

RESEARCH

Open Access



Adaptive middleware in go - a software architecture-based approach

Nelson Rosa* , David Cavalcanti, Gláucia Campos and André Silva

*Correspondence: nsr@cin.ufpe.br
Universidade Federal de
Pernambuco, Recife, Pernambuco,
Brazil

Abstract

Adaptive middleware is essential for developing distributed systems in several applications domains. The design and implementation of this kind of middleware, however, it is still a challenge due to general adaptation issues, such as *When* to adapt? *Where* to include the adaptation code? *What* to adapt?, and *How* to guarantee safe adaptations?. Current solutions commonly face these challenges at the implementation level and do not focus on the safety aspects of the adaptation. This paper proposes a holistic solution implemented in Go programming language for developing adaptive middleware centred on the adoption of software architecture principles combined with lightweight use of formalisms. Software architecture concepts work as an enabling approach for structuring and adapting the middleware. Meanwhile, the formalisation helps in providing some guarantees before and during the middleware execution. The proposed solution is evaluated by implementing an adaptive middleware and comparing its performance against existing middleware systems. As shown in the experimental evaluation, the proposed solution enables us to design and implement safe adaptive middleware systems without compromising their performance.

Keywords: Adaptive middleware, Software architecture, Go, Lightweight formalisation

1 Introduction

Design issues commonly considered in implementing adaptive software systems also impact the development of an adaptive middleware [30]: *Why* to adapt the software? *When* to adapt? *Where* is the need for change? *What* artefacts need to be modified? *How* is the adaptation performed? How to guarantee safe adaptations?. As observed in [29], further concerns have their origin in the middleware domain itself. Firstly, changes in the application's requirements (above the middleware) and infrastructure conditions (below the middleware) usually motivate the adaptation. Secondly, the appropriate time to trigger the middleware adaptation should take into account the application's state. Finally, the middleware adaptation is also more critical in the sense that changes can affect the middleware itself and the application built atop it.

Initiatives on designing and implementing adaptive middleware are not novel [4, 6, 17]. Meanwhile, the appearance of new application domains, such as cloud computing [25], IoT [21, 31], sensor networks [23], e-Health [16], along with the emerging

of new technologies like process mining [29], improvements on model checkers performance, adoption of software architecture concepts [28], and the rise of new programming languages [26] have been responsible for renewing the challenges and possibilities of developing middleware systems.

In this context, this paper still focuses on the classical problem of how to design safer and performative adaptive middleware systems. To face this challenge, however, it combines software architecture principles, lightweight formalisation, and features of Go programming language [9, 20]. In the end, the objective is to provide a holistic solution including a *development* framework and an *execution* environment to facilitate the job of adaptive middleware developers. The use of formalisms allows advances on the development and execution of safer middleware systems, while Go helps to improve the middleware performance and design.

Having the mentioned objective in mind, this paper extends an existing framework named *MidArch* [26, 28, 29]. The new solution, namely *gMidArch*, is fully reimplemented in Go, and extends the previous version in five main points: works with a richer set of formal operators to formally specify architectural elements, presents a programming language-agnostic ADL (Architecture Description Language), proposes additional architectural elements to build MOM (Message-Oriented Middleware), redesigns the execution environment to work with software architecture principles, and includes a new adaptation strategy, namely evolutive adaptation.

The rise of modern languages like Go helps to boost the implementation of adaptive middleware. In practice, the potential of Go is a two-way street between the implementation and design. Being a programming language, Go enables us to advance on implementation issues while some facilities already provided by the language allow some improvements on the middleware design. For example, the concurrency model that includes goroutines (lightweight threads) and CSP-inspired [13] channels enormously facilitate the implementation of concurrency issues. Meanwhile, the existence of dynamic plugins simplifies the design of the adaptation mechanisms.

The evaluation of *gMidArch* focused on comparing the performance of an application built atop different middleware flavours: RPC-based/MOM middleware systems implemented using *gMidArch*, two commercial middleware systems (*gRPC*¹ and *RabbitMQ*²) and an RPC-based adaptive middleware (*AFirM*) [33]. The comparison with the commercial middleware systems is essential to show the viability of using *gMidArch* middleware systems even in scenarios in which adaptation is not mandatory. Finally, even not being commercial, the comparison with *AFirM* is interesting because it implements adaptation strategies similar to *gMidArch*.

The rest of this paper is organised into six sections. Section 2 introduces basic concepts of adaptive middleware and briefly describes *MidArch*. Next, Section 3 presents details of *gMidArch*. Section 4 shows a performance evaluation of a middleware built using *gMidArch*. Section 5 presents existing researches on adaptive middleware. Finally, Section 6 presents conclusions and some future work.

¹<https://grpc.io/>

²<https://www.rabbitmq.com/>

2 Background

Before describing the proposed solution for building adaptive middleware systems, we present some basic concepts of adaptive middleware and introduce *MidArch*.

2.1 Adaptive middleware

Adaptive middleware is a particular kind of middleware whose behaviour can be modified at runtime. The design of this kind of middleware faces some of the 5W1H issues associated with self-adaptive software [30]. The first issue, *Why to adapt*, is usually motivated by the need of adapting the middleware to changes in application's requirements, changes of environmental conditions, fixing middleware's bugs or extending/improving the middleware functionality.

Middleware designers also have to decide *when* the adaptation needs to be applied, and when it is possible to do so. It is also necessary to determine if adaptation actions need to be carried out to react to undesired behaviours or proactively act to avoid them. For example, the adaptation should occur when the network becomes overloaded, e.g., forcing the data compression (reactive). Alternatively, the middleware adaptation can happen when the adaptation mechanism detects that a performance bottleneck of a middleware component becomes imminent (proactive).

Another critical aspect of designing an adaptive middleware refers to establish where the need for change is. In this case, the middleware developer has to define if the adaptation should be applied to a single component, a middleware layer or the whole structure, for example. Very related to the *Where* issue, the *What* one demands the definition of which artefact or attributes must be changed when an adaptation is needed. For example, the change may be something simple as an algorithm replacement or something more complex like to add or replace a component of the middleware architecture.

Finally, it is essential to define *how* the adaptation actions can be executed and implemented. For example, as the adaptation logic usually consists of several components (e.g., MAPE-K) and may be very complex, it becomes necessary to define if the adaptation logic is internal or external to the middleware.

2.2 MidArch

MidArch (adaptive *M*iddleware aid by software *A*rchitecture) [28] is a solution to help middleware developers to implement and execute adaptive middleware in a safe way. It uses lightweight formalisation, includes a software architecture-based framework and an execution environment. The framework facilitates the *middleware development* and the lightweight adoption of formal methods (CSP) integrated into the framework enables us to verify behavioural properties before and while the middleware executes. At runtime, an environment manages the *middleware execution*, implements a MAPE-K (Monitor, Analyze, Plan, Execute, and Knowledge) feedback loop [14] and uses process mining tools [1] to decide the right time to adapt.

Using the middleware framework, developers need to define a software architecture in a Java-based Architecture Description Language (ADL) to implement the adaptive middleware. The framework consists of a set of components and connectors implemented in Java, annotated with CSP specifications and freely reused and composed in a software architecture according to the needs of the middleware developer. Having defined

Table 1 *gMidArch* extensions

Extension	Rationale
Richer subset of CSP	The support to a more abundant subset of CSP operators used to specify the behaviour of architectural elements formally. The new set of supported operators includes external choice, parallel, and exception operators, and enable us to describe much more complex behaviours.
Agnostic ADL	An ADL, namely <i>mADL</i> , keeps middleware developers away from programming language specificities. Using the proposed ADL, developers only need to define the middleware software architecture that becomes the unique artefact needed to implement the middleware.
Richer subset of components	The architectural framework of <i>MidArch</i> was extended with execution environment and MOM specific components. This extension allows the implementation of publish/subscribe middleware systems in addition to the original support to RPC-based ones.
Customised execution environment	Use of software architecture principles to design the execution environment that is customised according to the middleware being implemented. This architecturization of the execution environment is an initial step for making viable its dynamic adaptation.
Evolutive adaptation	Development and runtime support to evolutive adaptation. The evolutive adaptation is important because it

the architecture, it is then verified to check behavioural (using FDR³) and structural properties, deployed and executed by the execution environment. At runtime, middleware execution logs are continuously monitored to trigger adaptations. In this case, an adaptation plan is generated and executed to change the middleware architecture that is verified again and redeployed.

MidArch supports two kinds of adaptations, namely *corrective* [29] and *proactive* [26]. A corrective adaptation is triggered when a problem in the middleware execution is detected and needs to be fixed, e.g., an error to establish a connection. In turn, proactive adaptation tries to anticipate a performance problem and triggers the adaptation before the problem occurs. Currently, we use process mining techniques (PROM⁴) to trigger reactive adaptation and PRISM models [18]) for proactive adaptations.

3 Go *MidArch*

Go *MidArch* (*gMidArch*⁵) is a full implementation of *MidArch* in the Go programming language that includes the extensions presented in Table 1.

As a result of the proposed new extensions, Fig. 1 shows an overview of *gMidArch*. Developers utilise elements available in *Architectural Library* to define the middleware software architecture using the proposed agnostic ADL, namely *mADL*. This architectural description serves as input to *Creator* that automatically produces a customised software architecture of the execution environment (*EE Software Architecture*). *Generator* produces CSP specifications that describe the behaviour of the adaptive middleware (*CSP Mid*) and execution environment (*CSP EE*).

It is worth observing that *Generator* uses both the software architecture artefact and specifications stored in the *Architectural Library* to generate the CSP specification of the whole architecture. In practice, the software architecture artefact defines how the components are connected (see Section 3.2), e.g., “*c1,t,c2*” means that components *c1* and *c2* are connected through connector *t*. CSP specifications of *c1*, *t*, and *c2* are already stored

³<https://www.cs.ox.ac.uk/projects/fdr/>

⁴<http://www.promtools.org>

⁵Source code available at <https://github.com/gfads/midarch/tree/nelson/evaluation>

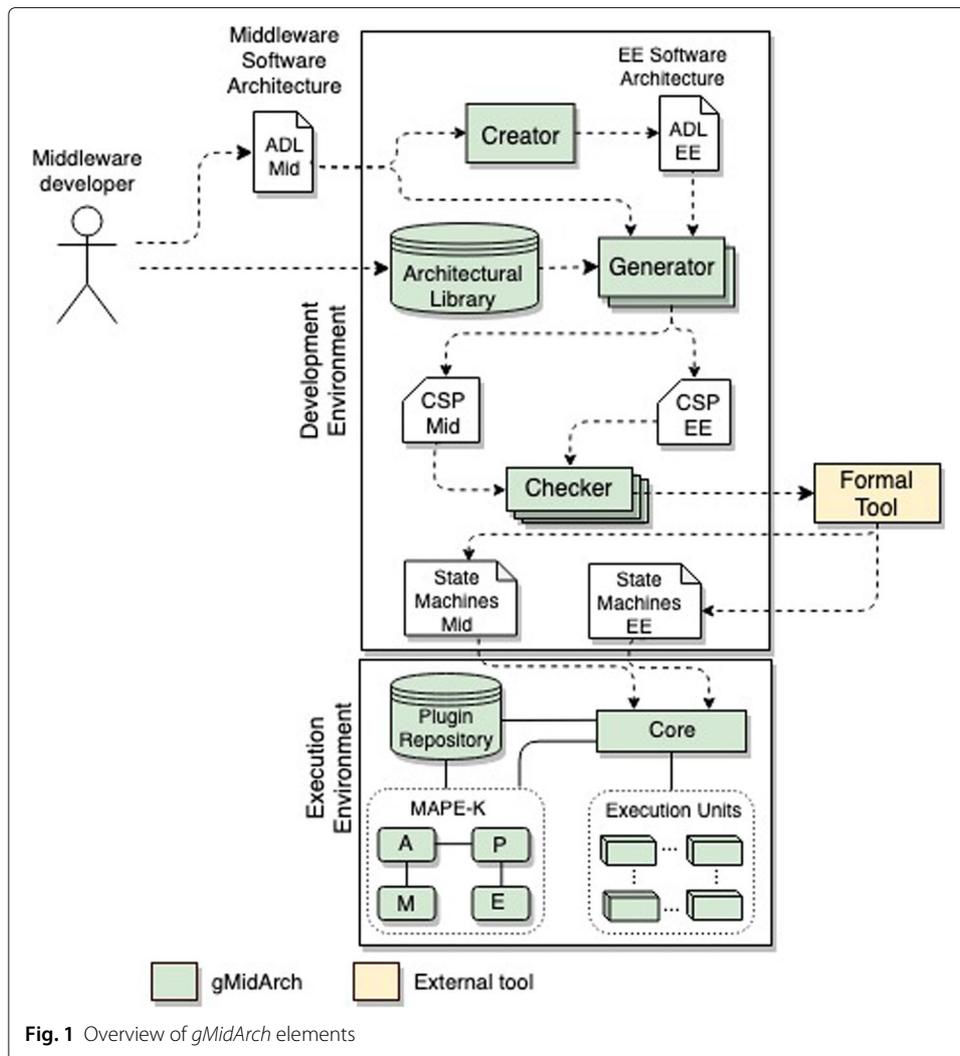


Fig. 1 Overview of *gMidArch* elements

in the *Architectural Library*. Then, *Generator* combines the specifications of $c1$, t and $c2$ in a CSP parallel composition.

Next, *Checker* invokes an external tool (*Formal Tool*) with two purposes: to verify desired properties of CSP specifications, e.g., deadlock freedom; and, if the properties are satisfied, to generate state machines to be deployed in the *Execution Environment*.

The execution environment is customised according to the middleware software architecture and consists of a *Core* that coordinates the interactions between *MAPE-K* elements and *Execution Units* whose role is to host architectural elements. Finally, *Plugin Repository* stores plugin versions of architectural elements. For example, when a new version of an existing component/connector is available, it is stored in the repository and becomes a candidate to replace the respective component/connector that is in execution.

Following subsections describe in details the proposed extensions.

3.1 Richer CSP subset

The lightweight formalisation is one of the essential characteristics of *gMidArch*. The formalisation consists of defining behavioural models in CSP that describe how

components/connectors behave while executing. These models are used to generate state machines that work as execution artefacts.

The CSP specification defines the temporal ordering of actions executed by a component or connector. The specification includes internal and external actions executed by the element. An internal action is something executed by the element, is usually associated with its business and does not involve any interaction with external elements. For example, a marshaller executes the (un)marshalling operation without the involvement of any external element. Meanwhile, external actions are used to describe interactions of the element with its partners (external world) and require synchronisation with them, e.g., the marshaller receives requests from external elements asking for (un)marshalling something, while sends responses to them.

In *MidArch*, behavioural specifications can only use the CSP prefix operator (\rightarrow), e.g., $e \rightarrow P$ informally means that an element performs the event e and then P runs. This operator enables us to specify behaviours as a sequence of actions where one is executed after another. For example, the behaviour of a middleware marshaller (B_M) can be easily specified like $B_M = invP.e1 \rightarrow i_Process \rightarrow terP.e1 \rightarrow B_M$. In this behaviour expression, the marshaller receives an *invocation* from external component $e1$ ($invPe1$), (un) marshalls it ($i_Process$), *terminates* the processing ($terPe1$) by sending a response to $e1$, and behaves like B_M again.

While useful, the prefix operator limits the kinds of behaviours that can be specified. In the previous example, the marshaller neither can receive a request from another component (e.g., $e2$) nor we can define behaviours in which the component fails. However, these specifications could also be extended to include the possibility of defining alternatives and/or choices to express more complex behaviours.

In practical terms, this limitation occurred due to the complexity of executing state machines in which, given a particular state, one or more actions are enabled to be executed simultaneously. Figure 2 shows examples of state machines generated only using prefix operator and one produced from more elaborated behaviours including CSP operators, such as external choice and parallel.

In this figure, the execution unit traverses the state machine and performs the action labelled in the edges. The execution of the second machine (Fig. 2b), however, requires the

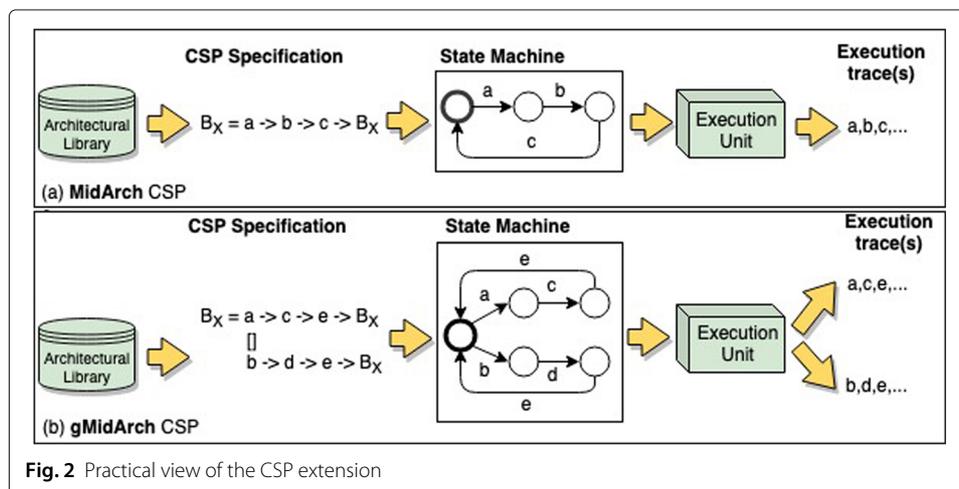


Fig. 2 Practical view of the CSP extension

selection of an event (e.g., *a* or *b*) from a list of events enabled simultaneously to be executed. The implementation of this kind of selection is not supported in Java. Section 3.5 will describe how the execution unit implemented in Go can execute the state machine shown in Fig. 2b.

3.2 Agnostic ADL

Whatever the implementation language (Java or Go), developers define the middleware software architecture using a programming language-agnostic ADL, namely *mADL* (Middleware Architecture Description Language). Along with *mADL*, we also implemented Go and Java generators able to create particular software artefacts to be deployed in the execution environment, according to the target language.

mADL is a declarative language and needs to be simple as it will be used by developers who are responsible for defining the middleware software architecture. The language provides three abstractions for specifying architectures: component, connector and configuration. A component is a piece of computation or a data storage that represents a broad range of distinct elements, e.g., a simple procedure or an entire application. A connector is an architectural building block used to model interactions among components. Lastly, the configuration describes how components and connectors are wired together [19, 32]. Additionally, *mADL* allows the definition of the required adaptation strategy: corrective, proactive and evolutive. The use of these abstractions is shown in the following example:

1. **Configuration** MiddlewareClient :=
2. **Components**
3. namingProxy: NamingClientProxy
4. appProxy : AppClientProxy
5. requestor : Requestor
6. crh : CRH
7. **Connectors**
8. t1 : NTo1
9. t2 : RequestReply
10. **Attachments**
11. namingProxy, t1, requestor
12. appProxy, t1, requestor
13. requestor, t2, crh
14. **Adaptability**
15. Evolutive
16. **EndConf**

This middleware configuration, named *MiddlewareClient*, includes four components (Lines 3-6) and two connectors (Lines 8-9) which are attached as shown in Lines 11-13, e.g., components *appProxy* and *requestor* are connected through connector *t1*. Finally, Lines 14-15 define the kind of adaptability required by the middleware, namely *evolutive*, *proactive* or *evolutive*. The first and second types of adaptations are supported at runtime by invoking process mining tools and using PRISM language as initially proposed in [29] and [26], respectively. Finally, the evolutive adaptation is a Go extension whose implementation is described in Section 3.6.

Components and connectors are typed, have associated a CSP specification, and are stored in the *Architectural Library* (see Fig. 1). Components have been defined according to existing architectural patterns widely adopted to implement RPC-based middleware systems [37]. Currently, we have defined eight different types of middleware components:

- *Client Proxy* supports the same interface as the remote component. it is used for remote invocations and serves as an entry point to the middleware. *Client Proxy* translates local invocation's parameters into parameters for the *Requestor*;
- *Marshaller* takes responsibility of (un)marshalling operations;
- *Requestor* constructs a remote invocation using parameters received from the *Client Proxy* and sends it to a remote component. In practice, it works as a coordinator that receives an invocation from the client, invokes a component to serialise it (*Marshaller*), and forwards the serialised invocation to the *Client Request Handler*. When the response is received from the remote component, it follows a reverse path until reaching the client.
- *Invoker* receives a remote invocation from the *Requestor*, unmarshals it and dispatches the invocation to a target remote component. Similarly to the *Requestor*, it coordinates the middleware actions on the server-side through interacting with the *Server Request Handler*, *Marshaller*, *Lifecycle Manager* and remote object.
- *Client Request Handler* (CRH) encapsulates communication activities at the client-side. For example, the CRH uses the socket API of the operating system to send/receive data to/from the server.
- *Server Request Handler* (SRH) encapsulates communication activities at the server-side. Similarly to CRH, the SRH uses the socket API of the operating system to receive/send data from/to the client.
- *Lifecycle Manager* implements lifecycle policies of remote components such as Static instance, Pooling and Client-dependent instance.
- *Lookup* works as a naming service.

In addition to the components, middleware developers can use four different kinds of connectors to define a software architecture in *mADL*: *OneWay*, *RequestReply*, *OneToN* and *NTo1*. *OneWay* mediates the interactions between components that simply send a message to each other (no reply); *RequestReply* is used in the communication of a component that makes a request and waits for a reply to/from another component; *OneToN* serves to replicate a message from one sender to N receivers; and *NTo1* receives messages from several senders and forwards them to a single receiver.

3.3 Formal models

The lightweight formalisation is one of the essential characteristics of *MidArch* and *gMidArch*. The formalisation consists of defining behavioural and probabilistic models in CSP and PRISM, respectively. The CSP models describe how components/connectors belonging to the framework behave while executes. Meanwhile, the probabilistic model is part of the *Non-Functional Properties* (quality properties) associated with components and connectors.

The behaviour specification is used to generate a state machine (graph) executed by at runtime. In *gMidArch*, the CSP specification of a typical middleware component, namely *Marshaller*, is defined as follows:

datatype PROCNAMES = e1

channel InvP, TerP : PROCNAMES

channel I_Process

Marshaller = InvP.e1 \rightarrow I_Process \rightarrow TerP.e1 \rightarrow Marshaller

In this specification, *Marshaller* receives an invocation from an external component *e1* (*invPe1*), processes it (*i_Process*), terminates the processing (*terPe1*) by sending a response to *e1*, and behaves like *Marshaller* again.

In relation to the PRISM models [26], *gMidArch* uses the continuous-time Markov chains (CTMC). The monitoring performed by the execution environment allows populating these models with rates measured while the middleware executes. Each architectural element is specified through a module consisting of some local variables, and whose values at any given time define the state of the module. For instance, the probabilistic model associated with component *Marshaller* is defined as follows:

1. **module** Marshaller
2. $s : [0..2]$ **init** 1;
- 3.
4. [invP] $s=1 \rightarrow (s'=2)$;
5. [invP] $s=2 \rightarrow (s'=s)$;
6. [process] $s=2 \rightarrow \text{rate_marshaller} : (s'=2)$;
7. [process_last] $s=2 \rightarrow \text{rate_marshaller} : (s'=1)$;
8. **endmodule**

The behaviour of *Marshaller* is described by four commands (Lines 4-7) and has been partially inspired in [24]. The command in Line 4 is labelled with action *invP* to indicate a possible engagement of this module to receive a request from an external component. The guard $s=1$ is a predicate over variable *s* that indicates the state of the *Marshaller*. Given that *Marshaller* is in the initial state ('1') and action *invP* occurs, the component's state is updated ($s'=2$). In Line 5, if a new request arrives while the component is processing the previous one, do nothing. This kind of strategy in the specification avoids blocking a component synchronised with *Marshaller*. As mentioned before, there is a rate associated with each state update. In Lines 6 and 7, *Marshaller* processes requests with rate *rate_marshall*.

3.4 Richer subset of components

gMidArch has a set of new components related to the middleware domain, namely MOM (Message-Oriented Middleware) components, and the execution environment. Three MOM-related components have been included in the architectural library following the generic publish/subscribe architecture introduced in [36]: *Notification Consumer*, responsible for notifying subscribers about the arrival of new messages; *Notification Engine* that implements operations to subscribe, unsubscribe and publish content; and *Subscription Manager*, responsible for managing subscriptions. Using these new components together with some existing ones (e.g., *SRH* and *CRH*), it becomes possible to use *gMidArch* also to design adaptive publish/subscribe middleware systems in addition to RPC-based ones as available in *MidArch*.

To implement the execution environment using software architecture elements, it was also necessary to extend the architectural library also to include new components (non-

middleware ones) whose functionalities are related to the elements of the execution environment, namely MAPE-K loop, execution units and core.

MAPE-K components include a set of monitors (*MAPEKEvolutiveMonitor*, *MAPEKCorrectiveMonitor*, *MAPEKProactiveMonitor*), an analyser (*MAPEKAnalyser*), a planner (*MAPEKPlanner*) and an executor (*MAPEKExecutor*). These elements are responsible for monitoring, making decisions about the need for adaptation, building adaptation plans and executing them, respectively. Meanwhile, execution units (*ExecutionUnit*) host architectural elements and work as a manageable unit in such a way that can be stopped, have the behaviour of its architectural element replaced and resumed dynamically. Finally, the *Core* is responsible for forwarding adaptation commands straight to the units.

3.5 Customised execution environment

As mentioned before, the execution environment was re-implemented in Go using software architecture principles in a similar way to the adaptive middleware. It is worth observing that in the Java version (*MidArch*), only the middleware was defined using software architecture abstractions. The use of these abstractions facilitates both the customisation at development time and adaptation at runtime of the environment.

As shown in Fig. 1, *Creator* generates a customised software architecture (*ADL EE*) of the execution environment from the middleware software architecture. The customisation means to define the elements of the MAPE-K loop and execution units needed to execute the middleware. For example, if the middleware requires a given type of adaptation, only execution elements needed to support it make up the execution environment architecture.

Figure 3 shows a software architecture (*MiddlewareClient*) and its respective customised execution environment (*EEMiddlewareClient*). *MiddlewareClient* is the client-side middleware software architecture (based on the middleware client architecture defined in Remoting Patterns [40]) and has four components (*namingProxy*, *appProxy*, *requestor* and *crh*), two connectors (*TA* and *TB*), and needs an *Evolutive* adaptation. The generated execution environment includes *Core*, *MAPE-K* elements, six execution units (one for each element of the middleware architecture) and six connectors (*t1*, *t2*, *t3*, *t4*, *t5* and *t6*).

As mentioned in the previous section, *Core* is responsible for coordinating the interactions between the MAPE-K elements and the execution units. Its behaviour consists of firstly initialising all execution units, waits for adaptations commands from the MAPE-K executor, and then executes then over the units.

The MAPE-K elements have been implemented as architectural components. It is worth observing that the *Monitor* function was subdivided into three parts: corrective monitor, proactive monitor and evolutive monitor. The corrective monitor was initially presented in [29], and the proactive monitor is described in [26]. The evolutive monitor consists of checking for new plugins in the *Plugin repository* and then sends information to the *Analyser* continuously.

Finally, each *execution unit* traverses the graph (state machine) whose nodes are the states of the architectural element, and the edges are labelled with actions executed by it. Algorithm 1 is implemented by the execution unit to perform this task.

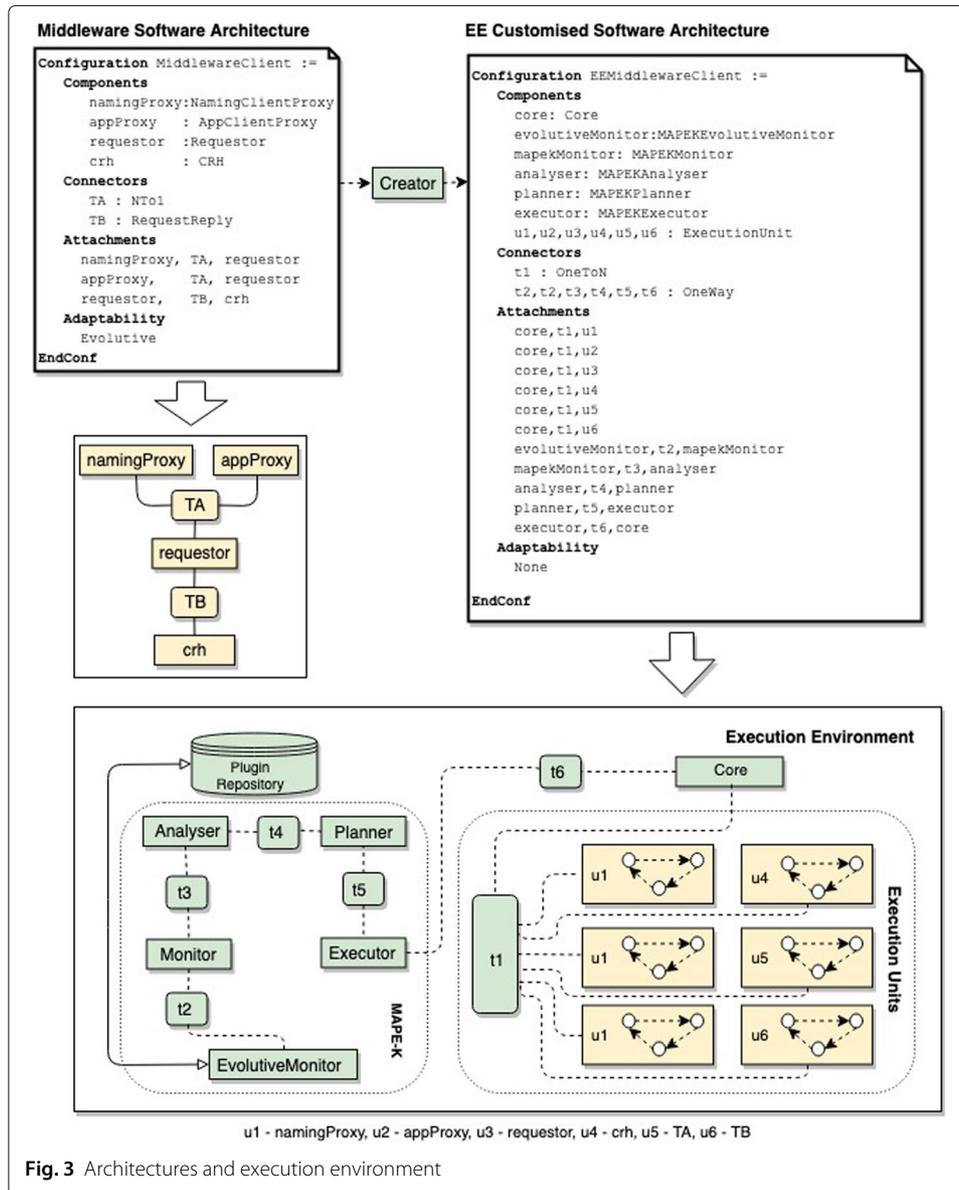


Fig. 3 Architectures and execution environment

This algorithm essentially traverses the graph whose nodes are the states of a component/connector, and the edges are labelled with actions executed by it. Given a node, it is necessary to get the node’s adjacent edges. If there is a single adjacent node (Line 5), it means that the next action to be executed is the one associated with the edge (Line 6). Otherwise, there is a branch in the graph that indicates the use of the choice or parallel operator (Line 8). In this case, function *Choice* (Line 9) must select an edge from a list (*edges*). The next node depends on the selected edge (Line 11). Finally, in both cases, the action associated with the edge is executed (Line 13).

The implementation of function *Choice* in Go is shown in the following:

```

1: func Choice(chosen *int, edges []wgraph.Edge) {
2:   cases := make([]reflect.SelectCase, len(edges))
3:   var value reflect.Value
4:   for i := 0; i < len(edges); i++ {

```

```

5:     cases[i] = reflect.SelectCase{Dir:
6:         reflect.SelectRecv, Chan:
7:         reflect.ValueOf(*edges[i].Action.P2)}
8:     }
9:     *chosen, _, _ := reflect.Select(cases)
10: }

```

Algorithm 1 Execution of the State Machine

```

1: function EXECUTESTATEMACHINE(graph)
2:     node = 0
3:     while true do
4:         edges = AdjacentEdges(graph, node)
5:         if len(edges) = 1 then                                     ▷ MidArch
6:             action = Event(edges[0])
7:             node = NextNode(edges[0])
8:         else                                                       ▷ gMidArch
9:             chosen = Choice(edges)                                  ▷ Function Choice (Go)
10:            action = Event(edges[chosen])
11:            node = NextNode(edges[chosen])
12:        end if
13:        Execute(action)
14:    end while
15: end function

```

The input parameters are a pointer to be set when a case is selected and the set of selectable edges, e.g., events *a* and *b* shown in Fig. 2. In Lines 2-8, this implementation uses reflection to create a set of case clauses (events) and the *select* statement that blocks until one of the events occurs (Line 9).

It is worth observing that the Go *select* statement lets a goroutine wait on multiple communication operations (events). The statement blocks until one of its cases can execute, then it runs that case. It chooses randomly if multiple cases are ready. The *select* statement made viable the implementation of an execution unit capable of executing the state machine shown in Fig. 2b. The use of reflection was also essential because the engine does not know the branches (*a* and *b*) statically and needs to define them at runtime. The list of branches (events) is created while the unit is transversing the state machine.

3.6 Evolutive adaptation

As soon as the execution environment starts the execution of the architecture, the adaptation manager (see Fig. 3) monitors it. Monitored data are forwarded to the analyser that decides for an adaptation. If an adaptation is necessary, the planner creates an adaptation plan and then forwards it to the executor (*E*) that performs the actions defined in the plan.

As mentioned in Section 2.2, *MidArch* already implemented corrective and proactive adaptations. *gMidArch* extends the adaptation strategies available in *MidArch*, and includes a new kind of adaptation, namely *evolutive adaptation*. Evolutive adaptations

occur when a new version of an existing component/connector (plugin) becomes available. After deployed, any architectural component can be potentially replaced as soon as a new plugin is available.

Whatever the adaptation strategy, all of them typically demand the replacement of a running element by another one. In Go, the replacement consists of loading a new plugin (new version of the architectural element), stopping the old one and starting the new plugin. Due to the strategy of using CSP annotations associated to components and connectors, two additional steps are performed: the checking of behavioural compatibility between the old and new plugins and the verification if the new component injects an undesirable behaviour in the architecture.

To understand how the plugin replacement works and facilitates the evolutive adaptation, it is necessary to observe that each architectural element (component and connector) has a non-plugin implementation (loaded statically when the architecture is deployed) and can have a plugin version (loaded dynamically). Mechanism similar to ones found in some existing languages, Go plugins allow developers to implement loosely coupled programs whose components can be dynamically loaded and bound at runtime. In practice, a plugin becomes an independent component whose development and lifecycle is entirely independent of other elements.

Figure 4 shows the replacement process. In parallel to the execution of the software architecture, the MAPE-K components start to run. Then, as soon as a new plugin is available (1), the evolutive monitor detects its presence (2) and then passes this information to the general monitor (3) who contacts the analyser (4). It is worth observing that the plugin also has a CSP behaviour (and its respective state machine) in a similar way to the non-plugin component. Next, the analyser checks the compatibility of the current behaviour and the behaviour of the new plugin. If they are compatible, the analyser generates a new CSP specification of the software architecture including the behaviour of the new plugin.

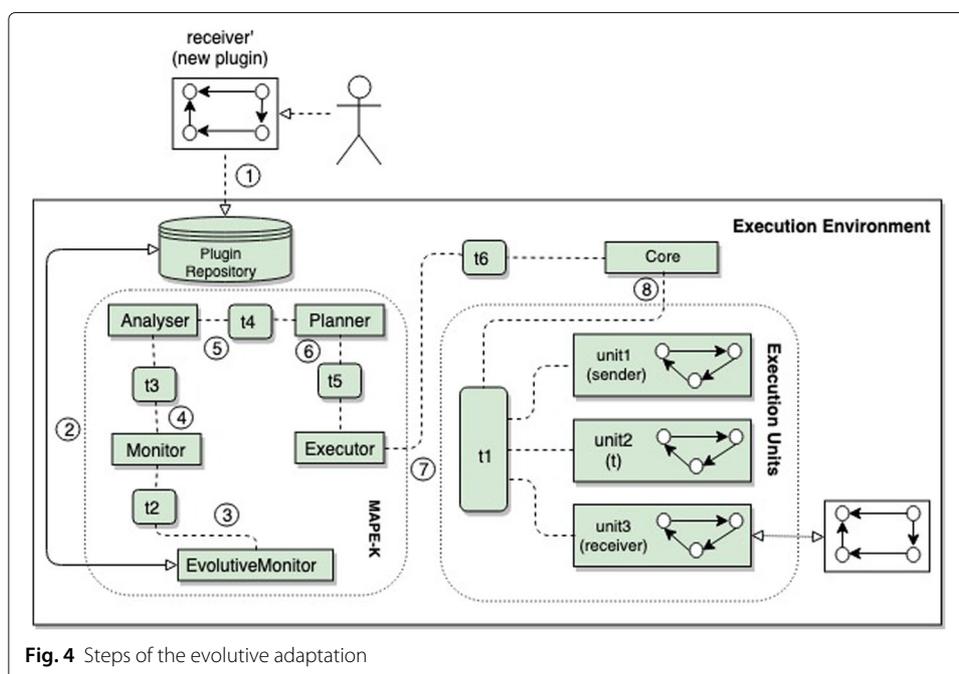


Fig. 4 Steps of the evolutive adaptation

If the specification has not an undesired behaviour like deadlock, it means that the new plugin can be deployed. Next, the analyser informs the planner about the new plugin (5). The planner creates an adaptation plan that needs to be executed to carry out the reconfiguration. The plan is passed to the executor (6). The executor notifies the core (7) that stops the execution unit, loads the new state machine and discards the old one (8).

As mentioned before, the replacement process may occur continuously. However, some points must be observed that are not apparent in Fig. 4. Firstly, the detection of a new plugin is simple, as they are stored in the repository. From time to time (configurable), the monitor checks the repository for new plugins. Secondly, two components are compatible if they have the same type (Go checking) and if the behaviour of the new plugin refines (in a particular semantic model) the behaviour of the old one (CSP checking). Using FDR4⁶ (The CSP Refinement Checker), this checking is performed through an assertion like *assert B1 [T = B2*, where *B1* and *B2* are the behaviours of the new and old plugins, respectively. Thirdly, the behaviour replacement only occurs when the old element is in its initial state that works as a quiescent state, i.e., no pending requests.

It is worth observing that the state machine execution only starts in an initial state and whatever the execution flow it returns to the beginning state. This behaviour is guaranteed as all CSP specifications associated with architectural elements are recursive, i.e., they have a general structure like $B = \langle \text{dosomething} \rangle - \rangle B$. Hence, if an element is in its initial state, no pending request or reply exist for this particular element. However, this fact does not mean that a client's remote request cannot be pending while a middleware element is replaced, for example. Some middleware elements are more time-sensitive to this strategy. For example, suppose that the CRH (see Section 3.2) is the element to be replaced. CRH is responsible for opening a connection, sending the request, receiving the response and closing the connection. In the approach adopted in *gMidArch*, this element cannot be replaced if a connection is opened.

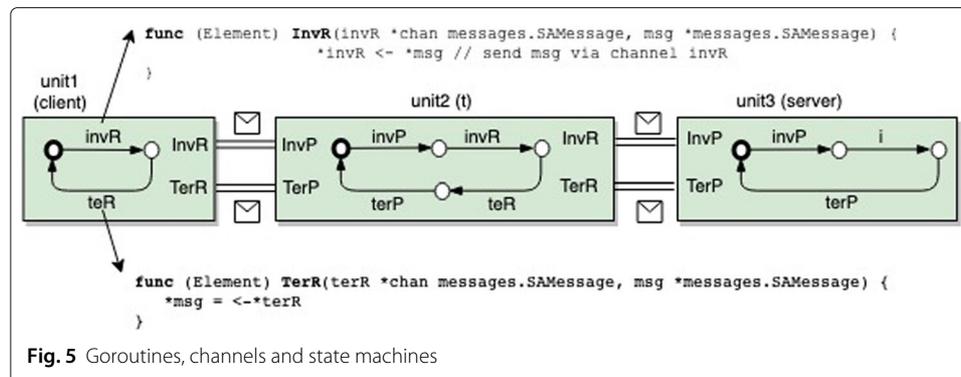
Fourthly, only stateless elements can be replaced in the current version. Fifthly, each execution unit and the MAPE-K element is defined as a goroutine (lightweight threads) that communicates through synchronous channels. Fifthly, solely the architectural element whose behaviour is replaced needs to be stopped. Finally, it is worth observing that the Go plugins need to implement two additional functions in relation to its non-plugin version: *GetTypeElement()* and *GetBehaviourExp()*. They return the plugin type and behaviour expression (CSP) of the plugin, respectively. Those functions are necessary for the type and behavioural compatibility checks mentioned before.

3.7 Go implementation details

All elements of the execution environment were implemented as goroutines, while they were Java threads in *MidArch* version. Goroutines have some characteristics like faster startup time than Java threads and a smaller number of OS threads as there is not a 1:1 mapping between them. Goroutines come with built-in primitives to safe communication between them, the so-called channels. The communication between elements of the execution environment occurs through non-buffered Go channels.

A key aspect to understand how the software architecture is executed in practice is to observe the relationships between goroutines, channels, CSP specifications and state

⁶<https://www.cs.ox.ac.uk/projects/fdr/>



machines (graphs). As mentioned in Section 3.1, the CSP specification that describes the behaviour of each architectural element (set of temporally ordered actions) is translated into a state machine whose edges are labelled by actions executed by the element.

As presented in Section 3.1, there are two kinds of actions, namely internal and external actions. There is a 1:1 mapping between external actions performed by the architectural element and the Go channels. Hence, when the graph that defines the behaviour of the architectural element is traversed, each external action associated with the edge has a function whose execution is to send/receive something via the channel. Four different external actions are allowed in *gMidArch*: *invR* and *terR* to send a request and receive a response from an external element, respectively; and *invP* and *terP* to receive a request and send a response to another element, respectively. Meanwhile, each internal action has a Go function associated that depends on the functionality implemented by the architectural element.

In Fig. 5, as execution unit *unit1* transverses the state machine, the actions within the edges are executed. The execution of the first action (*InvR*) means to invoke function *InvR* that places a message (*msg*) into channel *invR*. Channels *invR* and *invP* are shared between *unit1* and *unit2* and when *unit1* executes *InvR* and *unit2* executes *InvP*, goroutines *unit1* and *unit2* synchronise and the message moves from *unit1* to *unit2*. The same process repeats in relation to other units and actions.

4 Evaluation

The objective of this evaluation is to compare the performance of a distributed application built atop RPC/MOM implementations of *gMidArch* with existing middleware systems. Two widely adopted non-adaptive commercial middleware systems were considered, namely gRPC⁷ and RabbitMQ⁸. Meanwhile, a similar comparison was also carried out with a non-commercial RPC-based adaptive middleware implemented in Haskell [33]. Table 2 summarises the scenarios of evaluation that have been investigated.

It is worth observing some facts about these comparisons. Firstly, to the best of our knowledge, there are no commercial adaptive middleware systems. Secondly, these evaluations exercise the two different middleware models that can be implemented using *gMidArch*: RPC-based and MOM. Thirdly, it is important to evaluate the performance of non-adaptive versions of *gMidArch* because it helps to show the viability of their use even

⁷<https://grpc.io/>

⁸<https://www.rabbitmq.com>

Table 2 Scenarios of comparison

Scenario	gMidArch		Existing Middleware
	Flavour	Adaptation	
#1	RPC	Disabled	gRPC
#2	RPC	Enabled	gRPC
#3	MOM	Disabled	RabbitMQ
#4	MOM	Enabled	RabbitMQ
#5	RPC	Disabled	AFirM
#6	RPC	Enabled	AFirM

in scenarios in which adaptation is not mandatory. Finally, the comparison with an adaptive middleware that provides a similar kind of adaptation (evolutive) is an ideal scenario as both middleware systems become very comparable.

The metric used in the experiments was the *response time*, which is measured on the client side and refers to the time elapsed between the client makes a request and receives a response. As the focus is on the middleware, the remote function (namely *fibonacci*) invoked by the client is a simple one that recursively calculates a Fibonacci sequence number. In practice, each request passes through the client's side middleware (see configuration shown in Section 3.2) and its respective server side before is executed remotely. The application response also passes by both middleware configurations (client and server sides). While simple, the Fibonacci application uses all components and connectors of the middleware (similar to more complex applications), which is a fundamental requirement in the evaluation. Meanwhile, the application is also easy to be deployed. Finally, despite its simplicity, the Fibonacci can be highly demanding in terms of processing, e.g., $n > 34$.

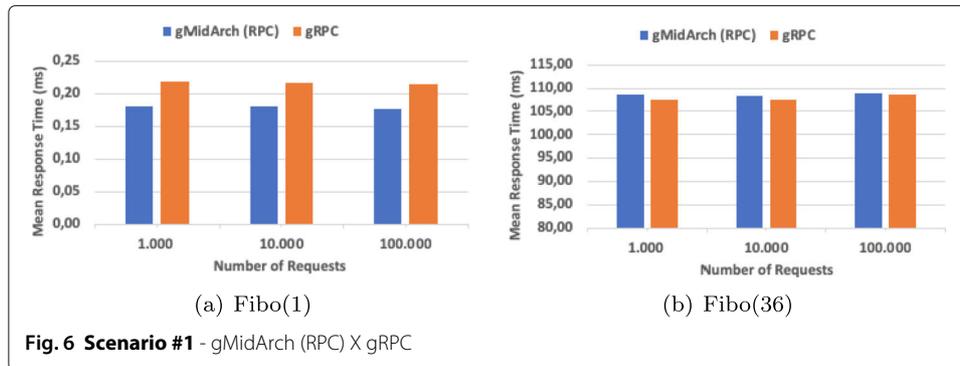
All elements of the application and middleware flavours were deployed in three official Docker containers for Go⁹: a container with the client, a container having the server and a container executing the *gMidArch* naming service. The containers run on a MacBook Pro with a 2,9 GHz Intel Core i7, 8 GB RAM, and MacOS Catalina, version 10.15.2.

In the experiments, several parameters were varied: the number of requests performed by the client (1.000, 2.000, 4.000, 8.000, 10.000, 50.000 and 100.000); versioning interval (1s, 10s, 300s), which means the time between updates of the middleware component being replaced; fibonacci number (1, 34 and 36); and middleware flavours (RPC/MOM *gMidArch*, *gRPC*, *RabbitMQ* and *AFirM*).

Figure 6 shows the results¹⁰ of the experiments to compare the performance of *gMidArch* and *gRPC* varying the Fibonacci number (1 and 36) and number of requests (1.000, 10.000 and 100.000). In this comparison, it is possible to observe that the performance of *gMidArch* is 20.77% better than *gRPC* when the remote operation has a quick response (*Fibonacci(1)*). However, these results become inverted when the remote operation is very processing-intensive (*Fibonacci(36)*). In the latter, it takes around 107 ms to complete, i.e., around 500 times longer than the former. In this case, *gMidArch* is 0.64% slower than *gRPC*. It is worth noticing that the implementation of Fibonacci is the same in both cases. A possible reason for this behaviour is the fact that *gMidArch* is more processing intensive due to the number of Go routines created. When the Fibonacci computation becomes more processing-intensive (*Fibonacci(36)*), there is a higher CPU contention between the Fibonacci and the middleware.

⁹https://hub.docker.com/_/golang/

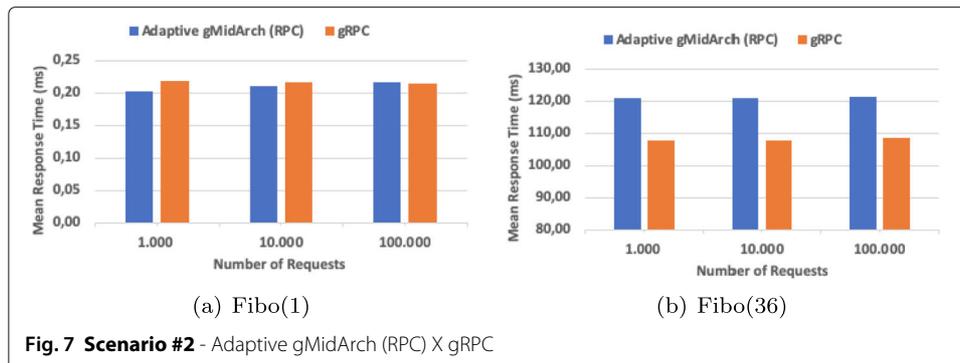
¹⁰<https://github.com/gfads/midarch/tree/nelson/evaluation>

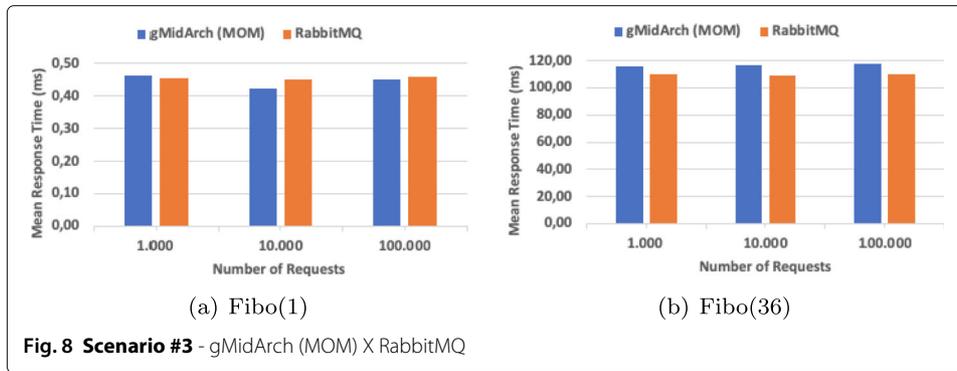


In the next scenario, Scenario #2, the RPC *gMidArch* implementation was configured to allow runtime adaptations. In practice, the *Adaptability* of the middleware architecture was set to *Evolutive* (see Section 3.2). As a consequence, the MAPE-K feedback loop becomes active while the middleware executes and triggers adaptations. To make possible the execution of this kind of experiment, a versioning injector was also implemented. It executes in parallel to the middleware and generates a new version of the element being adapted from time to time (versioning interval). In particular, in this scenario, the versioning interval was set to 1s, which means that a new adaptation is triggered every one second.

Figure 7 shows the results of some experiments executed in this scenario. The performance of *gMidArch* is still better than gRPC in the case of *Fibon(1)*, but there was a reduction from 20.77% (Scenario #1) to 2.93% (Scenario #2). Similarly to the previous scenario, the performance of gRPC is also better (10.84%) than *gMidArch* when the remote function is *Fibon(36)*. The gain of gRPC increases from 0.64% to 10.74% in this case. The versioning injector (not existing in the previous scenario) responsible for generating new plugin versions (every 1s) helps to justify this deterioration partially. Additionally, as the experiments for invoking *Fibon(36)* are much more longer than ones of *Fibon(1)*, the number of adaptations in the former is much more higher than the latter.

In order to evaluate the performance of a MOM implemented using *gMidArch* (*gMidArch* MOM), Scenario #3 shows a comparison of this middleware with RabbitMQ. In this scenario, the same client/server application used in Scenarios 1 and 2 was fully re-implemented atop *gMidArch* MOM and RabbitMQ. As the RabbitMQ is not adaptive, the





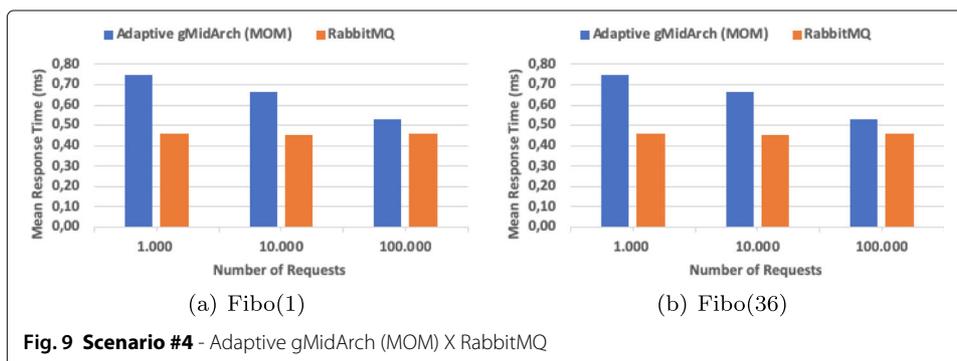
adaptation of the *gMidArch* MOM was turned off in this scenario. In both implementations, two queues are used in the client and server communication: *request* and *reply*. The client publishes a remote invocation in queue *request* and subscribes to replies in queue *reply*. Meanwhile, the server is a subscriber of queue *request* and publishes responses (the result of the calculation) in queue *reply*.

Figure 8 shows some results obtained in this scenario. Similarly to the previous experiments with gRPC, the performance of *gMidArch* MOM is 2.64% better than the commercial middleware (RabbitMQ) in the case of *Fibon(1)*. The only difference is the fact that the gain of *gMidArch* MOM was reduced from 20.77% (*gMidArch* RPC) to 2.64%, which may indicate that the MOM implementation of *gMidArch* is less performative than the RPC one. At the same time, the performance of RabbitMQ is 6,09% better than *gMidArch* MOM when the remote invocation is more processing-intensive (*Fibon(36)*). This result is also compatible with Scenario #1, in which the CPU contention helps to explain this behaviour.

In the next set of experiments, Scenario #4, an adaptive version of *gMidArch* is compared with RabbitMQ. An injector generates a new version of the *Invoker* every second like in Scenario #2.

The results presented in Fig. 9 show that the performance of RabbitMQ is 27.91% (*Fibon(1)*) and 19.82% (*Fibon(36)*) better than the adaptive version of the MOM *gMidArch*. This result is also influenced by the CPU contention mentioned before. However, the difference in performance also shows that the adaptation process in the MOM *gMidArch* is more costly in terms of time.

Unlike the previous scenarios, Scenario #5 shows a comparison with an adaptive middleware (AFirM). AFirM is an RPC-based middleware implemented in Haskell that



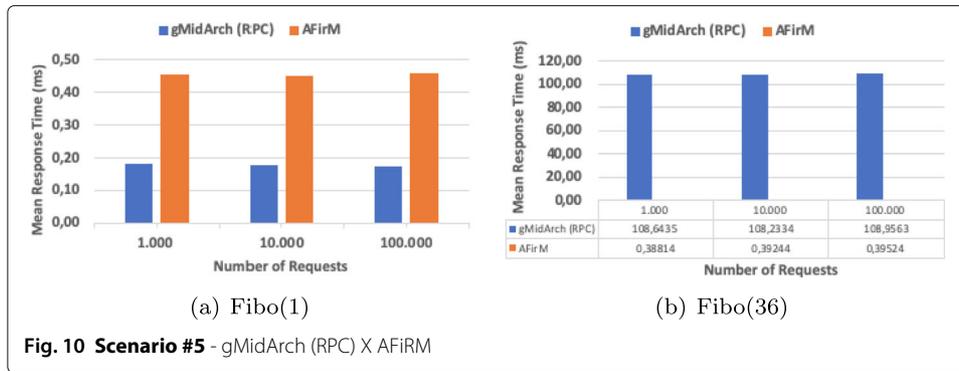


Fig. 10 Scenario #5 - gMidArch (RPC) X AFiRM

supports evolutive adaptations. Then, the Fibonacci client/server application was re-implemented in Haskell. Similarly to the previous scenarios, the application was executed having different configuration parameters.

Figure 10 shows the results obtained in this scenario. In the first case (*Fibo(1)*), the performance of *gMidArch* is 120.03% better than *AFirM*, which is a very good result. In the second case (*Fibo(36)*), something interesting happens. The performance of *AFirM* is much better than *gMidArch* and is very similar in both cases (*Fibo(1)* and *Fibo(36)*). This fact reveals that the Fibonacci implementation in Haskell is much more efficient than one in Go. In particular, this solution uses bitwise operations. As a consequence, the response time in *AFirM* is practically stable whatever the Fibonacci number. However, it worth observing that the behaviour of *gMidArch*, whose response time increases when the remote invocation changes from *Fibo(1)* to *Fibo(36)*, is similar to the behaviours of *gRPC* and *RabbitMQ* as shown in the previous scenarios.

In Scenario 6, the evolutive adaptation is active in *gMidArch* and *AFirM*. In both cases, a new version of a middleware component is generated every 1 second by a versioning injector. Figure 11 shows that the performance of *gMidArch* is better than *AFirM* in the case of *Fibo(1)*. The gain of *gMidArch* in this adaptive scenario increases from 120.03% (Scenario #5) to 535,31%. Similarly to Scenario #5, however, *AFirM* has a better performance when the remote invocation is *Fibo(36)*. As mentioned before, this significant difference is related to the efficiency of the implementation of Fibonacci in Haskell.

Finally, it is worth observing that the evaluation of *gMidArch* could also be focused on the development framework (*mADL* and library architectural elements). The assessment of the framework could be done by (1) evaluating the usability of *mADL*, (2) analysing the completeness of the architectural elements for implementing different middleware models, or (3) even measuring the programming effort for developing adaptive middleware

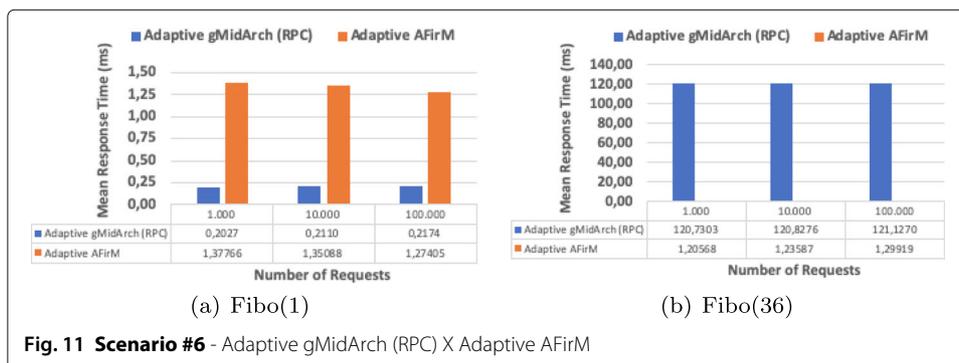


Fig. 11 Scenario #6 - Adaptive gMidArch (RPC) X Adaptive AFiRM

systems using the proposed framework, e.g., in a simple way, to count the number of lines of code required to implement a middleware with/without *gMidArch*.

While this kind of evaluation matters, to compare the performance of a new middleware with existing solutions is usually one of the first steps for deciding to use or not a middleware. Furthermore, performance evaluation is often very objective, depends only on code instrumentation to be done, and one of the criteria commonly used to compare middleware systems [8, 39].

5 Related work

The development of adaptive middleware is not a recent topic and pioneer works based on architectural reflection are widely known [4, 6, 17]. Meanwhile, the use of the MAPE-K feedback loop to manage the middleware adaptation is something very recent [23, 25, 29, 42]. Unlike the first group, *gMidArch* is founded on software architecture principles as enabling technology for adaptation.

Solutions to facilitate the implementation of middleware systems are not recent. Middleware frameworks such as Quarterware [34], PolyORB [38] and Arcademis [22] were pioneers in this topic. Even sharing the idea of an implementation framework, these solutions have neither formal elements nor focus on adaptive middleware.

Following the same idea of developing middleware by combining existing middleware elements, Issarny [15] and Costa [7] focus on composing service specifications to define software architectures and combining building blocks (metamodels) to create middleware configurations, respectively. Unlike *gMidArch*, they do not address adaptive issues, use formal techniques nor covers implementation and execution activities.

Moving to the integration of formal verification as proposed by *gMidArch*, Caporuscio [5] defined a methodology to simplify the verification of behavioural properties of middleware-based software architectures. However, the formalisation is restricted to the software architecture, and it does not cover adaptation issues neither is used at runtime.

Apart from the middleware community, Rainbow [12] is a framework that provides supporting mechanisms for self-adaptation and a language (Stitch) that can be used to codify adaptation techniques. It implements a MAPE-K feedback loop to manage and trigger the adaptation when a structural property is not satisfied. *gMidArch* also checks behavioural and quality properties in addition to structural ones. Furthermore, *gMidArch* is a middleware-specific framework and uses lightweight formalisation.

The combination of formal methods and the adaptation mechanism has a variety of faces: incorporation of formal methods into a reflective component model to verify whether a runtime adaptation would violate structural constraints specified in ALLOY [41]; use of Petri nets to formally check if a given configuration complies with functional and non-functional properties of the system [10]; use of the automata-based language Heptagon/BZR (combined with a transactional middleware) in the feedback loop of the adaptive middleware [35]; and the use of CADP tooling [27] and process mining [29] to trigger adaptations, and FDR to verify properties after adaptation of middleware systems. While the first three approaches use formal methods, they have not focused on taking benefits of formalisms while the middleware is being developed. Meanwhile, the last set of works has been extended in this paper in some ways: the use of software architecture as development and runtime artefact, and the definition of PRISM models to check quality properties.

Real-time reconfiguration based on the use of formal methods has been adopted in systems immersed in environments inherently dynamic, like cyber-physical systems. For example, [11] uses Petri Nets and [3] adopts constraint LTL over locks (CLTL_{oc}). Although these works use formal methods, their focus is on real-time applications having severe time constraints. Additionally, *gMidArch* also proposes a solution for helping adaptive middleware developers.

Finally, existing solutions also focus on verifying particular properties associated with fault-tolerance while the system executes [2]. Although the proposed environment (MAFALDA) monitors and identifies a violation of assertions, it does not act to fix the problem. Additional differences to *gMidArch* refer to the absence of the use of formal models focus on middleware specific issues and lack of a middleware development framework.

6 Conclusion and future work

Our unique contributions in this paper include the proposed architecture description language (*mADL*), the possibility of using an extended set of CSP operators to specify the behaviour of architectural elements, a richer set of architectural elements associated to the execution environment and MOM-based solutions, the possibility of customise the execution environment according to the middleware built atop it, and the evolutive adaptation. The extensive use of goroutines, channels and plugins also led to a very efficient implementation of the middleware whose performance is comparable to gRPC and RabbitMQ in the evaluated scenarios.

While the proposed extensions make advances on the original solution (*MidArch*), some decisions and paths took in *gMidArch* could be adopted for adaptive middleware systems in general. The use of software architecture concepts (component, connector, configuration, architectural style and architecture description language) serves as an interesting enabling technology for structuring the middleware whatever its implementation language or middleware model. *gMidArch* explicitly separates the adaption logic from the middleware and uses MAPE-K concepts to structure the adaptation manager. Adaptive middleware developers could use a similar approach as MAPE-K is a kind of de facto standard for implementing adaptive software systems. Finally, experiences of *MidArch* and *gMidArch* are movements to adopt formal methods beyond their traditional utilisation, i.e., as a specification tool only. *gMidArch* uses them effectively at execution time. As a consequence of this attempt is the possibility of advances in the development of safer adaptive middleware systems.

Considering the current state of the solution, one interesting future work is to allow dynamic changes in the execution environment, e.g., to replace the analyser to use machine learning techniques instead of PRISM models. Also, concerning adaptation, the proposed executor should be designed to allow the adaptation of simultaneous components/connectors, or even an entire middleware layer. A more elaborated experimental evaluation of the solution is also necessary: to work with applications having concurrent accesses to the middleware and replicated services; to better understand the Go's overhead for plug-in replacement; and to understand how the verification scales with the increase of the middleware architecture. Finally, as a framework, it is also interesting to evaluate its usability and how it reduces the programming effort to implement a middleware.

Abbreviations

IoT: Internet of things; ADL: Architecture description language; MOM: Message-oriented middleware; RPC: Remote procedure call; gRPC: Google remote procedure call; MAPE-K: Monitor-analyze-plan-execute over a shared knowledge; CSP: Communicating sequential processes; FDR: Failures-divergences refinement; PROM: Process mining

Acknowledgements

Not applicable.

Authors' contributions

SR was responsible for the identification of the problem, design and implemented the solution. JMC and MC carried out the experiments shown in this paper. Finally, SR and MC helped to draft the manuscript. All authors read and approved the final manuscript.

Authors' information

SR received his BSc, MSc and PhD degrees in Computer Science from the Federal University of Pernambuco, Brazil. He is an Associate Professor in the Centre for Informatics of the Federal University of Pernambuco, Brazil. His current interests include formalisation, design and implementation of middleware and service-oriented systems. He published more than sixty research papers, and he was the Brazilian leader on a three-year collaborative project with HP Labs (Palo Alto). He was the receiver of the IEEE International Services Cup award in 2011. He has served as the committee member in several conferences on computer networks and distributed systems.

JMC received the M.Sc. degree in computer science from the Centre of Informatics, Federal University of Pernambuco, Brazil. He is a Professor at Federal Institute of Education, Science, and Technology of Pernambuco, Palmares campus, Brazil. He is currently a Research Fellow at the Centre of Informatics, Federal University of Pernambuco, Brazil. His research interests include formalisation, service-oriented computing, and middleware.

MC received the M.Sc. degree in Computer Science from the Department of Informatics and Applied Mathematics (DIMAP), Center of Exact and Earth Sciences, Federal University of Rio Grande do Norte (UFRN), Brazil. She is an Assistant Professor in the Department of Computer Science at State University of Rio Grande do Norte, Natal, Brazil. She is currently a PhD student in the Centre for Informatics at the Federal University of Pernambuco, Recife, Brazil. She is now working with adaptive service-based applications. Her research interests include formalisation, service-oriented computing, adaptive applications, and business process management.

GPS holds a degree in Data Processing from Universidade Potiguar (1997) and a master's degree in Systems and Computing from the Federal University of Rio Grande do Norte (2010). He is currently an adjunct professor at the State University of Rio Grande do Norte and a PhD student in Computer Science at the Federal University of Pernambuco.

André has experience in Computer Science, with an emphasis on Distributed Systems, acting mainly on the following topics: Distributed Multimedia Systems, Adaptive Middleware.

Funding

This research is partially sponsored by CNPq (Brazilian Research Council) - 404728/2016-2.

Availability of data and materials

Data of the experimental evaluation available at <https://github.com/gfads/mid\discretionary-arch/tree/nelson/evaluation>.

Competing interests

The authors declare that they have no competing interests.

Received: 24 April 2019 Accepted: 22 April 2020

Published online: 14 May 2020

References

1. van der Aalst W. 2016. *Process mining - data science in action*: Springer.
2. Arlat J, Fabre JC, Rodriguez M. Dependability of COTS microkernel-based systems. *IEEE Trans Comput.* 2002;51(2): 138–63.
3. Bersani MM, Garcia-Valls M. Online verification in cyber-physical systems: Practical bounds for meaningful temporal costs. *J Softw: Evol Process.* 2017;30(3):.
4. Blair GS, Coulson G, Andersen A, Blair L, Clarke M, Costa F, Duran-Limon H, Fitzpatrick T, Johnston L, Moreira R, Parlavantzas N, Saikoski K. The design and implementation of Open ORB 2. *IEEE Distrib Syst Online.* 2001;2(6):.
5. Caporuscio M, Inverardi P, Pelliccione P. Compositional verification of middleware-based software architecture descriptions. In: *Proceedings. 26th International Conference on Software Engineering*; 2004. p. 221–30. <https://doi.org/10.1109/icse.2004.1317444>.
6. Clarke M, Blair GS, Coulson G, Parlavantzas N. An efficient component model for the construction of adaptive middleware. In: *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. London, UK: Springer-Verlag; 2001. p. 160–78. <http://dl.acm.org/citation.cfm?id=646591.697779>.
7. Costa FM, Morris KA, Kon F, Clarke PJ. Model-Driven Domain-Specific Middleware. In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*; 2017. p. 1961–71. <https://doi.org/10.1109/icdcs.2017.197>.
8. Dobbelaere P, Esmaili KS. Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In: *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, Association for Computing Machinery. New York, NY, USA: DEBS 17; 2017. p. 227–38.
9. Donovan AA, Kernighan BW. *The Go Programming Language*, 1st edn: Addison-Wesley Professional; 2015.
10. García-Valls M, Perez-Palacin D, Mirandola R. Extending the verification capabilities of middleware for reliable distributed self-adaptive system. In: *INDIN*; 2014. p. 164–9. <https://doi.org/10.1109/indin.2014.6945502>.

11. Garcia-Valls M, Perez-Palacin D, Mirandola R. Pragmatic cyber physical systems design based on parametric models. *J Syst Softw.* 2018;144:559–72.
12. Garlan D, Schmerl B, Cheng SW. *Software Architecture-Based Self-Adaptation*: Springer; 2009, pp. 31–55. https://doi.org/10.1007/978-0-387-89828-5_2.
13. Hoare CAR. *Communicating Sequential Processes*: Prentice-Hall, Inc.; 1985. https://doi.org/10.1007/978-3-662-09507-2_19.
14. IBM. An architectural blueprint for autonomic computing. Tech. rep: IBM; 2005.
15. Issarny V, Kloukinas C, Zarras A. Systematic aid for developing middleware architectures. *Commun ACM.* 2002;45(6):53–8.
16. Kliem A, Boelke A, Grohnert A, Traeder N. Self-adaptive middleware for ubiquitous medical device integration. In: *e-Health Networking, Applications and Services (Healthcom) 2014 IEEE 16th International Conference on*; 2014. p. 298–304. <https://doi.org/10.1109/healthcom.2014.7001858>.
17. Kon F, Roman M, Liu P, Mao J, Yamane T, Magalhaes L, Campbell R. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In: *Middleware 2000*, vol 1795. Springer; 2000. p. 121–43. https://doi.org/10.1007/3-540-45559-0_7.
18. Kwiatkowska M, Norman G, Parker D. PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan G, Qadeer S, editors. *Proc. 23rd International Conference on Computer Aided Verification (CAV'11), LNCS*, vol 6806. Springer; 2011. p. 585–91. https://doi.org/10.1007/978-3-642-22110-1_47.
19. Medvidovic N, Taylor RN. A classification and comparison framework for software architecture description languages. *IEEE Trans Softw Eng.* 2000;26(1):70–93.
20. Meyerson J. The go programming language. *IEEE Softw.* 2014;31(5):104. <https://doi.org/10.1109/MS.2014.127>.
21. Park S, Song J. Self-adaptive middleware framework for internet of things. In: *2015 IEEE 4th Global Conference on Consumer Electronics (GCCE)*; 2015. p. 81–2. <https://doi.org/10.1109/gcce.2015.7398741>.
22. Pereira FMQ, Valente MTO, Bigonha RS, Bigonha MAS. Arcademis: A Framework for Object-oriented Communication Middleware Development. *Softw Pract Exp.* 2006;36(5):495–512.
23. Portocarrero JMT, Delicato FC, Pires PF, Rodrigues TC, Batista TV. SAMSON: Self-adaptive Middleware for Wireless Sensor Networks. In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16*; 2016. p. 1315–22. <https://doi.org/10.1145/2851613.2851766>.
24. Qiu Q, Wu Q, Pedram M. Stochastic modeling of a power-managed system: construction and optimization. In: *Proceedings 1999 International Symposium on Low Power Electronics and Design (Cat. No.99TH8477)*; 1999. p. 194–199. <https://doi.org/10.1109/lpe.1999.799438>.
25. Rafique A, Van Landuyt D, Reniers V, Joosen W. Towards an Adaptive Middleware for Efficient Multi-Cloud Data Storage. In: *Proceedings of the 4th Workshop on CrossCloud Infrastructures & Platforms, Crosscloud'17*; 2017. p. 4:1–6. <https://doi.org/10.1145/3069383.3069387>.
26. Rosa CGMMN, Cavalcanti DJM. Lightweight formalisation of adaptive middleware. *J Syst Archit.* 2018. <https://doi.org/10.1016/j.sysarc.2018.12.002>. <http://www.sciencedirect.com/science/article/pii/S1383762118300936>.
27. Rosa N. Middleware Reconfiguration Relying on Formal Methods. In: *2015 IEEE International Conference on Computer and Information Technology*; 2015. p. 648–55. <https://doi.org/10.1109/cit/iucc/dasc/picom.2015.93>.
28. Rosa N, Campos G, Cavalcanti D. Using software architecture principles and lightweight formalisation to build adaptive middleware. In: *Proceedings of the ARM 2017, ARM '17*; 2017. p. 1:1–7. <https://doi.org/10.1145/3152881.3152882>.
29. Rosa NS. Middleware Adaptation through Process Mining. In: *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*; 2017. p. 244–251. <https://doi.org/10.1109/aina.2017.25>.
30. Salehie M, Tahvildari L. Self-adaptive Software: Landscape and Research Challenges. *ACM Trans Auton Adapt Syst.* 2009;4(2):14:1–42.
31. Sarray I, Ressouche A, Gaffé D, Tigli JY, Lavirotte S. Safe Composition in Middleware for the Internet of Things. In: *Proceedings of the 2Nd Workshop on Middleware for Context-Aware Applications in the IoT, M4IoT*; 2015. p. 7–12. <https://doi.org/10.1145/2836127.2836131>.
32. Shaw M, Garlan D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall; 1996.
33. Silva A, Rosa N. Afirm: An adaptive functional middleware. In: *Proceedings of the 18th Workshop on Adaptive and Reflexive Middleware, ARM 19*. New York, NY, USA: Association for Computing Machinery; 2019. p. 13–8.
34. Singhai A, Singhai A, Sane A, Campbell R. Quarterware for Middleware. In: Sane A, editor. *Proc. 18th International Conference on Distributed Computing Systems*; 1998. p. 192–201. <https://doi.org/10.1109/icdcs.1998.679502>.
35. Sylla AN, Louvel M, Rutten E. Combining transactional and behavioural reliability in adaptive middleware. In: *ARM 2016*. New York, NY, USA: ACM; 2016. p. 5:1–6.
36. Tarkoma S. *Publish / Subscribe Systems: Design and Principles*: Wiley; 2012.
37. Uddin M, Akbar M. Addressing Techniques in Wireless Sensor Networks: A Short Survey. In: *Proc. International Conference on Electrical and Computer Engineering ICECE '06*; 2006. p. 581–4. <https://doi.org/10.1109/ICECE.2006.355698>.
38. Vergnaud T, Hugues J, Pautet L, Kordon F. PolyORB: A Schizophrenic Middleware to Build Versatile Reliable Distributed Applications. 2004:106–19. https://doi.org/10.1007/978-3-540-24841-5_8.
39. Vinoski S. The performance presumption [middleware evaluation]. *IEEE Internet Comput.* 2003;7(2):88–90. <https://doi.org/10.1109/MIC.2003.1189194>.
40. Volter M, Kircher M, Zdun U. *Remoting Patterns: Foundations of Enterprise, Internet and Real Time Distributed Object Middleware*: Wiley; 2005.
41. Warren I, Sun J, Krishnamohan S, Weerasinghe T. An automated formal approach to managing dynamic reconfiguration. In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*; 2006. p. 37–46. <https://doi.org/10.1109/ase.2006.12>.
42. Yaghoobirafi K, Nazemi E. A Self-Adaptive Middleware for Attaining Semantic Self-Interoperation Property. In: *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*; 2016. p. 293–8. <https://doi.org/10.1109/fas-w.2016.70>.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.