
# D2.1(b): Identified Adaptation Possibilities and Methods

Project acronym: ADMORPH
Project full title: Towards Adaptively Morphing Embedded Systems
Grant agreement no.: 871259

| | |
|---|---|
| **Due Date:** | Month 22 |
| **Delivery:** | Month 22 |
| **Lead Partner:** | UniLu |
| **Editor:** | Marcus Völp, UniLu |
| **Dissemination Level:** | Public (P) |
| **Status:** | final |
| **Approved:** | |
| **Version:** | 2.1 |

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871259 (ADMORPH project).

## DOCUMENT INFO – Revision History

| Date and version number | Author | Comments |
|---|---|---|
| 12/12/2020 ver. 1.0 | Marcus Voelp | First draft |
| 21/12/2020 ver. 1.1 | Marcus Voelp | Internal review |
| 31/10/2021 ver. 2.0 | Marcus Voelp | First draft of Revised version D2.1b |
| 02/11/2021 ver. 2.1 | Marcus Voelp | Internal review |

## List of Contributors

| Date and version number | Beneficiary | Comments |
|---|---|---|
| 12/12/2020 ver. 1.0 | Marcus Voelp | Initial report structure |
| 31/10/2021 ver. 2.0 | Marcus Voelp | Revised version D2.1b |

## GLOSSARY

**BFT-SMR** Byzantine Fault Tolerant Statemachine Replication

**CPS(oS)** Cyber Physical System (of Systems)

**FIT** Fault and Intrusion Tolerance

**IPC** Inter-process communication

**QoS** Quality of Service

**SoS** System of Systems

# Contents

**Executive summary** 4

**1 Towards Individualistic and CPSoS-wide Resilience** 5
1.1 Fault and Threat Models . . . 5
1.2 Timeliness and Synchrony . . . 6
1.3 Classical Resilience and Architectural Hybridization . . . 7
1.3.1 Architectural Hybridization: . . . 8

**2 Adaptive Resilience** 9

**3 Adaptation Opportunities and Methods** 10
3.1 Adapting the System along Attack Pathways . . . 10
3.2 Adapting Internal Resilience . . . 11
3.3 Adapting Functionality . . . 12
3.4 Adapting to Lowering Threat Levels . . . 12

**4 An Abstract View on Control** 12
4.1 Cascade Control Systems . . . 13
4.2 Complex / Simplex Controller Pairs . . . 14

**5 Resilient Control** 14
5.1 Accidental Fault and Attack Tolerant Control Design . . . 15
5.2 Cascaded Control and Safety Kernels . . . 16
5.3 Leveraging Complex/Simplex Pairs . . . 17
5.4 Replication Control . . . 17

**6 Conclusions** 19

**7 References** 19

# Executive summary

Workpackage WP.2 sets out to develop the adaptation building blocks necessary for maintaining or, in extreme situations, gracefully degrading the systems' quality of service guarantees. The focus is on methods, protocols, tools and techniques, to increase the resilience of controllers of Cyber-Physical-Systems of Systems (CPSoS), to optimize the mapping, partitioning and scheduling of system components, to automate the design transformation towards a reliable, resource and physical requirement aware system, and to analyze and limit system reconfiguration times.

This is an updated version of deliverable D2.1, called D2.1b. Aside from more in depth analysis of adaptation possibilities, we present a more concrete mapping to our envisaged consensual control architecture and on the interplay between the CPS-local runtime monitor and adaptation manager and the CPSoS-wide communication monitoring and adaptation.

Adaptation serves four main purposes:

1. to evade faults, including accidental ones and adversaries in their ongoing attacks;
2. to improve the resilience of systems after experiencing faults or during ongoing attacks, possibly by degrading the systems' quality of service, if necessary;
3. to return the system to a state at least as safe and secure as initially; and
4. to optimize the system whenever the perceived threat level drops (e.g., because the CPS entered less harsh environments or because adversaries lost interest).

We identified a tension between the classical world of real-time and embedded systems, focusing on predictability as first principle and hence known upper bounds for computation and communication latencies, and the body of knowledge on resilience mechanisms, in particular fault and intrusion tolerance. The latter operate in what researchers in that domain call asynchronous or partially synchronous systems and protocols, where such bounds are not given per se, in particular while the system is under attack.

Our goal is to reconcile both worlds and to research and develop adaptive resilience mechanisms, specifically for controllers, that are able to provide timing guarantees, not from the predictability of all components, but from enough components not being affected by adversaries both in the value and time domain.

# 1 Towards Individualistic and CPSoS-wide Resilience

Concerning the specific contributions of this deliverable, it first provides a review of the state of the art (SOTA) on adaptive resilience, from which it is possible to conclude about open issues and opportunities that we explore in ADMORPH. Then, the deliverable introduces some fundamental concepts on dependability and security, also describing baseline assumptions that pave the way for defining appropriate system and fault models. The deliverable is then devoted to describe our view on a generic fault and intrusion tolerant (FIT) control architecture, including a description of its components, their role and interplay. Our view on how this architecture will support resilient adaptation, and what kinds of adaptation we anticipate at this stage of the project, are then described.

## 1.1 Fault and Threat Models

Correctness, and as we shall see in the next section, timeliness of CPS and CPSoS are threatened by components of these systems failing due to accidental and intentionally malicious causes. A system's resilience to such faults is characterized by its ability to tolerate the occurrence of such faults and to return the system to a state at least as safe and secure as initially. In this project we consider a whole spectrum of faults.

On the one end of the fault spectrum are accidental faults, which follow known statistics and are typically of a transient nature. That is, although transient accidental faults may manifest as errors (e.g., by flipping bits in critical state) their rate of occurrance is typically known and can in general be compensated through encoding techniques, such as error correcting codes (ECC), encoded processing, lockstep execution and similar techniques. Moreover, their transient nature implies that once a fault it corrected in a given place (e.g., by overwriting the flipped bit and the ECC that contains it), its re-occurrence follows the same statistical pattern as initially and as for all other locations. In particular, it is unlikely that the same fault keeps re-occurring over extended periods of time, which, as we shall see next has consequences on how accidental faults affect the timeliness of CPS and systems of them.

On the other end of the spectrum are targeted attacks by highly skilled and well equipped adversaries, attacking the system for the purpose of causing faults intentionally and when and where the effect of experiencing such a fault is most severe. Adversaries sometimes follow rational incentives, such as maximizing their own profit, but in general such assumptions cannot be sustained with high confidence. Adversaries whose primary objective is to disrupt service or, in case of CPS, cause damage to the physical world, will not necessarily follow accidental fault statistics, reveal their presence, or stop in their attack after the first partial success. Instead we have to assume faults to persist, in a concealed manner, and adversaries to exploit the knowledge they have gained from previous attempts to speed up and strengthen their attack. In particular, we have to assume that adversaries will strategically select other replicas to, over time, gain knowledge how to compromise enough of them to gain control over the majority, which jeopardizes naive fault tolerance mechanisms and demands for resilience.

## 1.2 Timeliness and Synchrony

Before discussing the principled resilience techniques, summarized in Table 1 and their adaptation possibilities, let us revisit the standard models for characterizing the time-domain of the real-time systems that are at the heart of CPS and CPSoS. In particular, we shall discuss the threats that accidental and malicious faults impose on this domain.

Real-time and embedded systems (RTES) typically operate under system assumptions that the dependability community [28] summarize under the *synchronous system model*. That is, RTES are typically assumed to operate in environments where communication and computation are sufficiently predictable to derive upper bounds for computation and communication, called worst-case execution (WCET) and worst-case transmission times (WCTT). Under such assumptions, it is possible to communicate sufficiently often to synchronize clocks to for example form the sparse time base of clock-driven systems or to ensure that at least locally, events occur at the same time and in the same relative order.

Accidental and malicious faults threaten the coverage of the above timeliness assumptions, by faults causing omissions in message delivery and execution to crash, to behave in an arbitrary (Byzantine) manner.

To compensate, the dependability body of knowledge has focused on relaxed synchrony models, such partial synchrony [11], or asynchronous systems where the above bounds hold only during frequently recurring and sufficiently long "good" periods, during which the protocol will make progress, respectively in which such bounds cannot be known to hold. Partially synchronous and asynchronous systems cannot guarantee timeliness and are therefore generally ill suited for real-time systems unless they are equipped with a fail safe synchronous core and unless failure in the safe state into which such a core maneuvers the system are tolerable. In Section 5, we shall return to this aspect in our discussion of complex vs. simplex controllers and in the role they play in a cascaded control setting.

Attacks to cyber-physical systems have already been demonstrated, for example in the Aurora [9] project, and they have already been exploited in the wild (e.g., with Stuxnet [15] and the cyber attack against the Ukrainian power grid [20]).

We shall later return to two important aspect that distinguishes accidental from intentionally malicious faults: due to the stochastic nature of the former, it becomes increasingly more unlikely to see the same fault re-occurring during subsequent execution sequences of increasing lengths, in particular if faults are transient and stochastic independent. The same is not true for intentionally malicious faults triggered by an adversary. Conversely, one cannot exclude the possibility of a component failing again within a given time-span and because of accidental causes, even if the mean-time-to-failure (MTTF) is larger than this span. Conversely, provided we construct the system in the right way, adversaries would need a certain time to again mount an attack until he/she again managed to compromise a component.

Let us exemplify what we just wrote again on the example of a bitflip. Despite a relatively high MTTF for radiation induced bitflips hitting the same word in memory, it remains possible with some probability. Whereas the same bitflip happening for example in consequence of a rowhammer attack requires mounting this attack again, which takes a certain time, in particular if to mount

it, the adversary must first compromise another component from which to mount this attack.

## 1.3 Classical Resilience and Architectural Hybridization

The classical design patterns, in particular to cope with transient, accidental faults, include:

1. monitoring components, for the purpose of detecting faults and intrusions,
2. isolating them, for the purpose of confining the propagation of faults, and
3. repairing failed components, by restarting and re-executing them, to return the system to a state at least as safe and secure as initially.

During the time between detection and restart, the component is down and its functionality is only available again after restart, which can take several seconds. Further, fault and intrusion detection are typically incomplete in that not all faults are detected and that the system has to cope with false alarms.

For systems that cannot tolerate downtimes of the sort described above or where re-occurrence of the fault during re-execution cannot be excluded with high confidence, replication suggests itself to mask occuring faults. Replication can be in the form of triple-modular redundancy (TMR) or Byzantine-fault tolerant state-machine replication (BFT-SMR). The fundamental difference between these two approaches is that TMR assumes a trusted invocation structure (i.e., all replicas must receive the same inputs, at least approximately, at the same relative point in their execution and are expected to produce (approximately) identical results). BFT-SMR protocols on the other hand are equipped to deal with situations where replicas lie inconsistently to others (e.g., a leading replica proposing operation A to one subset, while proposing B to another subset of replicas). This inconsistent lying is called *equivocation*.

In synchronous, time-triggered systems, TMR typically applies time-based omission detection (for example by expecting a result within a bounded amount of time, while considering late results as originating from faulty replicas). Although suitable for accidental faults, timed protocols become brittle in the presence of intentionally malicious faults, as has been demonstrated [10]. In particular, they are likely to fail under time-domain attacks that are capable of affecting multiple replicas simultaneously. In Krueger et al. [16, 17], we have developed a mitigation strategy for such time-domain attacks, by randomizing the schedule to ensure replicas are not co-scheduled and not scheduled deterministically after the same accomplice task. As such, time-domain attacks impact severely only a subset of replicas, which can be compensated with additional correct replicas operating in a timely manner.

Time-free failure detection [19], or, like in the seminal protocol PBFT [6], the use of local timers merely as a mean to eventually react to faults, eliminate the above vulnerability to time-domain attacks, however at the costs of guaranteeing progress only during the “good” times of the underlying partial synchrony model.

| Resilience Mechanism | Fault Model trans. | pers. | crash | accid. | mal. | Masking immediate | Comments |
|---|---|---|---|---|---|---|---|
| detection | x | x | x | (x) | (x) | - | ability to detect |
| re-execution | x | - | x | x | -* | - | same attack |
| re-location | x | x | x | x | -* | - | same attack |
| diversification + re-execution | x | - | x | x | x | - - | |
| + re-location | x | x | x | x | x | - | |
| replication (same) | x | x | x | x | -* | x | same attack |
| replication (n-ver.) | x | x | x | x | -* | x | exhaust |
| replication (dyn. div) | x | x | x | x | x | x | |

**Table 1: Overview of basic resilience mechanisms and their effectiveness in the presence of different faults (transient vs. persistent and crash, accidental, intentionally malicious faults). Cells are marked as supported x, partially supported (x), and not supported -. Comments abbreviate the reasons explained below. Obviously detection only lets the system know of an error and requires additional means to act on it (e.g., by falling back to a fail-safe state or by invoking one of the other mechanisms). Immediate masking indicates whether the mechanism is able to conceal the presence of the fault or whether the system can return to correct behavior only after handling the fault. Without dynamic diversification, adversaries oppose the same set of replicas. Sooner or later, they will exhaust the healthy majority.**

### 1.3.1 Architectural Hybridization:

Architectural hybridization [26] includes trusted-trustworthy components that follow a distinguished system and fault model (e.g., they may operate or even communicate synchronously and fail exclusively crashing or not at all). The purpose of these components is to provide a limited functionality that is however essential to make the system more resilient or that avoids complicated and costly situations in homogeneous protocols. The arguments that help justify distinguishing the fault models of trusted components versus the rest of the system are rooted in the isolation of these components (limiting access to them through well-defined interfaces) and by the simplicity of the functionality they have to provide. For example, USIG, the trusted component of MinBFT [27] and similarly CASH, the trusted component of CheapBFT [14], merely offer signatures over trusted monotonic counters, which suffices for the BFT-protocol to avoid equivocation by disallowing different messages to be sent with the same counter value.

The above mentioned synchronous local subsystem that is able to enter a fail safe state could be an instance of such a subsystem, but more generally the research question linked to this construction principle is: "What is the minimal functionality that needs to be trusted to achieve the system's desired safety and security properties?". We shall return to this aspect in Section 5 where we introduce our approach to securing control at the low-level of the CPS architecture.

Table 1 gives an overview of the principled techniques that can be applied to handle faults of different kind. As can be seen, simple re-execution or re-location (e.g, after detecting a fault, which in general remains imperfect and/or requires domain specific knowledge, e.g., about plausible sensor values) has limited effectiveness on intentionally malicious tasks, unless combined with diversification strategies to present adversaries a moving target. The same is true for replication schemes, where replicas are identical. Even static n-version programming fails over time, as adversaries exhaust the healthy majority. Immediate masking, that is the ability of a system to conceal fault handling and configuration delays requires replication to hide the behavior of faulty nodes behind a healthy majority operating in consensus. We shall see in Section 5 how leveraging ADMORPH results on inherent control stability of plants allow us to weaken immediate masking and tolerate faults through re-execution after rejuvenating replicas fast enough.

# 2 Adaptive Resilience

Most of the state-of-the-art on adaptive resilience mechanisms [5, 22, 7, 2], in particular in terms of Byzantine fault and intrusion tolerance techniques, suffer from the above described divergence of the dependability and real-time/embedded systems communities, which will require some re-consolidation effort to obtain time-free, partially synchronous resilience mechanisms (capable of withstanding time domain attacks) that exhibit predictability and timeliness, ideally as an emerging property.

In the first class of time-agnostic, but not real-time capable protocols, WHEAT [24] decreases client latency by selecting as leader the best connected replica. Adapt [3] builds on Abstract and Aliph [2] to perform AI-driven dynamic adjustments of the payload protocol. AWARE [4] improves over WHEAT by continuously monitoring connectivity and re-configures based on these observations. We are currently investigating the effects of manipulating one's transmission time on the reconfiguration of AWARE. Also, unlike the above protocols, which primarily focus on optimizing the system, we have investigated means to react to situations where the perceived threat level increases [23]. In such situations, the BFT-SMR protocol must activate additional replicas to compensate for the stronger adversary, which normally requires reaching consensus about the configuration of the system [5, 22, 2]. Building upon the group membership impossibility result by Chandra et al. [8], Simoes-Silva et al. [23] were able to show that this is impossible in a partially synchronous system and suggest reaching agreement proactively on how the system should react in case threats increase.

The above works unanimously assume the strongest fault model, namely malicious threats. Of course several alternatives become possible when threats are of a more benign nature, such as crash and omission faults or only transient faults, or when time-domain attacks are excluded.

For example, assuming crash and omission faults, Rodrigues et al. [25] and Wang et al. [31] investigate coordination of replicated execution in event triggered systems, an aspect that will not be necessary in time-triggered systems [1] as long as time-domain attacks are excluded. Unlike event-driven systems, which schedule tasks in consequence of their triggering event (e.g., the arrival of a message from a remote sensor), time-triggered systems require such communication to happen

at a specific time slot, prior to the slot where replicas computing their response, after which they send their replies to remote actuators in subsequent slots. Guarding communication thereby prevents replicas from acting outside their allocated slots.

Miedema et al. investigate single-event upsets and how adjusting DAGs with alternative paths improve tolerating such faults. See Deliverable D1.2 for further details.

# 3 Adaptation Opportunities and Methods

In the following, we will report on several adaptation opportunities and methods, which we already identified and on the additional research we plan to conduct. As illustrated in Deliverable D1.1, adaptation will be the key to compensate accidental faults and to fend off adversaries mounting a targeted attack to the system. Adaptation will act along the following four lines:

1. to replace and relocate components along the attack pathway, creating a moving target defense to buy the time that is necessary to repair and recover compromised components and in turn strip adversaries from their foothold in the system;

2. to adjust the internal resilience of components, by replacing configurations with more resource demanding configurations that exhibit a better resilience to match the perceived threat level;

3. to adjust the resource mapping of a component in case some resources become unavailable;

4. to adjust the functionality of the system to guarantee a degraded service in case not enough resources remain to sustain the full functionality of the system; and

5. to optimize the system whenever the perceived threat level drops.

It is important to realize that the first three happen while the system has detected the presence of an imminent threat, whereas the latter applies in those situations where the system has gained confidence that it is no longer exposed to the high risks it has prepared for. Lines 2–4 equally apply to accidental faults and targeted attacks, whereas replacing and relocating components along the attack pathway (Line 1) is only necessary to fend off adversaries. Of course, relocation also happens as a consequence of resources becoming unavailable (e.g., due to permanent failure of a resource). In the presence of targeted attacks, adaptation along Lines 1–4 must outpace adversaries in adapting faster to improve system resilience before the adversary can exploit the current vulnerable state from which the system tries to evade.

## 3.1 Adapting the System along Attack Pathways

Components are characterized by their state and executable and will interact with other components through well defined communication interfaces (e.g., the inter-process communication mechanism (IPC) provided by the PikeOS microkernel). Relocating a component from a faulty core and redirecting IPC connections to or from this components evades faulty resources and

presents adversaries a fresh instance, in particular if a component is not replaced by an instance the adversary was already able to analyze, but instead by instances the adversaries has not seen, yet. Depending on whether or not the state has been compromised, it may be either transferred from the previous component or re-instantiated from scratch or from a checkpoint. Adaptation of a component generally entails adaptation of other elements of the system, most notably the schedule, which defines how applications are mapped and multiplexed to resources.

Dynamically changing the attack pathways in the manner described above also allows the system to evade from accidentally failing resources, because the new instance will be mapped to a different subset. However, such relocation must happen more generally, including for resources that are not on the pathway. Hence, we introduce Line 3. Remapping tasks to a different subset of resources requires adversaries to redo the work they have already spent to prepare the attack of the old instance.

Let us exemplify the latter for an attacked component with a network connection to the outside. Such a component typically interacts with a network stack and network interface card (NIC) driver, which implement the communication protocol (e.g., TCP/IP) and network hardware interaction, respectively. Naturally, being exposed to the environment, the NIC driver and network stack are typical candidates for a first compromise of the system. However, aside from ongoing transmissions and open connections, the internal state of the network stack and NIC driver is largely disconnected from the state of the receiving component. It is therefore possible to periodically rejuvenate the stack into a state frozen after initialization and to repeat NIC initialization to remove adversaries that have entered this most external layer of the system. Rejuvenating and replacing the networked component, requires adversaries to repeat the work they have already performed for compromising this network layer before they can identify and exploit vulnerabilities in the receiving component. The latter is of course provided we can present the adversary with an instance he was not able to analyze.

## 3.2 Adapting Internal Resilience

Many components or conglomerate of components already exhibit some form of internal resilience to accidental and malicious faults. For example, triple modular redundant systems, or more generally replicated systems implementing Byzantine agreement operate $n$ replicas out of which $f$ replicas may become compromised before the adversary gains control over the conglomerate of these replicas. $n$ and $f$ are typically related (e.g., $n \geq 3f + 1$ for homogeneous, partially synchronous consensus [6]). Thus, increasing $n$ also increases $f$ and the internal resilience of this conglomerate.

By homogeneous consensus protocol, we refer to a protocol running in a homogeneous system. In such a system, all components follow the same fault model. In contrast architectural hybridization [10, 26] allows identifying trusted-trustworthy components that follow a distinguished fault model (e.g., they fail only by crashing, while the remaining system may exhibit arbitrary, possibly Byzantine faults).

In the most general case, changing the membership of which components belong to a group of replicas boils down to a consensus problem, requiring agreement for addition and removal.

It therefore depends on the system model whether reaching such an agreement faster than the adversary compromises more than the initial $f$ replicas is still possible. We are currently exploring possibilities to proactively prepare for such an adjustment to later be able to react without first having to reach consensus how this reaction should look like. In particular, we investigate how the inherent resilience of a plant to missed, wrong or held actuation helps performing these adaptations more efficiently.

For example, the state-of-the-art body of knowledge requires $2k$ additional replicas to operate safely and securely through attacks while up to $k$ replicas are repaired and returned to a state at least as secure as initially. The ability to tolerate some deadline misses in the case of continuous disagreement among smaller quorums allows us to delay the activation of these replicas or to avoid them in the first place.

### 3.3 Adapting Functionality

Deliverable D5.1 already identifies how the ADMORPH use-case scenarios can adapt their functional and non-functional properties to adjust to those situations where not enough resources remain to sustain full system functionality. We draw functionally reduced components from an initial pool of deployed alternatives, but consider also means to supply such a pool dynamically.

### 3.4 Adapting to Lowering Threat Levels

As already mentioned, the fundamental difference between the above three and this fourth adaptation opportunity is the time until which the adaptation must have happened in relation to the time the adversary needs to break into the system. When optimizing, the system is already confident about the absence of faults and attacks at the system's current threat level. Therefore, there is no bound until which the adaptation must succeed other than the desire to quickly return to a more efficient modus operandi and to do so in steps that will not jeopardize the timing guarantees the system has to provide. In particular, optimization can always be aborted to react to increasing threat levels.

## 4 An Abstract View on Control

Before showing more concretely how the above strategies map to the resilient control framework, developed in WP.2, let us abstract from the concrete usage scenarios that drive our developments (see also Deliverable D.5.1) to then return in D.2.2 and see how the individual resilience mechanisms can interplay in an adaptive manner.

Figure 1 illustrates common building blocks of a CPS. Control typically happens at various levels with the lowest level directly interfacing with the plant through actuators, while observing the state of the plant through sensors. At this level, any incorrect control signal sent to the actuators risk jeopardizing the stability of the plant and possibly safety of the system. Common resilience patterns therefore include interposing actuation signals with trusted circuits to ensure k-out-of-m

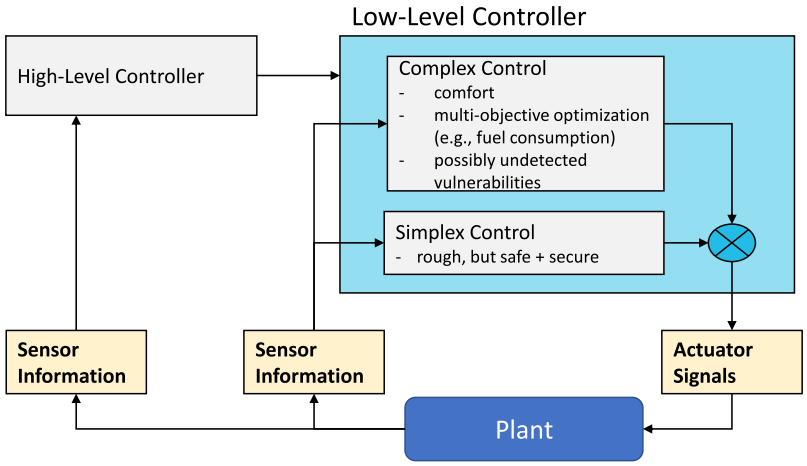


**Figure 1: Abstract view on a generic control architecture. Shown is a cascaded high-level controller that interacts with a complex/simplex pair to operate a plant.**

monitors agree with the control signal, respectively to perform a majority among replicated control tasks. The complexity of the latter thereby impacts how trustworthy the whole low-level control loop is.

## 4.1 Cascade Control Systems

Cascade control is a pattern to extend the functionality of a lower-level controller by controlling the desired value to which the low-level controller should settle. A prominent example of such a high-level controller is trajectory generation, leveraging additional sensors such as Lidar and cameras to obtain information about a vehicles environment in order to find a safe trajectory for crossing this environment. The high-level controller then passes low-level control the desired location on the trajectory, which low-level control follows by adjusting the steering signals to remain on track. Often the complexity of high-level controllers and hence their vulnerability to accidental and malicious faults exceeds the complexity of the low-level controller by far. In the cascaded configuration, we therefore prevent the high-level controller from interacting with the plants directly, which allows us to intercept and validate desired values and to shield the low-level controller from operating on incorrect inputs. Also the control loop frequency for high-level controllers tend to be much higher (e.g., in the second range for trajectory generation), which allows applying more costly validation on the generated values.

Since the low-level controller can not be protected as described above, we apply the simplex/-complex split discussed next and replicate the simplex part for further safety and security against accidental and malicious faults.

### 4.2 Complex / Simplex Controller Pairs

To further reduce the complexity of critical control tasks, controllers can be split into simple control loops, which focus exclusively on maintaining safety of the plant, but possibly at the costs of high energy consumption or while operating the system in an uncomfortable way. A prominent example is a simplex autopilot, tasked to perform aggressive, fuel demanding maneuvers in case the complex autopilot fails to return the system into a safe state [18] (and similar for aerial vehicles [29]). The complex part of such a simplex/complex controller pair aims at avoiding such undesired behavior, trading off controller simplicity and hence accepting a higher susceptibility to accidental and malicious faults.

The simplex/complex controller pair forms a control loop with the plant where in case safety is at risk the simplex controller takes over control from the complex controller to return the plant to safety. This switch is typically triggered by a monitor detecting a flaw in the complex controller's state or detecting sensor value that leave the margin where the complex controller may operate without intervention of the simplex.

## 5 Resilient Control

Individualistic resilience of each CPS will be essential for obtaining CPSoS-wide resilience. This is because each individual CPS acts in the physical world and must therefore be safe irrespective of the swarm. Unhandled accidental faults risk this safety and compromise subverts the CPS into a threat to other CPS or worse, the humans that operate in their proximity.

Attacks of the above kind have already been shown. For example, by exploiting a vulnerability in the radio telemetry subsystem of their Jeep Cherokee [12], researchers Charlie Miller and Chris Valasek had full remote control over their vehicle and from there its only a matter of imagination and good or bad intend whether such a compromised vehicle is turned into a cyberkinetic weapon against the platoon of cooperatively driving cars.

Our primary goal is therefore to adapt CPS to make each individual strong enough to offer a minimal residual safety and security, and to ultimately recover, possibly with the help of healthy CPS in the individual's proximity.

What makes us confident that concentrating on the CPS will extend smoothly to CPSoS is that we can already identify self-similarities in the techniques and principles applied within the CPS. For example, much like a fernleaf repeats its own pattern, cascaded complex control (e.g., learning-based, complex components as they are required for autonomous driving, or in our case taxiing on the airfield) may well be tolerated to fail, or be protected by more lightweight mechanisms if they can rely on the abstraction of a resilient controller, capable of preventing both time- and value-domain faults from manifesting at the level of the actuator. The same general pattern recurs when coordinating the actions of a multitude of CPS in a CPSoS. If each CPS individually exhibits the notion of an in principle resilient system (possibly with degrading QoS in extreme situations), CPSoS wide cascaded controls may leverage this internal resilience and the fact that individuals no longer fail in the most pessimistic manner, but perform a detectable, coordinated and fail operational reduction of QoS into fail safe states.

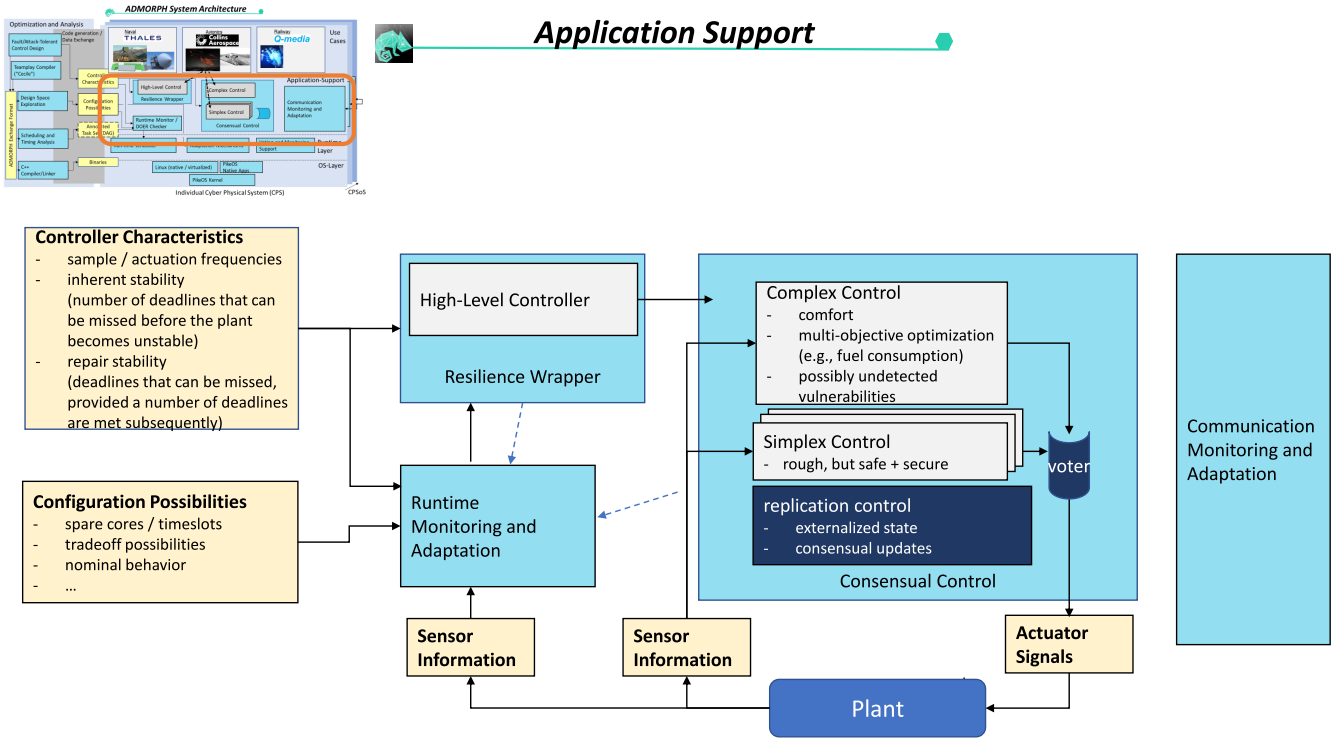


**Figure 2: Resilient control, monitoring and adaptation framework and its embedding into the Admorph architecture (top left). As can be seen, the cascaded and simplex/-complex control pair pattern from Figure 1 embeds smoothly into our framework and is supported by resilience wrappers for the high-level controller and a replication control unit for the simplex controller, which we replicate with the help of a trusted-trustworthy voter. Runtime monitoring and adaptation observes these units (dashed lines) and reconfigures the system as needed.**

In the following, we introduce our monitoring framework and some of the resilience patterns we have developed to obtain resilient control. Figure 2 illustrates this framework.

## 5.1 Accidental Fault and Attack Tolerant Control Design

Our resilient control framework embeds into the Admorph architecture at the application support level and provides resilience mechanisms, monitoring and adaptation for resilient control, both at high-level and at low-level for controllers with direct access to plants.

The Run-Time Monitoring and Adaptation Agent draws configuration possibilities from design space evaluation and triggers adaptations once critical situations are detected in one of the controllers or in the controlled plant. For that, the monitoring component of the agent draws information from the plant's sensors but also from additional sensors in the resilience wrappers and in the voter to see when adaptation is necessary. The adapration part of the agent then pre-

pares a new configuration, while leveraging the old configuration to still mask faults until the new configuration is ready. Once this is the case, the agent atomically transitions control to the new configuration using a special interface at the voters. This way, downtimes and cascading effects are avoided because the new configuration can be prepared while the old one is still operational and functional, although it must be expected that soon this resilience will be exhausted. Hence the adaptation to switch to a new configuration before this happens.

The developed resilience mechanisms leverage the advanced controller characteristics obtained from the ADMORPH fault tolerant control design tools described in Maggio et al. [21] and Vreman et al. [30]. The former shows conditions under which a plant can tolerate its controller to miss a certain number of deadlines, before stability is at risk. The latter extends this to a holistic stability analysis and reveals an even larger number of tolerable deadline misses, provided no deadline is missed in several subsequent executions of the control task.

We leverage the former in our replication control component to operate critical control tasks (e.g., the simplex controller) in a replicated manner, but with a number of replicas that are only sufficient to detect faults, but not mask them. Replication control regains masking capabilities through a re-execution strategy, which we explain below.

Should the monitor detect frequent errors during such a re-execution, it may leverage the result from the second paper [30] and trigger a reconfiguration of the system. The plant will remain stable during the time required for this reconfiguration but must then experience a phase where subsequent deadlines are hit by the control task. In the presence of accidental faults and targeted attacks, this is only possible if the controller is then changed to mask faults immediately. As we have shown in Table 1, this asks for active replication with sufficiently many replicas to proactively rejuvenate and diversify replicas to prevent adversaries from exhausting healthy majorities.

AI-assisted monitoring and adjustment of the network ensures that communication with other CPS in the CPSoS remains stable. In the worst case, fail safes are identified and entered should communication remain disrupted.

## 5.2 Cascaded Control and Safety Kernels

Cascaded control gives rise to a design principle originally developed in the context of split applications [13, 32]: identify and split control into a large high-level and small low-level controller, such that the small and therefore more trustworthy controller ensures safety of the system, while the large high-level controller offers additional functionality. Returning to the above trajectory following example, this split is not yet perfect, unless the low-level controller is able to ensure absense of collision while following this route. However, then, no matter what trajectory the high-level controller suggests, the cascaded low-level controller will verify and follow the trajectory only as long as it is safe. One further requirement for such a design is that the low-level controller will always be able to enter a fail safe state should it be asked to follow a faulty trajectory. Stopping on the runway is no such state and therefore requires other mechanisms, which we explore in Task 4.5 of WP.4. Rather than relying solely on local components, low-level control could also receive guidance from peer CPS in the CPSoS and may therefore benefit from remote nodes taking over the role of the high-level controller, adapting until the faulty component can be replaced.

From the above, the reader may assume that the complex/simplex pair forming the low-level controller necessarily has to implement all additionally safety checks. This is possible, but not required if a safety kernel in the monitor assumes this role. Safety kernels offer application-specific validation of monitored outputs and may serve as an indicator to adapt the high-level controller locally or to switch to remote takeover.

## 5.3 Leveraging Complex/Simplex Pairs

Leveraging complex/simplex pairs is a further means to split functionality into a necessarily trusted (simplex) and tolerably untrusted part (complex). While complex remains within safe bounds, simplex or a corresponding safety kernel in the monitor observe complex to identify when complex causes the plant to leave a region where simplex must take over to not risk safety. The monitor then adapts the system to divert control to simplex. However, in such a configuration, a single-instance simplex controller, as depicted in Figure 1, forms a single-point of failure, while being large enough to not justify trusting this controller to not fail. For this reason, we replicate the simplex controller to mask faults as they occur. Further reasons to replicate the simplex controller stem from high safety integrity levels, where failure rates of less than $10^{-7}$ accidentally caused failures per hour must be met on hardware that is only qualified for $10^{-5} - 10^{-6}$. Replication can be deployed as a standard configuration with $n \geq 2f + 2k + 1$ replicas that actuate the plant after reaching consensus and that are rejuvenated and diversified to tolerate accidental faults while presenting adversaries a moving target. However, as discussed above, it will also be possible to leverage the inherent stability of the controller to achieve masking with a number of replicas $n$ that are only sufficient to detect faults (i.e., $n \geq f + 1$).

## 5.4 Replication Control

Operating replicas with a quorum that only allows for detecting faults, requires coordinating replication and equipping the replicated controllers with additional functionality to enable extremely fast restart and recovery. The basic idea is to let the detection quorum contribute actuation proposals for the same sensor input until enough proposals could be collected to reach agreement. In case no error is detected, this is the case immediately after the $f + 1$ replicas propose a request. The voter will conolidate these proposals and actuate the plant. In case of error, a total of $f + 1$ matching proposals must be collected of which in each round only one is guaranteed to be correct (due to our fault assumption of up to $f$ faulty and one healthy replicas). Our approach is to rejuvenate replicas after each round so that all but persistent faults are removed, but instead of the rejuvenated replicas acting as themselves, we rotate their identities and provide them with the same sensor signals as in the previous round in case an error was detected in that round. This continues until enough matching proposals arrive to reach agreement. Obviously, one must in addition prevent agreement on faulty actuation outputs. For accidental faults, we equip the voter with a means to decide on a round based encoding of the output, such that stuck proposals will match only in the round in which they got stuck. For the same reason, we cannot tolerate compromise of more than $f$ replicas faster than the time we require to reach consensus, which can

span up to $f + 1$ rounds. Further implementation details, in particular on the voter to support such replication control, follow in Deliverable D.2.2.

Let us here only briefly mention the mechanism that is required to allow for an extremely fast restart of replicas every round. Inherent in most control tasks is a structure that can be briefly summarized in the following pseudocode:

```
1  while (true) {
2    read sensors
3    compute control law
4    actuate plant
5    wait for next round
6  }
```

In particular, Line 3 obtains most of the information from the sensors captured in Line 2 and only relies on rather small amounts of internal state. For example, PID controllers merely maintain the previous actuation signal to compute an error for the derivative portion and an accumulator to compute the integral part of PID. More complex controllers, such as adaptive model predictive control, may update the control matrix, but again this matrix is relatively small in size and updates are rare compared to the cells that are read. For this reason, we leverage voting and the tight coupling of replicas for a second operation, namely to keep these elements in consensually updated memory. In addition, we introduce support for providing sensor values from previous instances, by capturing sensor values in a read-only mapped ring buffer, such that replicas may revert back to the values of previous rounds if an error was detected. All remaining state (like stack, heap, etc.) that does not comprise read-only data, can be reset at the beginning of the above while loop. Allowing for an extremely fast rejuvenation, by just resetting the controller and alternating between occasionally replaced read-only mapped code images to create a moving target defense. The resulting pseudocode becomes:

```
1  control(id, round):
2    if (voter.mismatch) {
3      ctrl_round = voter.round
4    } else {
5      ctrl_round = round
6    }
7    read sensor from ringbuffer at ctrl_round
8    compute control law
9    propose // as replica (id + (f+1) * (round mod (f+1)))
10     the actuation output
11     all updates to controller internal state (e.g., acc, prev)
12   wait for next round
```

Here, id is the identifier of the replica executing thread and round is the current round. The thread starts in continuation style with empty stack and heap in the function control and is returned there by the OS every round. Observing the voter, it can now identify whether the previous vote already succeeded. If not, it will read the sensors from the ringbuffer of the round where replicas failed to reach agreement. Being equipped with these values and assuming a deterministic control law, the thread computes the actuator output and all control state updates in Line 8 and then acts in a rotating manner as the replica id, id + f+1, id + 2(f+1), etc., which is enforced in the

voter. Proposals are made both on the actuator output and on updating the controller internal state (e.g., the error and accumulator in case of PID), which leads to the actuation of the voter and possibly consensus and a return of the voter to a matching state.

Internally, the voter keeps all proposals of this thread and its peers from previous rounds until a matching quorum of f+1 identical proposals could be found, in which case it actuates the plant and updates the consensually updated and otherwise read-only shared memory in which all replicas keep controller internal state. ECC ensures that this memory remains intact despite accidental faults.

# 6 Conclusions

In this extended version of the deliverable D.2.1, denominated as D.2.1b, we have extended our analysis of adaptation possibilities to tolerate accidental faults and targeted attacks. We highlight in particular the adaptation possibilities of the application support layer that leverages adaptation to secure control from faults and attacks and that ultimately leads to our framework for monitoring and controlling plants in a resilient manner.

# 7 References

[1] *The Time-Triggered Architecture*, pages 285–297. Springer US, Boston, MA, 1997.

[2] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. *ACM Trans. Comput. Syst.*, 32(4), 2015.

[3] Jean-Paul Bahsoun, Rachid Guerraoui, and Ali Shoker. Making BFT protocols really adaptive. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 904–913. IEEE, 2015.

[4] Christian Berger, Hans P Reiser, João Sousa, and Alysson Bessani. Resilient wide-area byzantine consensus using adaptive weighted replication. 2019.

[5] Carlos Carvalho, Daniel Porto, Luís Rodrigues, Manuel Bravo, and Alysson Bessani. Dynamic adaptation of byzantine consensus protocols. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 411–418, 2018.

[6] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186, USA, 1999. USENIX Association.

[7] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002.

[8] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, page 322–330, New York, NY, USA, 1996. Association for Computing Machinery.

[9] CNN. Xmouse click could plunge city into darkness, experts say", April 2018.

[10] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, pages 174–183. IEEE, 2004.

[11] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[12] Andy Greenberg. Hackers remotely kill a jeep on the highway—with me in it, July 2015.

[13] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. Reducing tcb size by using untrusted components: Small kernels versus virtual-machine monitors. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, EW 11, page 22–es, New York, NY, USA, 2004. Association for Computing Machinery.

[14] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 295–308, 2012.

[15] Gregg Keizer. Is stuxnet the 'best' malware ever?, Sept. 2010.

[16] K. Krüger, M. Völp, and G. Fohler. Vulnerability analysis and mitigation of directed timing inference based attacks on time-triggered systems. In *ECRTS*, 2018.

[17] Kristin Krüger, Nils Vreman, Richard Pates, Martina Maggio, Marcus Völp, and Gerhard Fohler. Randomization as mitigation of directed timing inference based attacks on time-triggered real-time systems with task replication. *Leibniz Transactions on Embedded Systems*, 7(1):01:1–01:29, Aug. 2021.

[18] Seong Kyung Kwon, Ji Hwan Seo, J. Lee, and K. Kim. An approach for reliable end-to-end autonomous driving based on the simplex architecture. *2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pages 1851–1856, 2018.

[19] Gérard Le Lann and Ulrich Schmid. How to implement a time-free perfect failure detector in partially synchronous systems. Technical Report Research Report 28/2005, Technische Universität Wien, 2005.

[20] Robert M. Lee, Michael J. Assante, and Tim Conway. Analysis of the cyber attack on the ukrainian power grid, March 2016.

[21] Martina Maggio, Arne Hamann, Eckart Mayer-John, and Dirk Ziegenbein. Control-System Stability Under Consecutive Deadline Misses Constraints. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:24, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[22] Michael K Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, 1996.

[23] Douglas Simoes Silva, Rafal Graczyk, Jeremie Decouchant, Marcus Völp, and Paulo Esteves-Verissimo. Threat adaptive byzantine fault tolerant state-machine replication. In *40th International Symposium on Reliable Distributed Systems (SRDS)*, Sept 2021.

[24] Joao Sousa and Alysson Bessani. Separating the WHEAT from the chaff: An empirical design for geo-replicated state machines. In *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, sep 2015.

[25] Paulo Verissimo, Luis Rodrigues, and Antonio Casimiro. Priority-based totally ordered multicast. Technical report, 1995.

[26] Paulo E. Veríssimo. Travelling through wormholes: A new look at distributed systems models. *SIGACT News*, 37(1):66–81, March 2006.

[27] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2011.

[28] Paulo Veríssimo and Luís Rodrigues. *Distributed Systems for System Architects*. Springer, 2001.

[29] Prasanth Vivekanandan, Gonzalo Garcia, Heechul Yun, and Shawn Keshmiri. A simplex architecture for intelligent and safe unmanned aerial vehicles. In *RTCSA*, 2016.

[30] Nils Vreman, Anton Cervin, and Martina Maggio. Stability and Performance Analysis of Control Systems Subject to Bursts of Deadline Misses. In Björn B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:23, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[31] Yun Wang, E. Anceaume, F. Brasileiro, F. Greve, and M. Hurfin. Solving the group priority inversion problem in a timed asynchronous system. *IEEE Transactions on Computers*, 51(8):900–915, 2002.

[32] Carsten Weinhold and Hermann Härtig. Vpfs: Building a virtual private file system with a small trusted computing base. *SIGOPS Oper. Syst. Rev.*, 42(4):81–93, April 2008.