, the monolith should be split into asynchronous *(Micro-)Services*, described in Part 3. This may also help with scalability.
* The business logic can be isolated from the underlying platform, transports and 3rd party libraries by applying *Layers* (from Part 3 of this series), followed by *Hexagonal Architecture* (described in Part 4). This also has a limited positive effect on the project complexity and improves testability. Another good option is going for *Pipeline* (Part 3), but that architecture only fits a specific subset of applications.
* Scalability and fault tolerance are achieved firstly by *sharding* (Part 3), with finer control being possible through the use of such granular architectures as *Pipeline* (Part 3), *SOA* (Part 5) or even *Nanoservices* (Part 3).
* Contradictory non-functional requirements will lead to the fragmentation of the monolith into *Layers* (Part 3), *Hexagonal Architecture* (Part 4) and *Hexagonal Hierarchy* (Part 5) or *(Micro-)Services* (Part 3), probably over a *Shared Repository* (Part 4), that may grow into a *Cell-Based Architecture* (Part 5). Less common options include *Nanoservices* (Part 3) and *SOA* (Part 5).
* Customizability is achieved with *Plug-Ins* or *Domain-Specific Language* (both from Part 4). *Pipeline* (Part 3) is also a good option if it fits the project’s domain.

It is noteworthy that the fact that “If anything crashes, everything crashes” is both a benefit and a drawback. Software architecture is all about making decisions with important benefits and ignorable drawbacks, while what exactly counts as important or ignorable depends on the project. That also holds true for the simplicity of software design – monoliths are easy to start coding, but as the project grows, they become hard to maintain and evolve. One should choose one’s method based on what is important now and what is expected to be important in the future. Everything has a cost.

*Common names*: [**Big Ball of Mud**](https://martinfowler.com/bliki/MonolithFirst.html), Hello World, KISS + YAGNI*.*

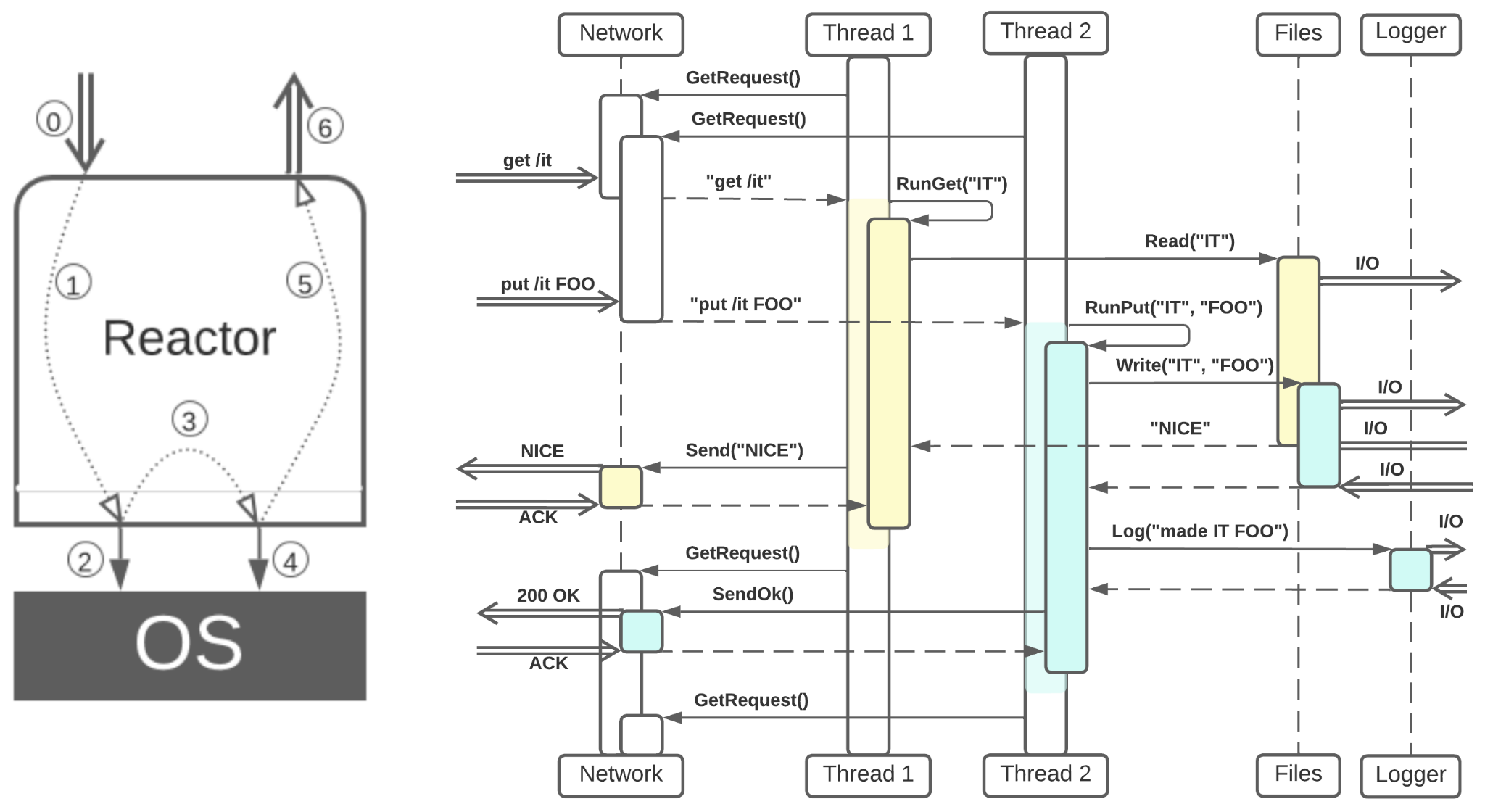
*Software architecture*: **Reactor** [[POSA2](#kix.hefqusybmb4l)], Proactor [[POSA2](#kix.hefqusybmb4l)], Half-Sync/Half-Async [[POSA2](#kix.hefqusybmb4l)].

*System architecture*: **Monolith**.

Basically, any unstructured (for the current discussion, meaning “not divided by asynchronous interfaces”) code belongs here. This is usually good (simple and stupid) for a small amount of code without any special requirements.

Handling an incoming request/event triggers a chain of data processing operations (*data flow*) or a choice of reactions (*control flow*) that can be implemented as either *Reactor* or *Proactor*, which can also be mixed in a couple of ways (as is common with software architecture).

### Reactor [[POSA2](#kix.hefqusybmb4l)] (synchronous calls)



*(Parallel use cases are shown in colors)*

***A thread per task – imperative programming.*** Every incoming request is served by a dedicated (for the duration of processing the request) thread, every call out of the *reactor* (to the OS, over the network or into libraries) is blocking. Typically, multiple request handling threads are used (e.g. *Leader/Followers* [[POSA2](#kix.hefqusybmb4l)] or *Thread Pool* (*Master-Slave* [[POSA1](#wuckyka5hvdv)]), both discussed in Part 3). Multithreaded reactors have to protect their state with mutexes, are indeterministic and may be considered the default approach for *data-flow*-dominated systems (naive backend implementation). The basic request processing code is simple, but it is [hard](https://hillside.net/plop/2020/papers/poltorak.pdf) for one of the request processors to influence others (i.e., to cancel or edit a running request). Thus, a multithreaded reactor is a kind of simplistic *sharding* (to be discussed in Part 3 of the publication).

*Benefits:*

* The code for use cases is simple.
* Debugging is easy.

*Drawbacks (common):*

* There is no multicast support (It’s impossible to send several outgoing requests in parallel, unless one goes through the trouble of implementing promises).

*Drawbacks (single-threaded variation):*

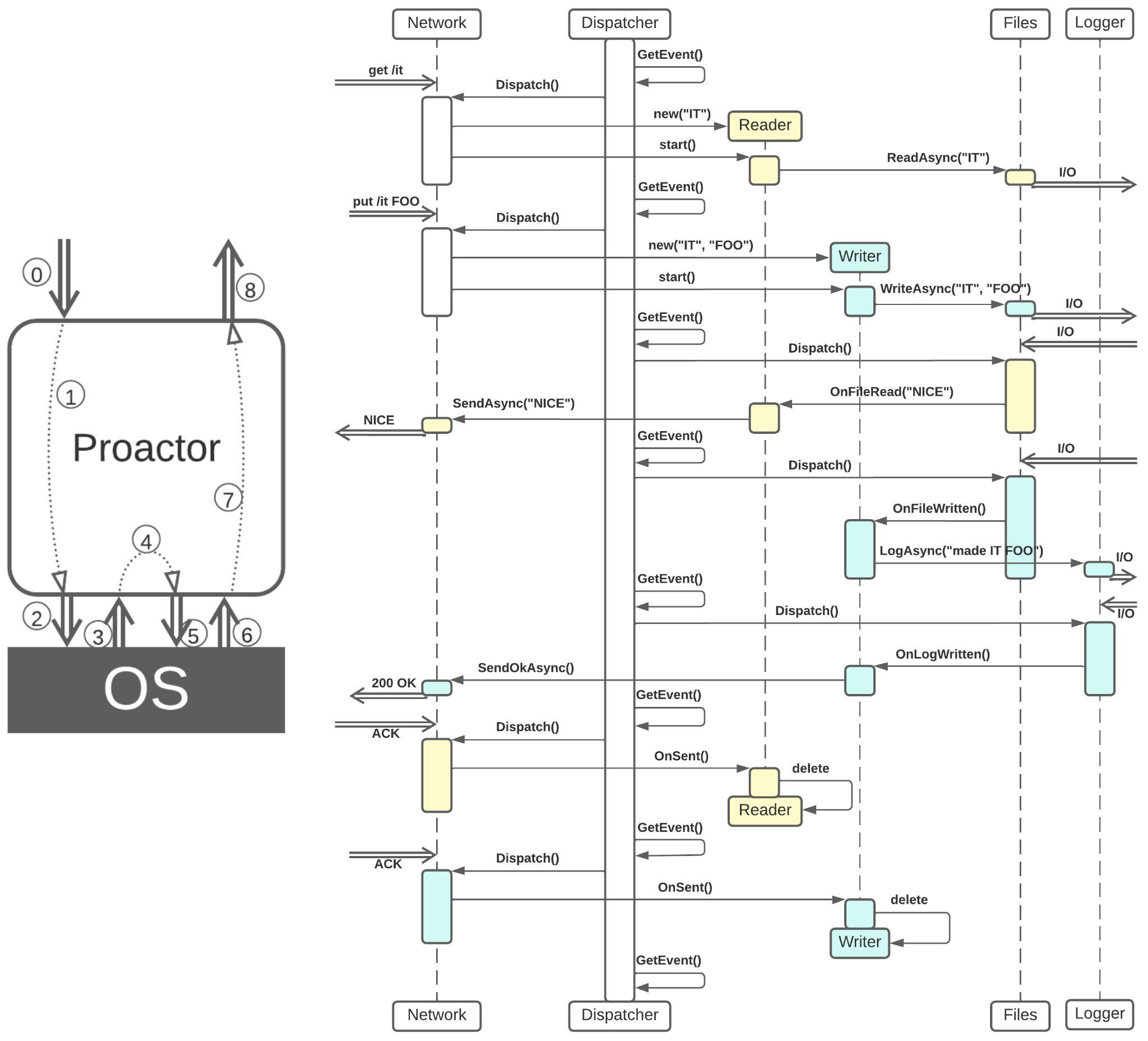
* The throughput is low – only one request is being processed at any given moment.

*Drawbacks (multi-threaded variation):*

* The system is indeterministic, making it impossible to reproduce bugs or restore the reactor’s state by replaying the incoming requests.
* Threads use much resources and are quite slow to switch.
* If the reactor has a mutable state, it should be protected with mutexes, sometimes resulting in performance degradation or even deadlocks.
* Tasks cannot easily interact.

Though multithreaded reactors are common for backends, [hardware](http://ithare.com/chapter-vd-modular-architecture-client-side-client-architecture-diagram-threads-and-game-loop/) and [database](http://ithare.com/multi-coring-and-non-blocking-instead-of-multi-threading-with-a-script/3/) adapters often need to be single-threaded in order to both protect the managed device from conflicting requests and optimize its performance. In that case, the reactor blocks for the duration of the hardware’s processing of the request, and as soon as it gets a response from the hardware, the reactor proceeds with accepting and running the next request for the wrapped device.

### Proactor [[POSA2](#kix.hefqusybmb4l)] (asynchronous messages)



***Multitasking without multithreading – reactive programming.***A single thread handles all the events: the incoming requests and the responses from the underlying hardware or libraries. All the outgoing communication is asynchronous (even reading/writing to HDDs); thus, the thread never blocks, every event is processed extremely quickly, the single thread is enormously productive and responsive, and resource usage stays low. It is even possible to keep the proactor thread warm (busy waiting on the event queue) to improve the response time. The proactor’s state does not need to be protected by mutexes (as there is only one thread inside the proactor).

As incoming events are serialized by the actor’s event queue, any request at any processing stage may change the proactor’s state, thereby [influencing the behavior](https://hillside.net/plop/2020/papers/poltorak.pdf) of all the subsequent events, including the events involved in parallel tasks. The execution is [deterministic](http://ithare.com/multi-threading-at-business-logic-level-is-considered-harmful/) – if incoming messages are recorded, it is possible to replay them later to reproduce bugs or restore the state of the proactor in case of hardware failures (*event sourcing* [[MP](#kix.hccjphx9ia0l), [DDIA](#kix.q1vb5yx6icby)]).

All the benefits come at a price: the code for every use case is [dispersed](https://hillside.net/plop/2020/papers/poltorak.pdf) over multiple asynchronous event handlers, making use cases very hard to understand and/or debug – it’s the dark side of reactive programming.

The approach is good for *control flow*, when it is important to govern system state and make changes in real time to scenarios that are already running. The drawback of being limited to a single CPU core is usually unimportant for control flow logic, as no big data processing is involved. It is also quite natural for control-flow-dominated systems not to have explicitly coded use cases (as there are so many possible interrupts or branches that any imperative programming attempts turn into a nightmare), in which case there is not much to lose in code structure.

*Proactor* is opposed to *Reactor*: Reactor has simple business logic code, but it is unresponsive and RAM-heavy if multithreaded. Proactor, on the other hand, sacrifices the readability of business logic for flexibility, real-time responsiveness and low resource consumption.

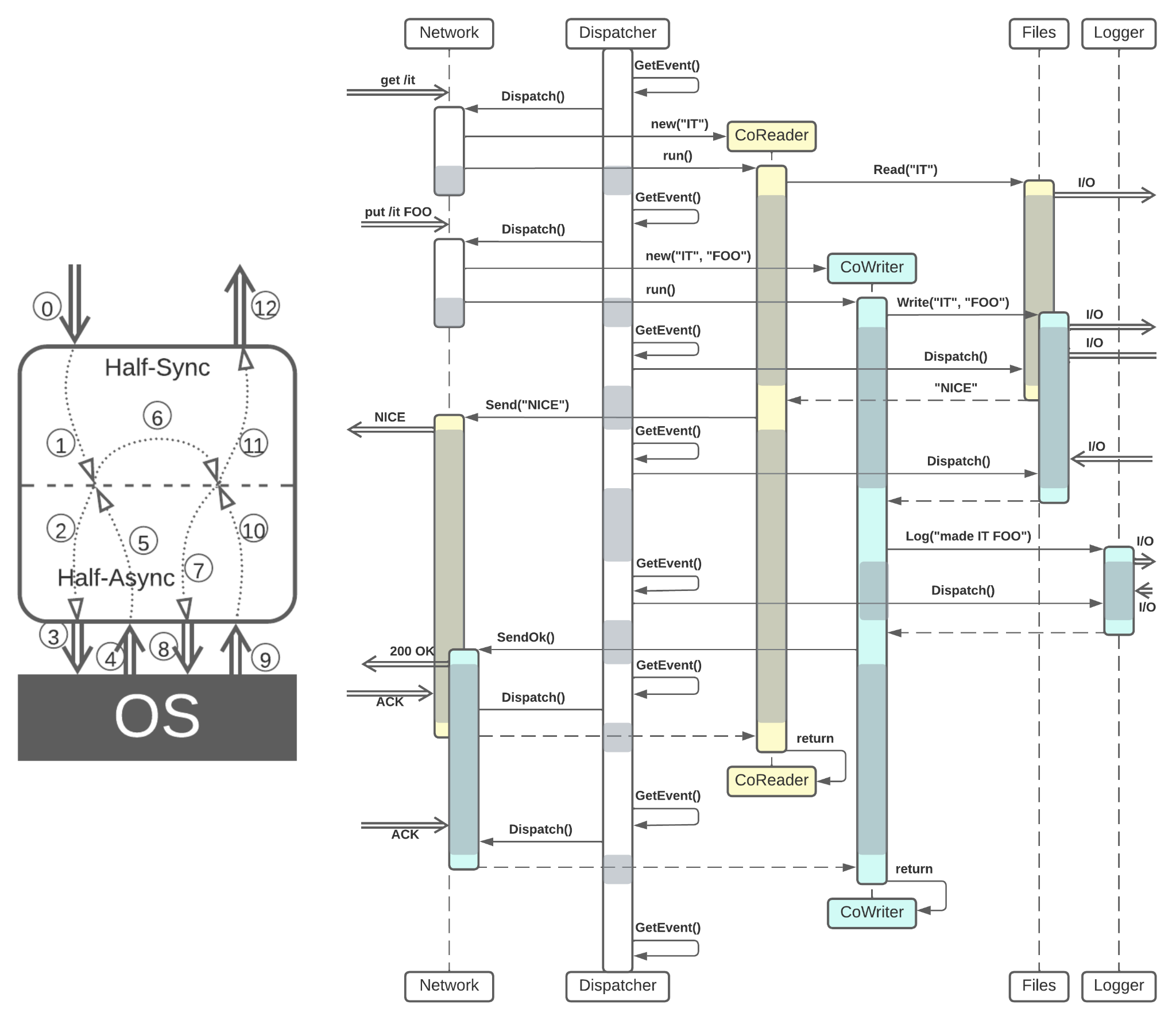
*Benefits:*

* Tasks may easily influence each other during execution via the actor’s state.
* There are no thread context switches, leading to maximal responsiveness.
* The resource (CPU, RAM) consumption for each running task is minimal, which maximizes the number of requests that can be processed in parallel.
* The system is deterministic, which means that [bugs are reproducible](http://ithare.com/deterministic-components-for-interactive-distributed-systems-with-transcript/).
* Multicasting is natural – any step of request processing may send out several requests in parallel, with response handlers collecting the data and continuing the task as soon as all the data is available.

*Drawbacks:*

* Use cases gain extremely ugly code and are hard to debug. The source code is grouped by messages (including responses from the periphery) it handles, not by scenarios of business logic.
* A single CPU core is used. It is possible to run a proactor multithreaded, but doing so will devoid the actor of most of the proactor’s benefits.

### Half-Sync/Half-Async [[POSA2](#kix.hefqusybmb4l)] ([coroutine-based reactor](http://ithare.com/multi-coring-and-non-blocking-instead-of-multi-threading-with-a-script/2/))



*(The periods of time when the CPU serves another stack are grayed out)*

The upper half contains multiple *reactors* (implemented by threads or coroutines) which block on calls to the underlying *proactor*. The proactor’s thread serves events from the system (OS, libs) and may also serve the calls from the reactors, acting as an RPC implementation layer. The system as a whole represents a middle ground between Reactor and Proactor; hence its description being placed alongside them.

*Half-Sync/Half-Async is, in actuality, Layers plus Sharding (both of which are approached in Part 3 of the publication) or, rather, a kind of Microkernel (from Part 4). However, as it is used for message handling, it is discussed here.*

***Multithreading without multithreading – imperative tasks over a reactive engine.*** When implemented with coroutines (a *coroutine* is a call stack that does not own an execution thread), *Half-Sync/Half-Async* keeps the benefits of single-threaded actors, namely: determinism and low resource consumption. In this case, a single execution thread switches between its original stack in the lower (proactor) half and multiple coroutine (reactor) stacks of tasks running in the upper half. Every incoming request is turned into an upper-half coroutine and run as a synchronous (blocking) chain of calls in the actor’s single thread. However, any blocking call from the synchronous scenario yields the thread to the lower non-blocking half instead of going into the OS kernel. And when the lower half gets a response from the OS to a request running on behalf of one of the upper half’s coroutines, it yields the execution thread to the coroutine in what looks like a return from a blocking call. Thus, *Half-Sync/Half-Async* kind of reimplementsOS threads, a runtime and a scheduler in user space.

*Benefits (for a single-threaded coroutine-based implementation):*

* The code in use cases is simple.
* Debugging is easy.
* Context switching is relatively cheap.
* There is relatively low RAM consumption per running request.
* The system is deterministic, which means [bugs are reproducible](http://ithare.com/deterministic-components-for-interactive-distributed-systems-with-transcript/) with event replay.
* A task may cancel other tasks by throwing exceptions into their (blocked) call stacks.

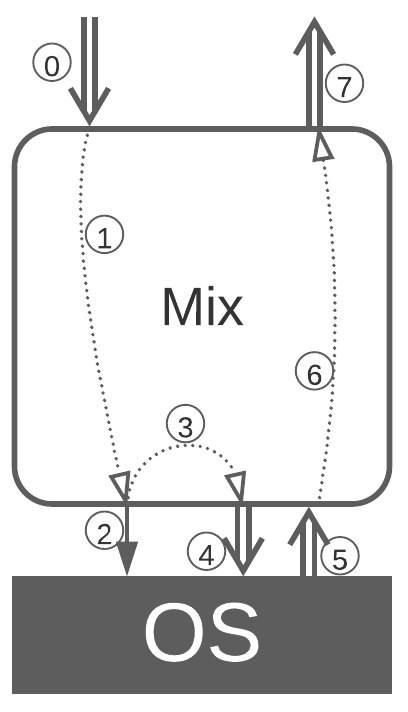
*Drawbacks:*

* A complex infrastructure code is needed to support the coroutines engine.
* A task can usually only cancel other running tasks, not change their behavior, as the tasks are written in the active (imperative) paradigm.
* A single CPU core is normally used. It is possible to utilize multiple threads; however, doing so will lead to indeterminism, the need to protect shared resources, and possible troubles with inter-task interactions.
* Multicasting is not easy (as it requires some dedicated code in the async layer).
* Coroutines may not be supported with older compilers or exotic hardware.

*Here we have yet another notion that haunts us throughout this cycle of publications: if there are two options, there’ll usually be a third one holding the middle ground, featuring some of the benefits of both but often being quite complex to implement. This is on top of the trouble that stems from many architectural patterns potentially being considered special cases of other patterns, generalizations of third ones, or combinations of several fourths and fifths, which is further complicated by the fact that most of them are known by quite a few unrelated names.*

*Half-Sync/Half-Async* is more complex than both *Reactor* and *Proactor*, but it may still be a good choice for larger projects if the costs of implementing and supporting the async infrastructural layer are offset by lowering the resource consumption of *Reactor* without losing the simplicity of the synchronous business logic. This is similar to RPC’s status as an adapter between procedural synchronous calls and reactive asynchronous messaging.

### A mixed case



Some outgoing requests (like writing to an SSD or communicating over Thunderbolt) are quick and blocking, while others are asynchronous. Combining blocking and non-blocking calls has been [described](http://ithare.com/multi-coring-and-non-blocking-instead-of-multi-threading-with-a-script/2/) as an optimization that minimizes the amount of context switching and simplifies the high-level code. The same approach often seems to be used in microservice systems, where communication between a microservice and its database may be synchronous while the microservices send asynchronous messages to each other.

## Request State

Every request processing method uses RAM to store the task’s *state* (progress and data):

* With a (multi-threaded) *Reactor*, the request’s state resides on the stack of the thread that processes the request. This makes it impossible for requests to influence each other, as the stack structure is opaque (OS-dependent). In many cases, the detriment that arises as a result of high resource consumption by multiple threads is balanced out by the fact that the code for business logic (tasks) is easy to read and debug, being written in the imperative style.
* With a (single-threaded) *Proactor*, everything is stored as explicit data structures in the actor’s state, together with the actor’s own static variables. This makes it easy for one task to access the states of other tasks. Moreover, as every step of a task starts as a callback from an event handler, a task must explicitly read and check its own state at every step; thus, there is no way it will miss the changes to its state made by some other task. It’s paid for with the fragmented business logic code characteristic of reactive programming.
* There is an intermediate case for a (single-threaded) coroutine-based *Half-Sync/Half-Async*: the states of tasks reside on call stacks (like with *Reactor*), but the programming language-dependent coroutine runtime provides ways to interact with blocked coroutines. The business logic code is simple, though it comes at the cost of the complex extra infrastructure layer that reimplements the functions (thread scheduling, blocking calls over async I/O), which are normally provided by the OS.

## Summary

* The difference between *data flow* and *control flow* was identified.
* The most common approaches to request processing were analyzed in detail.
* The interplay between the ways request states are stored, the possibility of running requests to interact and the complexity of business logic code were hinted at.

In the subsequent installments, we’ll investigate the ways to split a monolith by asynchronous interfaces (or to assemble a system from several monoliths).

## References

* 1. [DDIA] Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. *Martin Kleppmann. O’Reilly Media, Inc. (2017).*
  2. [MP] Microservices Patterns: With Examples in Java. *Chris Richardson.* *Manning Publications (2018)*.
  3. [POSA1] Pattern-Oriented Software Architecture Volume 1: A System of Patterns. *Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. John Wiley & Sons, Inc. (1996).*
  4. [POSA2] Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects. *Douglas C. Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. John Wiley & Sons, Inc. (2000).*

* 1. [POSA5] Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages. *Frank Buschmann, Kevlin Henney, Douglas C. Schmidt*. *John Wiley & Sons, Ltd. (2007).*
  2. *Editor:* [Josh Kaplan](mailto:joshkaplan66@gmail.com)
  3. *Part* 1 **2** 3 4 5
  4. Creative Commons License
  5. This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-nc-sa/4.0/).