Half-Proactor/Half-Async Architecture for Real Time Device Management[1]

DENYS POLTORAK, Keenetic Limited.

Traditionally, embedded and soft real time systems are built with the asynchronous Actors approach. However, as system complexities and amount of business logic increase, asynchronous code becomes forbiddingly hard to handle. The present article proposes a non-blocking semisynchronous architecture featuring fast event response times, deterministic behavior, straightforward coding and better support for debugging. An overview of an embedded telephony gateway application is provided as an example of Half-Proactor/Half-Async, a specialization of Half-Async/Half-Async architecture.

Categories and Subject Descriptors: **•Software and its engineering~Software organization and properties~Software system structures~Real-time systems software•Software and its engineering~Software organization and properties~Software system structures~Software architectures~Publish-subscribe / event-based architectures•***Software and its engineering~Software organization and properties~Software system structures~Embedded software•Software and its engineering~Software organization and properties~Software system structures~Software architectures~Layered systems*•Software and its engineering~Software organization and properties~Software system structures~Software architectures~Object oriented architectures

General Terms: Architectural patterns

Additional Key Words and Phrases: Actor model, non-blocking programming

ACM Reference Format:

Poltorak, D. 2020. Half-Proactor/Half-Async Architecture for Real Time Device Management. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 27 (October 2020), 10 pages.

1. INTENT

Real time systems are traditionally implemented with the Actors approach. However, the inherent asynchrony of the Actors is both a blessing and a curse: a blessing when it comes to fast and non-blocking event processing, and a curse as soon as use cases require complex cooperation from several actors. Below is described a paradigm shift from the Actors' asynchronous business logic halfway towards the Half-Sync/Half-Async's convenient programming. It removes most of the Actors' code complexity without sacrificing non-blocking event processing or performance and even gains several useful properties, including abstraction layers and determinism, on the way.

2. INTRODUCTION

Embedded software is a wide and diverse area of engineering that covers everything from 8-bit IoT controllers with 1KB RAM on board to high-end multicore systems for the automotive and aerospace industry. On the other hand, the number of developers in this field is relatively low, which results in the lack of proven ready-to-use approaches when compared to more popular specializations like web or mobile programming. One of the consequences is mouth-to-mouth knowledge transfer, with company traditions lasting for generations of developers; another one is that a "craft", not "scientific" approach is used (POSA5).

One of the challenging areas in embedded programming is real time systems, where something bad happens if incoming events are not responded to quickly enough. By definition, real time programming is event-based, with the software usually modelling its target real-world system to be able to react to signals fast (with remote queries not being an option due to the timing

[1]Author's email: descri@gmail.com.

restrictions). Traditionally, soft real time systems are built according to the Actors paradigm (see Lohstroh at al. 2019 for review), with every modelled entity living in a dedicated thread or fiber, all data being private. However, as time goes on, users expect more complex behavior, thus the amount of business logic and number of participating components increases, making asynchronous code prohibitively complex and often unstable. The present article reviews the polar approaches of Actors and Half-Sync/Half-Async and establishes a middle ground with the advantages of both.

2.1 Sample System

A telephony gateway makes a good case to compare the architectures: it is relatively simple and conforms to standards while serving events from multiple independent sources. Real-time constraints are weak, but there is a heavy dependency of control flow on components' states. Let out sample system contain a (minimal) set of:

- 2 *SIP accounts* (lines), registered with IP telephony servers
- 3 *DECT handsets*, registered with a local USB DECT base
- *Calls*, created as needed or taken from an object pool
- *Phonebook*, in RAM, backed up to flash
- *Calls history*, in RAM, backed up to flash

We will consider the next example event sequence:

(1) SIP INVITE - an incoming call that should result in a {CC-SETUP} message sent to each handset and "100 Trying" sent back to the server. The application should also match the caller number in the phonebook and in case of success send contact name to all the handsets in a later {CC-INFO} message.

(2) Handset 2 starts ringing and sends back {CC-ALERTING} which should be translated into "180 Ringing" towards the SIP server. If the caller's number was found in the phonebook, now it's time to send it in a {CC-INFO} message.

(3) At the same time SIP CANCEL is received from the server. The application should send a {CC-RELEASE} message to each of the handsets, "487 Terminated" to the server, and store the call's data (caller's name, number and call start time) in the calls history.

2.2 Actors Approach

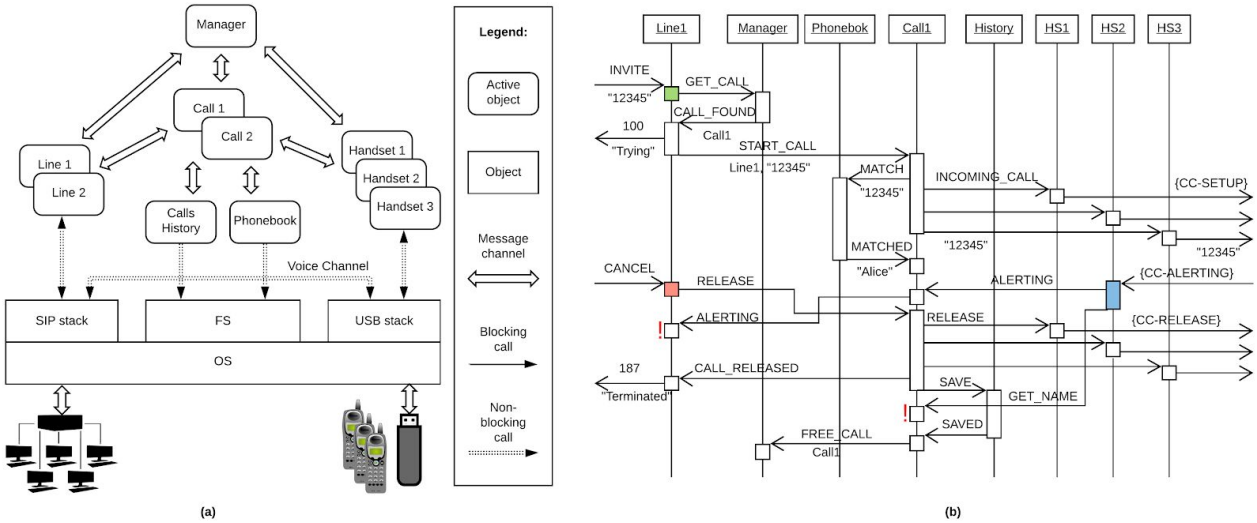

Fig. 1. Actors. (a) structural diagram for the sample system. (b) sequence of messages for the example use case, outdated messages are marked with !

Actors (Figure 1a) is a completely asynchronous architecture where domain entities are mapped to Active Objects (POSA2) which interact via messaging. Each Actor lives in its own thread or fiber and acts as a stateful proxy (GoF) for its real-world counterpart. This results in good system responsiveness, but any non-trivial business logic is hard to implement (async messages don't have return types, are not supported by common IDE code analysis tools, and each async step needs to make precondition checks for the actor's state as some unrelated event may have changed it) and debug (a simple use case turns into scores of independent message handlers in different threads; moreover, multithreading brings in non-determinism, making event replay impossible), see Figure 1b. Also, by default the Actors approach doesn't bring in any abstraction layers, so changing HW or SW vendor of system components may be painful. Thus, the Actor model is very good for simple systems, but any tightly coupled business logic does not scale. It is somewhat similar to the Microservices approach (Fowler 2015).

2.3 Half-Sync/Half-Async Approach

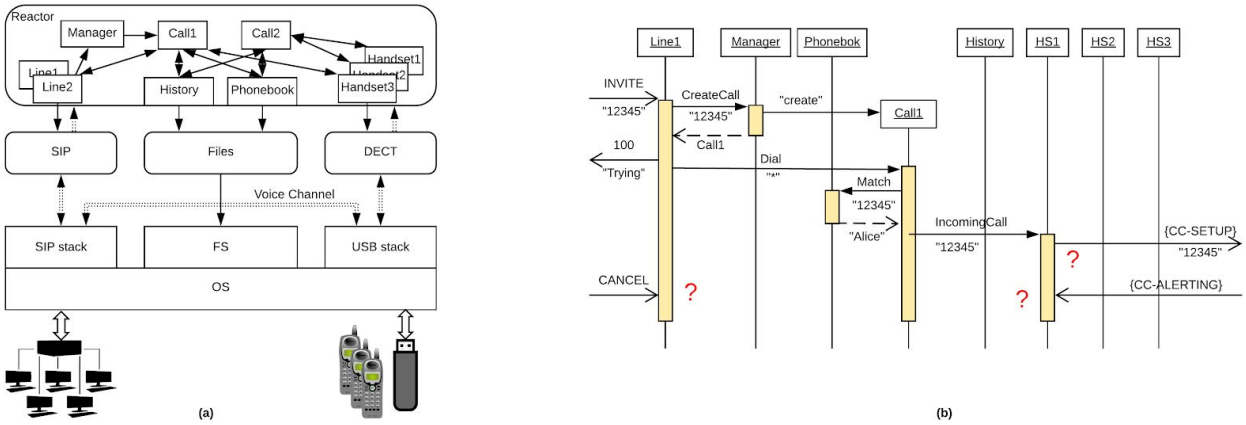

Fig. 2. Half-Sync/Half-Async. (a) structural diagram for the sample system. (b) method call sequence for the example use case, next critical issues are marked with ?: inability to multicast, cancel scenario request, intermediate message not obeying request/confirm paradigm.

Half-Sync/Half-Async (POSA2) is a synchronous Reactor-based multithreaded architecture mostly used for web servers. Application logic is very easy to implement and debug, thanks to synchronous request handling, when there is no shared state at the business logic level. As an extra bonus, it encapsulates application logic in a separate layer, providing good vendor abstraction (Figure 2a). It also does scale well. However, as Figure 2b shows, it does not work under multiple event sources and shared resources conditions, as scenarios started by one event may need to be cancelled midway and rolled back by another event. Implementing such a roll-back capability requires a complicated scenario/thread reflection, if possible at all. Use of shared resources would also require thread synchronization, which moves us further away from good code (Ignatchenko 2015) and fast response times. Moreover, whenever events from different sources spread over the system in opposite directions, deadlocks are very likely. Another complication is that request/confirm interface is required for all the devices (to wake up the request processing thread that waits for the confirm) while many real-world systems send indications which don't directly map onto the request/confirm paradigm.

Here should be mentioned a coroutines-based approach with application logic residing in a single thread (Ignatchenko 2018), which makes the transition from Half-Sync/Half-Async to Half-Async/Half-Async (POSA2). With the coroutines the Reactor layer that contains the entire application logic is split into the upper half, built of request handling use cases, each running in a

dedicated coroutine, and the lower half, where low-level business logic (services, device state machines, protocol implementation) and coroutines engine (subscription for events, resuming the correct coroutine(s) if any is subscribed, exception generation if supported) reside. This way the entire logic runs in a single thread, so no mutexes are needed, no deadlocks possible, and all the state is synchronously accessible. However, the Half-Async/Half-Async architecture is still impractical for most embedded/telecom projects because both the request/confirm API restriction applies, and supporting multiple event sources, shared resources and event multicast blurs the original vision of coroutines to such an extent that the code may become very complex.

3. HALF-PROACTOR/HALF-ASYNC

As we have seen above, application logic in the Actors model is hard to scale because of its inherent asynchrony, while easily scalable Half-Sync/Half-Async does not fit several of the domain's requirements. Is there any venerable pattern between the Active Object (Actor) and the Half-Sync/Half-Async's upper Reactor (POSA2)? Surely, it is Proactor (POSA2) featuring single-threaded (no synchronization) processing of events from multiple sources. Below are the steps taken to transform an Actors architecture into Half-Proactor/Half-Async (Figure 3):

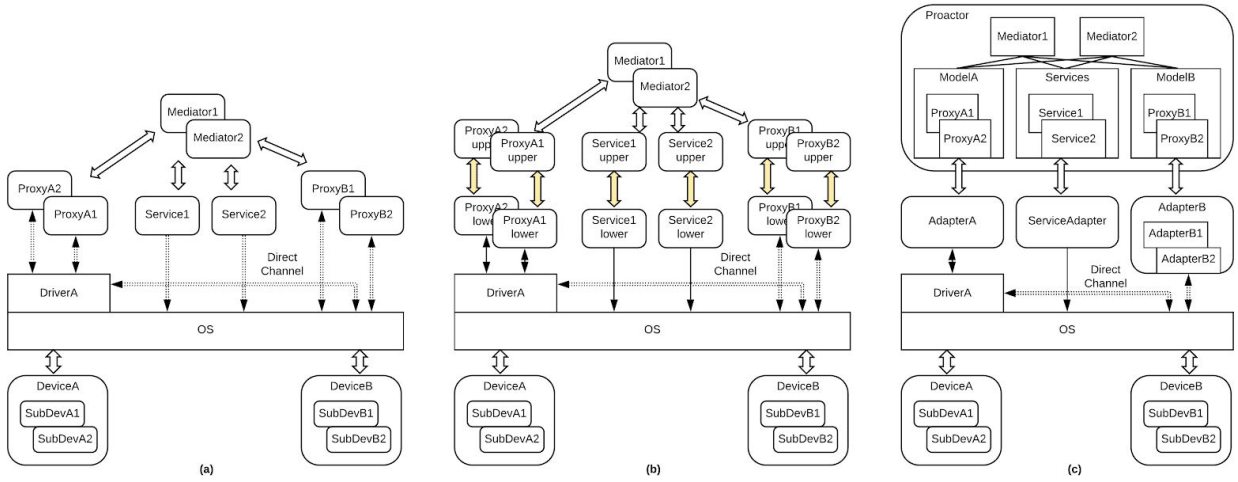

Fig. 3. Transformation from Actors to Half-Proactor/Half-Async. (a) initial Actors system. (b) actors split into upper and lower halves. (c) merged Half-Proactor/Half-Async.

(1) Split each actor that works with the outer world (network, hardware, FS) into the upper (logic and state) and lower (vendor-specific) halves using Half-Object Plus Protocol pattern (POSA4), Figure 3a-b.
(2) Merge all the upper halves of the splitted actors and the remaining internal actors into a huge single actor responsible for the entire application's logic. This creates the upper layer of Half-Proactor/Half-Async, Figure 3c.
(3) Remove most of the state from and merge the lower halves of same-type actors, creating the lower (vendor abstraction) layer of Half-Proactor/Half-Async, Figure 3c.
(4) Provide a message dispatch engine, see below.

3.1 Message Dispatch Options

There are 2 ways to deal with the messages from the lower layer to the Proactor (Figure 4a and b). In the first approach, each of the upper side Actor halves is left with its own message queue, so that the structural changes on moving from Actors to Half-Proactor/Half-Async are minimal (Figure 4a). As all the message channels are handled by the same thread, no synchronization is

needed. The advantage of this separate message channels approach is its extreme simplicity and very low system requirements. The disadvantage is its lack of hierarchical structure and determinism.

The second approach is creating in parallel a composition hierarchy for the application modules (Figure 4c) and the corresponding inheritance hierarchy for the messages (Figure 4d). The message dispatch is done via recursive Visitor (GoF). This way messages from a single event queue can reach any of the app modules (Figure 4b). It is better structured (thus more scalable) and deterministic (the messages are always processed in the order they are put to the Proactor's event queue) but requires language support (C++ for Visitor). If the system is built from scratch, this approach is recommended. It should be noted that if there exist multiple objects (models, proxies, adapters) of the same kind, the simple Visitor dispatch would not work, and the dispatched message should contain id of or pointer to its destination.

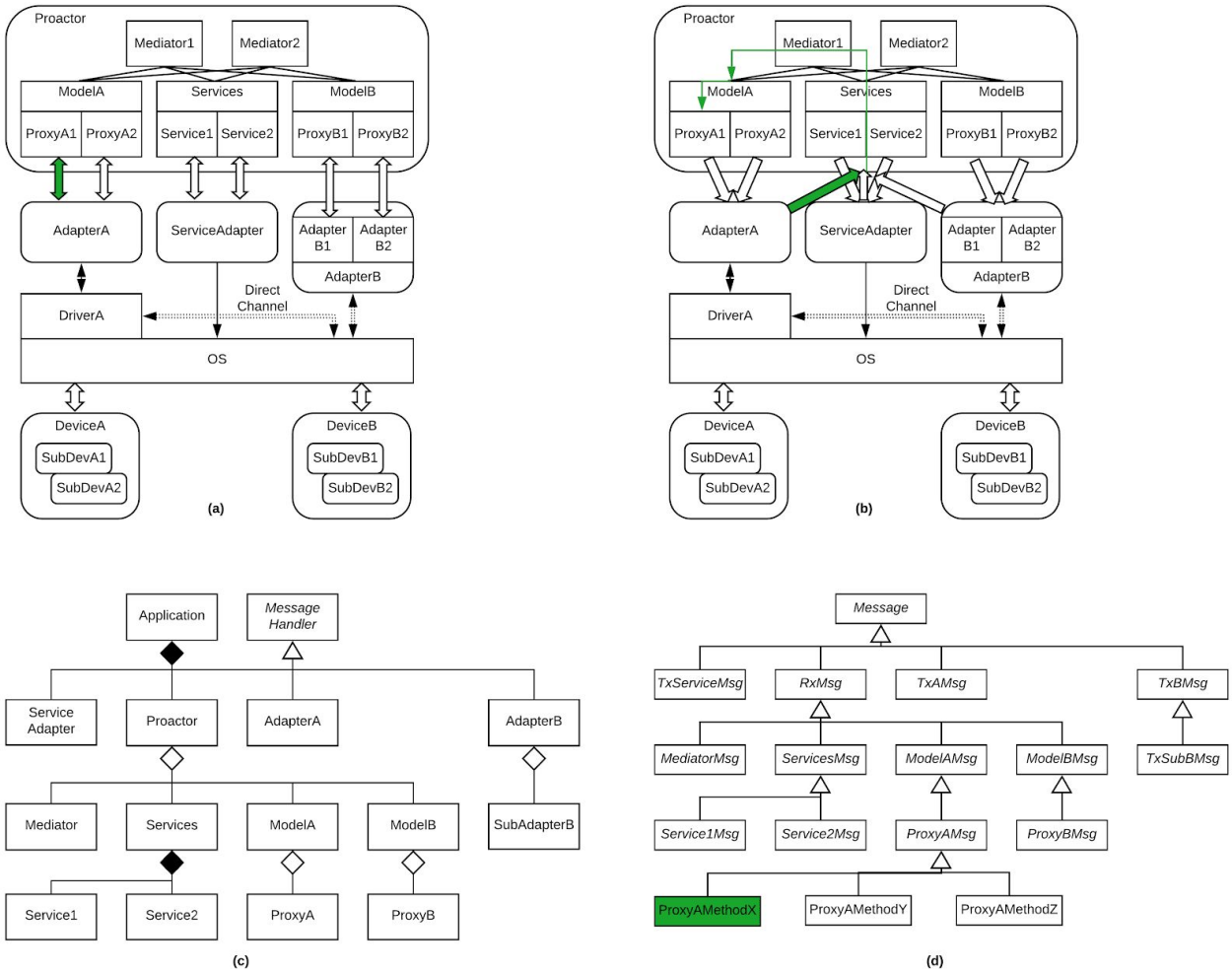

Fig. 4. Message Dispatch for Half-Proactor/Half-Async. (a) multiple message channels. (b) Visitor-based dispatch. (c) application structure. (d) message hierarchy, most leaf classes omitted. One message dispatch is shown in color.

3.2 Use Case Revisited

After describing the system we may consider how it handles our sample use case (Figure 5). The entire scenario is processed by the logic layer in 3 synchronous steps (which equals the number of handled events) compared to 10+ steps with Actors. State checks on entering the event handlers

are simpler than for Actors because the entire system's state is guaranteed to be consistent (any previous event has been fully processed and all the involved proxies were notified). Moreover, the entire set of design patterns becomes available to the application developer, unlike with Actors where the modules communicate with asynchronous messages.

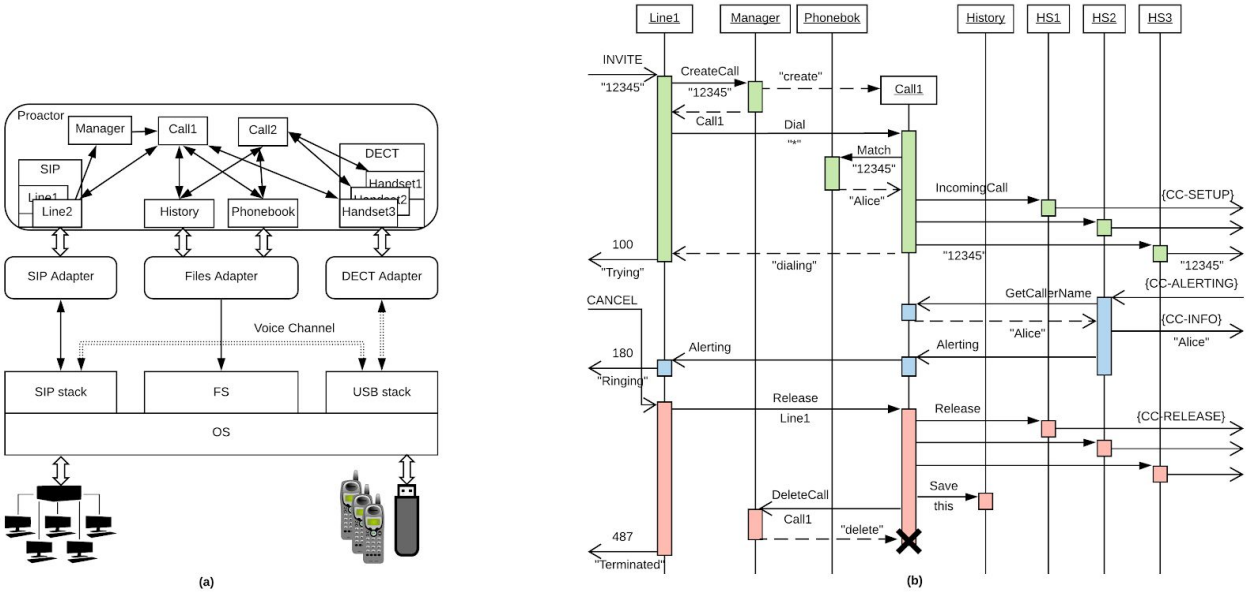

Fig. 5. Half-Proactor/Half-Async. (a) structural diagram for the sample system. (b) method call sequence for the example use case, debuggable sequences shown in colors.

4. FORMAL PATTERN DESCRIPTION

Half-Proactor/Half-Async is a specialization of the Half-Async/Half-Async architectural pattern for use in soft real time systems with multiple event sources. For the context and alternative architectures see section 2.

These are the relevant forces:

- The software manages a set of different physical or logical self-contained devices and new device types may be added. Thus, abstraction layers and hierarchies are welcome.
- Events from the devices trigger application logic scenarios, which spread over the entire system of the devices in different directions. Easy access to all parts of the system's state is necessary.
- Multiple scenarios may run simultaneously and change states of shared objects, thus breaking each other's control flow. This prohibits explicit active scenarios (represented by threads or coroutines) in favor of implicit reactive scenarios.
- Both single device management (driver-like) logic and multi-device (request-like) use cases may grow arbitrarily complex and coupled, with their control flows depending on states of multiple components. Thus, coding and debugging application logic should be kept as simple and uniform as possible.
- The devices are limited, thus high resource competition is expected.
- Event response times should be kept low, so device states should be available in RAM and no blocking calls or long critical sections are allowed.
- System resources and compiler support may be very limited, so an implementation in C should be possible.

4.1 Applicability

The pattern should be used when:

- An existing Actors system grows out of control by getting coupled logic.
- Designing a new device management system in a tightly coupled domain for development speed, code scalability and maintainability.

The pattern should not be used when:

- Events spread over the system in uniform direction and control flow is predictable - in that case Pipes and Filters (POSA1) should be used as it gives more control over the system structure and threading model. An example is PX4 autopilot[2].
- There is a single request source, the requests are not cancellable, and the device interfaces comply with the request/confirm paradigm. Half-(A)Sync/Half-Async (POSA2) applies with its better structured code.
- The amount of application logic in use cases (Mediators) is much greater than in the device support (Models, Proxies, Services) layer. Aim for coroutines with Half-Async/Half-Async which simplifies code for use cases at the cost of very complex framework and device management (with workarounds for the request/confirm, cancel and multicast issues).
- Use cases are simple and don't involve steps relying on states of multiple remote devices (low coupling). The supported hardware interface is known to never change. Use Actors as a simpler alternative.
- The business logic may hit the single CPU core performance limit, even while data is passed via Direct Channels. If there are highly loaded use cases which don't rely on most of the system's state, they may be separated into Direct Channels, each served with a dedicated CPU core. Otherwise, try sharding. As the last resort, return to Actors or Pipes and Filters that may utilize multiple cores or distributed computing for business logic processing.

Feasible domains include: telecom, high-level IoT, robotics.

4.2 Structure and Interactions

The pattern describes an *application* managing external physical or logical *devices*. The application, as shown in Figure 4b, consists of:

- *Proactor* that contains all the business logic, runs non-blocking in a dedicated thread and exchanges messages with the lower layer modules, namely Adapters.
- *Adapters* between the Proactor and underlying OS or vendor-specific libraries and protocols. The Adapters provide messaging interface towards Proactor and may use blocking or non-blocking calls to the underlying OS or libraries. Mutexes may be needed for multithreaded libraries. In case of low-level device protocol (e.g. RPC for HW register access) the device's Adapter may serve as a device driver which turns the Adapter itself into a quite complex proxy module.
- *Direct Channels* that provide for highly efficient data transfer between the managed devices by making shortcut paths at device support stack or even OS drivers level.

Proactor is built of the following parts:

- *Proxies* that store the last known state of corresponding devices, receive notifications from the devices via device type adapters and send back commands. For domains featuring polymorphic device behavior (like telephony) it may be convenient to split a proxy into the lower half dealing with the device type (or standard) protocol and polymorphic upper half with most of the business logic for the proxied device.

[2] https://dev.px4.io/master/en/concept/architecture.html

- *Services* that provide non-blocking access to resources (file system, remote database, etc.).
- *Models* that describe for the application and manage a group of similar devices or resources.
- *Mediators* that contain the highest level application logic (serving request-like use cases) for connecting and managing the Proxies and Services.

4.3 Consequences

Half-Proactor/Half-Async has the next benefits compared to Actors:
- Application logic scales well, even when it is tightly coupled and involves multiple objects.
- Application logic is platform-agnostic and can be debugged on a desktop PC.
- Application logic is deterministic, event recording and replay are easy to implement.
- Protection from vendor or OS lock-in.
- Several levels of polymorphism are provided for device management, reducing the amount of code for and simplifying the addition of new types of devices.
- Faster and much simpler processing of events that rely on state of or trigger actions for multiple devices.

Half-Proactor/Half-Async can be used in a wider range of systems compared to Half-Async/Half-Async because:
- The supported devices are not required to be managed with a strict request/confirm paradigm.
- Multiple request (or indication for event-driven systems) sources are supported without any extra code (like thorough state checks and/or coroutine reflection).
- Requests are easily cancellable by any involved party at any stage (compare to coroutine reflection and rollback with Half-Async/Half-Async).
- There are no specific compiler requirements.

The next drawbacks remain:
- Application logic is harder to debug than in a fully synchronous implementation (Half-Sync/Half-Async), inherited from Actors.
- Infrastructure code is complex (compared to Actors), inherited from Half-Async/Half-Async.

As we see, Half-Proactor/Half-Async clearly wins over Actors in development speed as soon as business logic becomes non-trivial (development cost for the logic is higher then for the framework) or hardware tends to change often (requiring abstraction layers). The coroutines-based Half-Async/Half-Async would have been a viable alternative were it applicable to generic event-based systems (it was designed for a backend-style environment with a single Adapter and multiple Services; any deviations from this model make both the coroutine support framework and the lower-level application code much more complex).

Another interesting observation is that both recommended architectures for complex real time or high load systems are derived from Half-Async/Half-Async, with Half-Proactor/Half-Async being simple and flexible, while the coroutines approach is heavily inclined to optimize the code for request handling scenarios at the cost of much extra complexity at the lower layers. It is likely that more Half-Async/Half-Async variants exist and they will be discovered and documented for other kinds of demanding software systems.

4.4 Known Use

SIP<->(DECT|FXS) gateway application in Keenetic routers is based on Half-Proactor/Half-Async[3].

4.5 Related Patterns

Half-Proactor/Half-Async is a blend of Half-Async/Half-Async (which is itself a variation of Half-Sync/Half-Async (POSA2)), Proactor (POSA2) and Active Object (POSA2) patterns.

[3] https://dou.ua/lenta/articles/telecom-application/

The patterns used by Half-Proactor/Half-Async are:

- Layers (POSA1), used recursively: first for structuring the system into the Sync layer with application logic and Async layer with vendor-specific code, and later to divide the generic application Mediator code from the device type - specific Proxy code.
- Caching Proxy (GoF), used to store states of the managed devices to allow for synchronous control flow decisions.
- Half-Object Plus Protocol (POSA4), applied recursively: first to split the device Actors into upper and lower half, and later it may be used again to split the upper half of the Proxy into a generic interface facing the Mediator and a device type - dependent lower half facing the Async layer.
- Mediator (GoF), for the Sync part connecting the proxies.
- Message Passing Active Object (POSA2) aka Actors paradigm for all the modules in the Async layer and for the Sync layer object itself.
- Recursive Visitor (GoF) or Reactor (POSA2) plus Message Channel (POSA4) for message dispatch.

The patterns reviewed in the course of this article differ in which architectural entities receive threads (or coroutines). With the Actors approach domain entities are given or subscribed to threads. With Half-(A)Sync/Half-Async user requests and device drivers get threads. With Half-Proactor/Half-Async user logic is single threaded (because the domain itself is so tightly coupled that any borders inside the domain representation make lots of trouble) while each of the device drivers gets a thread. With Pipes and Filters event or data processing steps are the entities which may (or may not as threading here is very flexible) run in their own threads.

All these patterns exist because multithreading greatly improves response times and may often help with throughput, but the drawback is that the communication between threads becomes much more complicated than between entities in the same thread. And it takes both skill and experience to find the correct abstractions and divide the system in such a way that multithreading turns beneficial without making the code too complex to survive multiple years of active development.

5. DISCUSSION

Table 1 Comparison of Different Architectures for a Device Management Application

Architecture	***Half-Sync/Half-Async***	***coroutines-based Half-Async/Half-Async***	***Half-Proactor/Half-Async***	***Actors, loosely coupled logic***	***Actors, tightly coupled logic***
Higher-level app logic (request handling scenarios)	✓Quite simple (but beware of mutexes)	✓Quite simple (some state checks)	✗Moderate (few async steps)	✗Moderate (more async steps)	✘Hard (many async steps + state caching)
Lower-level app logic (device support and protocols)	✔Simple (local state change, maybe run a use case thread)	✘Hard (explicitly considers possible interactions between the scenarios)	✔Simple (global state change by a direct call to higher level)	✔Simple (local state change, send messages)	✔Simple (local state change, send messages)
App logic scalability	✗Moderate (mutexes)	✔Good	✔Good	✗Moderate	✘Poor
Async steps (all actors involved)	✔O(1)	✓O(NumEvents)	✓O(NumEvents)	✗O(NumEvents * NumActors)	✘O(NumEvents * NumActors^2)
State caching required	✔No	✔No	✔No	✔No	✘Yes
State change protection	✘Mutexes	✗State machines in lower layer and state checks after await	✔State driven behavior	✔State driven behavior	✗State driven behavior + cache invalidation
Framework complexity	✗Complex (adapters layer + RPC engine)	✘Very complex (adapters layer + coroutines engine)	✗Complex (adapters layer + message dispatch)	✔Simple	✔Simple
Vendor abstraction	✔Yes	✔Yes	✔Yes	✘No	✘No

Latency, same device	✘ Poor (mutexes)	✗ Average (messages + coroutines management)	✓ Good (messaging)	✔ Best (interrupt)	✔ Best (interrupt)
Latency, device to device	✘ Poor (mutexes)	✗ Average (sequential logic)	✔ Best (async multicast)	✓ Good (some messaging)	✓ Good (lots of messaging)
Message replay	✘ No (mutexes)	✔ Yes	✓ Yes if single queue	✗ Yes for any single actor	✗ Yes for any single actor

Table 1 compares the reviewed approaches in context of soft real time systems. Actors clearly win for small loosely coupled real-time systems (very low latency, low infrastructure cost, easy to implement in C); Half-Proactor/Half-Async is good for complex tightly coupled real time systems thanks to its mostly synchronous application logic and multiple abstraction layers. The coroutines-based Half-Async/Half-Async moves complexity from the use cases to the device and protocol support layer, making the latter very complicated when some of its prerequisites (Table 2) are not fulfilled, so it is used mainly for high-load backends (thin lower layer) and not for complex hardware management. Half-Sync/Half-Async mirrors Actors in that it is good for simple systems, but its use of mutexes becomes a burden when code complexity increases (Ignatchenko 2015), and it is not applicable for real time systems. Pipes and Filters approach is extremely flexible regarding threading, but it works only if all the events are processed in the same way, and it requires the system to be loosely coupled (next to no feedback capabilities).

Table 2 Prerequisites for the Mentioned Architectures

Architecture	***Half-Sync/Half-Async***	***coroutines-based Half-Async/Half-Async***	***Half-Proactor/Half-Async***	***Actors***	***Pipes and Filters***
Event sources	✘ Single	✗ Multiple with complications	✔ Multiple	✔ Multiple	✔ Multiple
Control flow	✘ Single direction	✗ Flexible (complex code)	✔ Flexible	✔ Flexible	✘ Single direction
Feedback	✔ Return value or exception	✔ Return value or exception	✔ Return value or exception	✗ Messages	✘ Requires extra pipeline
Device interface	✘ Request/confirm	✘ Request/confirm	✔ Event-based	✔ Event-based	✔ Event-based
Request representation	✗ Explicit (thread)	✗ Explicit (coroutine)	✔ Implicit (proxies' states)	✔ Implicit (actors' states)	✘ Static (pipeline structure)
Requests are cancellable	✘ No	✗ Exceptions and rollbacks with complex code	✔ Yes (global state change)	✔ Yes (local state change)	✘ No
Compiler requirements	✔ C	✘ C++20 or Boost	✓ C or C++	✔ C	✔ C
Intended use	Simple backend	High load backend or huge real time systems with many user scenarios	Tightly coupled complex real time systems	Loosely coupled or simple real time systems	Any systems with static control and data flow

REFERENCES

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. 1996. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. John Wiley & Sons.

Frank Buschmann, Kevlin Henney and Douglas Schmidt. 2007. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. John Wiley & Sons.

Frank Buschmann, Kelvin Henney and Douglas Schimdt. 2007. *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. John Wiley & Sons.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Martin Fowler. 2015. MonolithFirst. *martinFowler.com*

Sergey Ignatchenko. 2015. Multi-threading at Business-logic Level is Considered Harmful. In *Overload Journal* #128.

Sergey Ignatchenko. 2018. "Multi-Coring" and "Non-Blocking" instead of "Multi-Threading". ACCU 2018.

Marten Lohstroh, Martin Schoeberl, Andrés Goens, Armin Wasicek, Christo-pher Gill, Marjan Sirjani and Edward A. Lee. 2019. Invited: Actors Revisited for Time-Critical Systems. In *The 56th Annual Design Automation Conference 2019 (DAC '19), June 2–6, 2019, Las Vegas, NV, USA*. ACM, New York, NY,USA, 4 pages. https://doi.org/10.1145/3316781.3323469

Douglas C. Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann. 2000. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons.

<ccs2012>

<concept>

<concept_id>10011007.10010940.10010971.10011679</concept_id>

<concept_desc>Software and its engineering~Real-time systems software</concept_desc>

<concept_significance>500</concept_significance>

</concept>

<concept>

<concept_id>10011007.10010940.10010971.10010972.10010975</concept_id>

<concept_desc>Software and its engineering~Publish-subscribe / event-based architectures</concept_desc>

<concept_significance>500</concept_significance>

</concept>

<concept>

<concept_id>10011007.10010940.10010971.10010564</concept_id>

<concept_desc>Software and its engineering~Embedded software</concept_desc>

<concept_significance>300</concept_significance>

</concept>

<concept>

<concept_id>10011007.10010940.10010971.10010972.10010974</concept_id>

<concept_desc>Software and its engineering~Layered systems</concept_desc>

<concept_significance>300</concept_significance>

</concept>

<concept>

<concept_id>10011007.10010940.10010971.10010972.10010979</concept_id>

<concept_desc>Software and its engineering~Object oriented architectures</concept_desc>

<concept_significance>100</concept_significance>

</concept>

</ccs2012>