Multiple transport protocols in an adaptive RPC-based framework

Daniel C. Brandão
Universidade Federal de Pernambuco
dcb@cin.ufpe.br

Nelson S. Rosa
Universidade Federal de Pernambuco
nsr@cin.ufpe.br

Abstract

The growing demand for distributed systems running in many environments and built atop heterogeneous transport protocols is apparent. However, existing middleware solutions commonly are built atop a unique protocol like TCP. This paper extends an existing framework for building middleware systems by adding several communications protocols. The proposed extensions allow developers to implement a middleware using distinct communication protocols (e.g., UDP, HTTP) or even replace them at runtime. An experimental evaluation was conducted (1) to show the impact of the new extensions on the application's performance and (2) to compare the performance of the proposed extensions with existing commercial middleware systems.

1. Introduction

Distributed systems are widely adopted nowadays in different application domains and execute in increasingly heterogeneous and dynamic environments. On one side, the heterogeneity of environments leads to the diversity of communication protocols, interaction patterns and configuration setups. On the other side, dynamic environments generally mean that both systems' workloads and resource availability change while the system executes. Whatever the support provided to system developers, heterogeneity and adaptability issues are commonly under the adaptive middleware's responsibility.

While the middleware community has widely incorporated adaptive mechanisms into middleware solutions [1], these solutions usually lack support to communication protocols' heterogeneity. In practice, existing non-adaptive and adaptive middleware systems usually work atop a unique transport protocol, e.g., TCP or HTTP. Consequently, middleware solutions usually have a single built-in communication protocol fixed at development time that keeps immutable at runtime. While this is a limitation, accommodating different communication protocols having incompatible behaviours or security setups in a single middleware architecture is challenging.

This paper presents an extension of an existing framework named gMidArch [1]. gMidArch is an architecture-based framework in which middleware developers reuse a library of architectural elements to implement adaptive middleware systems in the Go programming language. The proposed extensions enrich the framework with several new communication components that implement different communication protocols. In the end, middleware developers can statically (development time) or dynamically (runtime) select the proper communication protocol to use.

In practice, the framework incorporates seven new transport components, namely UDP (User Datagram Protocol), TCP (Transmission Control Protocol) over TLS (Transport Layer Security), QUIC (Quick UDP Internet Connections), RPC (Remote Procedure Call), HTTP (HyperText Transfer Protocol) and HTTPS (Hypertext Transfer Protocol Secure) version 1.1 and HTTP version 2. The selection of these new components has different reasons: UDP as the fastest transport protocol, TCP over TLS to bring more security, QUIC as a promising new protocol, RPC as a classic transport method used to build client/server applications, HTTP/1.1 to enable a lot of new features, HTTPS as a security layer over the HTTP/1.1, and HTTP/2 to give more agility to HTTP features. These new protocols are made available as gMidArch components that can be used in implementing adaptive middleware systems.

The inclusion of new protocols gives more flexibility in selecting the middleware's transport mechanism. However, it is necessary to assess their impact on applications' performance and compare their performance against similar middleware systems widely adopted like gRPC [2] and RabbitMQ [3].

Before and during its execution, the middleware protocol configuration is beneficial because developers

URI: https://hdl.handle.net/10125/80261
978-0-9981331-5-7

can customise it to satisfy individual applications' demands built atop the middleware. For example, distributed applications implemented inside a single organisation can use a less secure (and more efficient) middleware protocol. Meanwhile, streaming applications may prefer using a middleware that provides UDP-based communication instead of TCP one.

The rest of this paper is organised into five more sections. Section 2 introduces basic concepts about gMidArch. Next, Section 3 describes the proposed extensions in detail. Section 4 presents the performance evaluation of the proposed elements. Section 5 discusses the related works. Finally, Section 6 presents the conclusions and future directions of this work.

2. gMidArch

Before presenting the proposed solution, it is worth describing the framework gMidArch[1] that is implemented in the Go programming language. gMidArch is an RPC-based framework that helps in the design, implementation, and safe execution of adaptive middleware systems. The main characteristics of the framework are shown in the following.

Middleware components gMidArch includes a set of reusable components specially designed for implementing middleware functionalities. According to their role in the middleware layers, these components are organised into four categories: infrastructure, distribution, common services, and specific services. The infrastructure layer is responsible for communication issues inside the middleware, e.g., send/receive data, establish TCP connections. The distribution layer implements transparencies that help to hide the distribution complexity. Finally, common and specific layers provide services for several applications (e.g., security) and particular domains (e.g., profile matching), respectively.

Software architecture gMidArch provides an agnostic Architecture Description Language (ADL) , named *m*ADL (middleware Architecture Description Language), used to describe the middleware software architecture. The architecture describes the components that make up the middleware, connected components, and adaptation strategy. This artefact is the only one defined by developers in implementing the middleware at development time. Meanwhile, it is also used to represent the middleware at runtime. Hence, developers can customise the middleware's transport protocol by setting the desired component in the architecture. At runtime, automatic adaptation mechanisms allow the transport protocol's replacement by considering the adaptation strategy defined. For example, a given transport protocol (TCP) can be replaced by another (UDP) while the middleware executes.

Next *m*ADL specification shows an RPC-based middleware software architecture on the client-side.

```
1  Configuration midfibonacciclient :=
2     Components
3        proxyn    : Namingproxy
4        proxyf    : Fibonacciproxy
5        requestor : Requestor
6        crh       : TCP-CRH
7
8     Connectors
9        t1 : Ntoone
10       t2 : Requestreply
11
12    Attachments
13       proxyn,t1,requestor
14       proxyf,t1,requestor
15       requestor,t2,crh
16
17    Adaptability
18       Evolutive
19 EndConf
```

This *m*ADL description defines four components used in the middleware (lines 2-6), how these components interact with each other (lines 8-10), how these components are connected (lines 12-15) and the configured adaptation mechanism (lines 17-18). At development time, to change from TCP to UDP, Line 6 should be the only one to be altered, i.e., from "chr:TCP-CRH" to "crh: UDP-CRH".

Lightweight formalisation Each component or connector available in gMidArch has associated a formal specification of its behaviour in the formal language CSP (Communicating Sequential Processes) [4]. This specification is used to check desired properties (e.g., deadlock freedom) of the software architecture before its execution and when an adaptation occurs. It is worth observing that CSP specifications are not visible to middleware developers as they are only used internally in the framework.

Runtime adaptation Middleware developers can explicitly configure, as defined in the *m*ADL description shown before (line 18), which adaptation mechanisms can be used at runtime. In practice, developers can change the behaviour of the middleware at runtime. gMidArch currently supports *evolutive*, *corrective* and *proactive* adaptation. Evolutive adaptation means that if a new version of a component used in the architecture becomes available, the new version replaces the old one. The corrective approach replaces a component if a bug (undesired behaviour) is detected at runtime. Finally,

a proactive strategy tries to identify a performance problem of a given component before it occurs.

Execution environment Architectures defined by middleware developers are deployed in this execution environment. The execution environment coordinates the execution of components that make up the middleware architecture and triggers adaptations according to the adaptability configured in the *m*ADL. This environment also implements an adaptation logic based on the MAPE-K (Monitor, Analyse, Plan, Execute - Knowledge) [5].

In gMidArch, the monitor continuously watches for predefined events, e.g., changes in the repository of components, errors during execution, performance degradation. Monitored data are then sent to the analyser that decides whether an adaptation is necessary or not, e.g., a new component is available. If an adaptation is needed, the analyser informs the planner. The planner creates an adaptation plan (set of actions) issued to the executor, who is responsible for executing the actions that lead to the change of the middleware.

3. gMidArch Extensions

This section presents the proposed extensions to gMidArch in details. Initially, it presents a general overview of the extensions and reasons behind the selected protocols. Next, it is explained how a new component is incorporated into gMidArch. Finally, each new component is presented with highlights of how they were implemented and work together.

3.1. General Overview

The proposed extensions concentrate on attaching new components to the infrastructure and distribution layers. Figure 1 presents existing and new components of gMidArch. At this point, it is worth observing that the inclusion of new components to the infrastructure layer required new ones to the distribution layer along with new messages. For example, the novel *crh-http* required a new proxy (*proxy-http*) and new message (*http-request*) for its proper functioning. Furthermore, as the new components implement communication protocols, it is natural that the support to a new protocol means adding client- and server-side elements. For instance, the support to QUIC requires new client (*crh-quic*) and server (*srh-quic*) components. The first one implements the communication logic on the client-side, and the second one realises actions required on the server-side of QUIC.

These new components encapsulate transport protocols widely adopted by distributed systems in different application domains: UDP, TCP+TLS, QUIC, RPC, HTTP, HTTPS and HTTP/2.

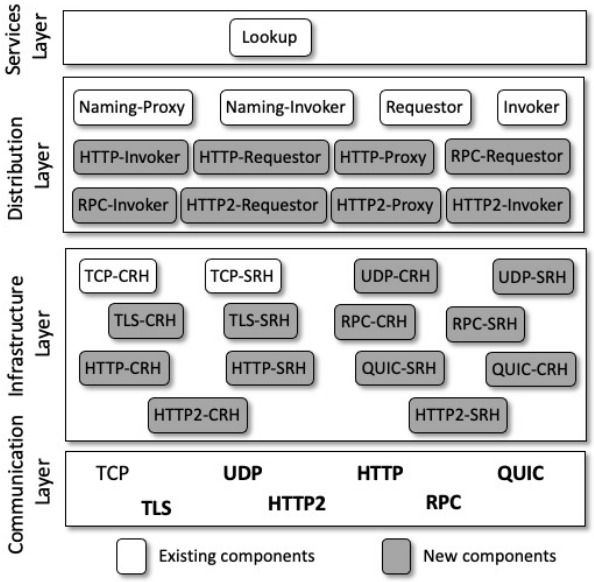

Figure 1. gMidArchcomponents

The new protocols have been chosen almost like an evolution from the most straightforward protocol to a more complex one. They were selected based on two criteria: client/server style and functionality. Concerning the first criteria, a suitable transport protocol should fit well with the client/server interaction model as it is usually used to implement RPC-based applications. Regarding the diversity of the new protocols, they should aggregate value to the middleware, e.g., security characteristics.

It is worth observing that although HTTP, HTTPS and HTTP/2 are application-layer protocols and not transport-layer protocols like TCP and UDP, they work as transport mechanisms from middleware developers' point of view. Hence, they bring new possibilities for developing distributed applications, such as communicating directly with browsers and generating web APIs and pages.

The components were designed according to the RPC architecture to address the complexity of accommodating different protocols having different behaviours in the same middleware. New components working atop traditional transport protocols (TCP and UDP) were fully implemented in the infrastructure layer. The implementation of the remaining components (e.g., HTTP, RPC) spreads out through the infrastructure and distribution layers. This strategy made it possible to implement the technological transparency of protocols

in the middleware.

Component model Each new component is associated to a given protocol and has been defined in compliance with the gMidArch component model. It means that every component has a type (e.g., UDP-CRH), a CSP specification, a state machine generated from the specification, and its implementation in Go. The type is used in checking type compatibility when an existing component needs to be replaced by a new one.

Whatever the protocol, its CSP specification describes how the component behaves while executes. The specification consists of a set of internal (not visible by other components) and external (to interact with other components) actions performed by the component. This specification is also used to generate a state machine (graph) executed by the execution environment (see Section 2).

Security Concerns From the point of view of the middleware, all transport protocols have similar behaviours, i.e., they send/receive messages whatever the actual way they work internally. Meanwhile, the security support of each protocol varies from one to another, which may raise security concerns, e.g., automatically change a secure protocol by an insecure one creates vulnerabilities to applications built atop the middleware. To address this concern, two classes of transport mechanisms have been taken into account: non-secure (UDP, TCP, RPC and HTTP) and secure (TCP+TLS, QUIC, HTTPS and HTTP2). All extensions with secure protocols use TLS 1.3, being similar in terms of security.

Hence, the developer should choose between a non-secure or secure protocol at development time. Choosing a non-secure protocol would enable the middleware to adapt to any protocol since alternating between non-secure and secure would not raise security issues. Conversely, only secure protocols can be used at runtime if a developer chooses a secure protocol at development time.

Alternatively, middleware developers can also avoid using particular secure protocols, e.g., QUIC may suffer from DoS attacks because it aims 0-RTT (Round-Trip Time), which the developer may not accept. In this case, the developer should explicitly define that QUIC cannot be used at any time.

3.2. UDP

UDP is the simplest and fastest transport protocol. As widely known, its speed comes from the simplicity of UDP due to lack of reliability, absence of error checking, correction and retransmissions. Developers adopt UDP in time-sensitive applications in which dropping packets is less harmful than waiting for retransmissions. There is a wide range of time-sensitive applications like games and others that use the Internet of Things sensors.

Despite the difference in how they work, TCP and UDP components have similar high-level behaviours in gMidArch. On the client-side, the UDP component waits for data to be sent, sends the data to the server-side, waits for a response and sends the result back to the caller. The TCP version has an additional step to create a connection before sending data. As clients and servers need to use the same transport protocol, the support to UDP includes two new components, namely UDP Client Request Handler (*UDP-CRH*) and UDP Server Request Handler (*UDP-SRH*).

At this point, it is worth observing that Client and Server Request Handlers are widely adopted middleware architectural patterns [6] whose responsibilities are to manage all aspects of communication on clients and servers, respectively. They open/close connections in connection-oriented protocols, work as the only touchpoint with socket APIs provided by operating systems and isolate other middleware components from the complexity of dealing with message transport issues.

3.3. TCP+TLS

As mentioned before, gMidArch already has TCP client- and server-side components. This new extension differs from the existing one as it includes a new layer implemented using Transport Layer Security (TLS). This cryptographic protocol improves security by preventing eavesdropping and tampering. Consequently, developers can implement a middleware with a secure transport mechanism commonly required in several business domains.

Similarly to the UDP extension, two new components were added: Client Request Handler (*TLS-CRH*) and Server Request Handler (*TLS-SRH*). The main difference from the TCP component is that they generate the TLS configuration based on TLS 1.3, as specified in RFC 8446 [7]. At this point, *TLS-CRH* and *TLS-SRH* need certificates to generate TLS configurations obtained through configuration files.

3.4. QUIC

QUIC (Quick UDP Internet Connections) [8] is a transport protocol over UDP designed by Google. The initial goal of QUIC was to improve traffic

on the Internet. Hence, it was designed over UDP as a general-purpose transport layer to reduce latency compared to TCP. The essential features of QUIC include reduced connection establishment time, improved congestion control, multiplexing without a head of line blocking and connection migration. Therefore, QUIC uses the simplicity and speed of UDP but implements reliability and security over it.

Despite using UDP, QUIC is similar to TCP+TLS extension as they use the same TLS 1.3 configuration. Unlike TCP+TLS, QUIC implementation needs to accommodate features like control over the stream, using an additional connection for the same client. Utilising QUIC, clients can establish a connection and use the same connection to communicate over one or more streams. It is worth noticing that this feature increases the speed of transfers by enabling multiple requests simultaneously.

Similarly to previous extensions, two new components were implemented using QUIC protocol based on RFC 9000 [9]: Client Request Handler (*QUIC-CRH*) and Server Request Handler (*QUIC-SRH*). Since Go has not a native implementation of the QUIC protocol, the package used in the new components is $quic - go$, a Go package built by the community that is not yet in the stable version but still the most functional QUIC package available for Go. The following piece of code is part of the QUIC Server Request Handler (*QUIC-CRH*) implementation and is needed to accept a connection, accept a stream and receive messages from the client.

```
1  func acceptRead(currCon int, c chan []byte){
2    ctx := context.Background()
3    // Accept connection
4    conn,err := LnSRHQuic.Accept(ctx)
5    if err != nil {...}
6    ConnsSRHQuic = append(ConnsSRHQuic, conn)
7    currCon++
8
9    // Accept Stream
10   stream,err := conn.AcceptStream(ctx)
11   StreamsQuic = append(StreamsQuic, stream)
12   if err != nil {...}
13
14   // Receive message (size)
15   size := make([]byte, MSG_SIZE, MSG_SIZE)
16   _, err = stream.Read(size)
17   [...]
```

Lines 3-7 implement the steps to accept a connection similar to TCP Server Request Handler (*TCP-SRH*). Lines 9-12 explicit the difference between QUIC and TCP, as QUIC needs this extra control over the communication through streams, almost like an additional connection. Lines 13-15 show how a message is sent, using the stream and not the connection.

3.5. RPC

RPC is a traditional middleware [10] based on adapting remote communications similar to a local procedure call (access transparency). In this case, the built-in Go RPC has been used in gMidArch as a request-reply transport mechanism.

Then, four new components have been implemented: Client Request Handler (*RPC-CRH*), Requestor (*RPC-Requestor*), Server Request Handler (*RPC-SRH*) and Invoker (*RPC-Invoker*). The Server and Client Request Handlers are the components responsible for using the Go RPC library.

3.6. HTTP/1.1

HTTP/1.1 brings various new possibilities to gMidArch as the middleware server-side can act as a web server and interact with browser clients. The HTTP/1.1 components partially implement the HTTP/1.1 specification and do not include TLS. This extension was implemented on top of package *net* of Go language. Since HTTP/1.1 is not a transport layer protocol, it demanded changes throughout the entire RPC chain of components. In practice, five new components were added to the framework: Client Proxy (*HTTP-Proxy*), Requestor (*HTTP-Requestor*), Client Request Handler (*HTTP-CRH*), Server Request Handler (*HTTP-SRH*) and Invoker (*HTTP-Invoker*).

The first component added to support HTTP/1.1 was a Client Proxy (*HTTP-Proxy*). Whatever the communication protocol, client Proxies act as interfaces to remote objects in the RPC architecture [6], and each remote object has a proxy. Unlike proxies that work atop TCP, *HTTP-Proxy* has to support the structure of HTTP messages, status codes and error identification. HTTP uses status codes to identify if a request is completed successfully or has any error and what kind of error. Hence, *HTTP-Proxy* components should understand these status codes.

The second added component, namely *HTTP-Requestor*, coordinates the middleware's actions on the client-side, i.e., it receives a request from the client proxy, marshalls it, and forwards the request to the client request handler [6]. After receiving the response from the server-side, it sends back the response to the client proxy. This behaviour is very similar to the existing *Requestor*. However, the *HTTP-Requestor* has a different serialisation as HTTP/1.1 messages are just human-readable strings with a specific field ordering format for HTTP/1.1 request messages and another format for HTTP/1.1 response messages in plain text.

The third added component, namely *HTTP-CRH*, is

responsible for receiving bytes and sending them to the correct destination. *HTTP-CRH*) differs from the TCP one (*TCP-CRH*) because the length of the message's body (*content-length*) used to announce the length of the next packet is specified in the middle of the HTTP message and not in the beginning.

Server Request Handler (*HTTP-SRH*), the fourth added component, receives bytes from clients and forwards them to the invoker. *HTTP-SRH* differs from *TCP-SRH* because it has to unmarshal the request message while reads the header to get the *content-length*. Then, it can read the whole message and send it to the invoker.

The last component added, *HTTP-Invoker*, is also a middleware pattern [6]. Invokers are responsible for coordinating the actions of the middleware on the server-side. The invoker receives a request message (sequence of bytes) from the server request handler, unmarshalls it, forwards it to the remote procedure, receives the object's response, marshalls the response and sends it back to the server request handler. *HTTP-Invoker* uses HTTP messages and HTTP marshall like the other components. Two additional data structures, *HTTP-Request* and *HTTP-Response*, were also created to manipulate and transfer information throughout the middleware.

Finally, it is worth mentioning that client proxies and invokers are part of the middleware, which could mean that they are generic. However, they are the only middleware components whose implementations are coupled to the application logic, i.e., they have implementations that are particular to each remote procedure/object being invoked. RPC-based commercial middleware systems usually provide a solution to avoid the need for application developers to mediate on client proxy and invoker implementations. These solutions use computational reflection at the programming language level or provide a compiler that automatically generates them.

The HTTP/1.1 invoker (*HTTP-Invoker*) is more straightforward than *TCP-Invoker* because the HTTP protocol already has the support of different programming languages such as Java, Nodejs and Go, e.g., facilities to create HTTP requests and responses. Then, it was possible to reuse this support to implement *HTTP-Invoker*. The implementation of *HTTP-Invoker* is shown in the following:

```
1  func I_Process(msg *messages.SAMessage,
2      info [] *interface{}) {
3    payload := msg.Payload.([]byte)
4    request := messages.HttpRequest{}
5    request.Unmarshal(payload)
6
7    response := messages.HttpResponse{}
8    impl.RequestListener(request, &response)
9
10   msgTemp := response.Marshal()
11   *msg = messages.SAMessage{Payload: msgTemp}
12 }
```

As mentioned in Section 3.1, the implementation of each gMidArch component consists of one or more internal actions that encapsulate the component's logic. In this code, the logic of *HTTP-Invoker* is implemented by function *I_Process* (Lines 1-2). Lines 3-5 describe how to get and unmarshal request messages. Line 8 calls the *RequestListener* function implemented by the developer containing the remote object's business logic. This point is the key difference that allows developers to decouple the business's code from the middleware. Despite decoupling the business logic, developers still have to change Line 8 to point to the particular business logic source code and its respective function. Lines 10-11 marshal response messages and forward them back to the Server Request Handler.

3.7. HTTPS

A new transport protocol was added to gMidArch to deal with "non-secure" pages, namely HTTPS. As mentioned in the TCP+TLS extension (Section 3.3), TLS 1.3 improves both the confidence about the identity of the server and that no one will be able to read messages exchanged between client and servers. Two new components were added to support HTTPS: *HTTPS-CRH* and *HTTPS-SRH*. This new HTTPS extension is similar to HTTP one and reuses HTTP components *HTTP-PROXY*, *HTTP-Requestor*, and *HTTP-Invoker*. However, it uses TLS as a security layer to transport messages.

3.8. HTTP/2

Unlike the HTTP extension that uses TCP with the *net* package of Go to implement HTTP/1.1, the HTTP/2 extension uses the *http* Go library that already supports HTTP/2. However, as the Go *http* library has its own message structure and behaviour, it was necessary to reimplement all RPC chain elements, i.e., proxy (*HTTP2-Proxy*, requestor (*HTTP2-Requestor*) invoker (*HTTP2-Invoker*, client request handler (*HTTP2-CRH*) and server request handler (*HTTP2-SRH*). The structure to handle messages was also implemented, namely *HTTP-Message*.

Although the HTTP/2 specification does not require encryption, many major implementations only support HTTP/2 over TLS. Hence, the proposed new components to support HTTP/2 also uses TLS 1.3.

The HTTP/2 component demultiplexes a URL to a function using routes configuration created by the developer and mapped through annotations. With this

extension, developers do not need to create a new invoker to access a new remote object/procedure (see Section 3.6).

Since the Go programming language does not support annotations, well-formatted comments in the code were used having the same purpose. The comments use keywords like Java Spring Controller because they are already widely disseminated and recognised and avoid creating a new meta-language. HTTP/2 components are then automatically scanned, looking for comments with annotation keywords at deployment time.

The use of annotations, reflection and a configuration file created by the developer are the inputs to start the webserver. The file lists desired routes to be created on the server. They are necessary to address the impossibility of making calls using Go language reflection without a pointer to the function being called. To illustrate how annotations are used in practice, the following code shows the HTTP/2 implementation of the Fibonacci invoker.

```
1  //@Controller
2  //@RequestMapping("/api")
3  package impl
4
5  import (...)
6
7  //@GetMapping(value="/fibo")
8  //@RequestParam(key="place")
9  func GetFibonacci(place int) string {
10  return strconv.Itoa(imp.Fibonacci{}.F(place))
11 }
```

Line 1 contains the annotation *@Controller* to identify the source code as one to be scanned. *Controller* has the business logic that will be demultiplexed within the invoker component. Line 2 has the optional annotation *@RequestMapping* that specifies the base URL of all functions within the source code.

Lines 7-11 implements the *GetFibonacci* function and its respective annotations. *@GetMapping* specifies that the HTTP method used is "GET" and it is mapped into route "/api/fibo", where "/api" comes from annotation *@RequestMapping*, and *@RequestParam* specifies the parameters used in that route. All annotations are scanned and mapped into routes to create a link from the invoker to the proper business logic. Any *Controller* inside the project implements the business logic, e.g., in a browser, the user can navigate to the server's address at URL /api/fibo?place=*value*, where value is the parameter to be passed to *GetFibonacci* function.

Go language does not allow the use of reflection to make function call only using its names. A pointer to the function is required to make the call. Due to this limitation and even using annotations, developers still need to associate the function's name with its respective pointer, as shown in Line 3 of the following code:

```
1 func GetFunction(name string) interface{} {
2   switch name {
3   case "GetFibonacci": return imp.GetFibonacci
4   default: return nil
5   }
6 }
```

4. Experimental Evaluation

This experimental evaluation has two main objectives: to analyse the performance of all transport components added to gMidArch; and to compare the performance of different flavours of gMidArch (using different transport protocols) and existing widely adopted commercial middleware systems (gRPC [2], Go RPC [11] and RabbitMQ [3]). For both objectives, the same client-server application was implemented atop different configurations of gMidArch, Go RPC, gRPC and RabbitMQ. Go RPC is the built-in implementation of RPC available in Go language, and gRPC is the Google open-source implementation of RPC. Finally, RabbitMQ is a widely adopted messaging system that also allows the development of RPC-based applications.

The metric used throughout the experiments was the *response time*, which is measured on the client-side and refers to the time elapsed since the client makes a request and receives a response. Having the focus on the middleware, the remote procedure (namely $fibonacci(N)$) invoked by the client recursively calculates a Fibonacci sequence number. In practice, each request passes through the client and server sides middleware before be executed remotely. While simple, the Fibonacci application is easy to deploy and uses all middleware components (similar to more complex applications), a fundamental requirement in the evaluation.

The environment used for the evaluation is a Docker Swarm cluster in which each component (client, server, naming service, and message broker) runs in a separate container. Each container runs over a Debian Buster image with a constraint of memory limits and reservations to 64MB RAM, sharing an Intel Core i7-9700T CPU @ 2.00GHz. In the experiments, clients make 10,000 requests to remote $fibonacci(N)$, where *N* was set to 2 (low processing demand) or 38 (high processing demand), in both cases with a small payload size. In practice, when *N=2*, the business time (time to calculate the Fibonacci) is lower than the middleware time (time the request/response passes inside the middleware). In the case *N=38*, the middleware time becomes lower than the business one.

4.1. Comparing gMidArch flavours

As mentioned before, the first objective of the evaluation was to compare the performance of different components just added to gMidArch. Hence, the client-server application executes atop gMidArch instances configured with different transport mechanisms (flavours), separated as secure (TCP+TLS, QUIC, HTTPS and HTTP2) and non-secure (UDP, TCP, RPC and HTTP) protocols.

Figure 2 depicts the quartiles and median of the results for $fibonacci(2)$, i.e., an application with low processing demand. According to these results[1], the flavours have the following order from the shortest *mean response time* to the longest one: HTTP (0.9907 ms), HTTPS (1.0503 ms), RPC (1.0523 ms), UDP (1.1027 ms), TCP (1.1303 ms), TCP+TLS (1.1570 ms), HTTP2 (1.5408 ms) and QUIC (1.6329 ms). By using the T-Test with a confidence of 95%, all samples have significant differences between them. Although it is possible to rank the protocols, a difference of 0.6422 ms between the fastest and the slowest protocols represents a *mean response time* 64% greater, and yet would be significant only in scenarios where there is a need for a high response rate. Otherwise, any of the protocols would be acceptable.

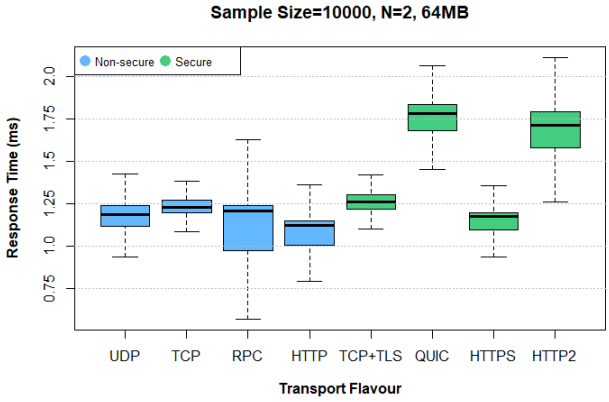

Figure 2. Performance of gMidArch transport flavours (N=2)

It is worth noticing that despite the HTTP2 and QUIC *mean response time* is higher than other flavours, both can multiplex several requests in a single connection, which is an advantage. This benefit, however, was not explored in the experiments due to the single sequential requests to $fibonacci(N)$. Also, as mentioned in Section 3.8, the HTTP2 extension uses reflection and annotations to decouple the business code from the middleware one, and this feature decreases the middleware processing time. As for the QUIC extension, a possible reason for the high *mean response time* is the use of a non-stable Go package implementation of the QUIC protocol.

Figure 3 shows the results for $fibonacci(38)$. An application with a high processing demand clarifies that performance cannot be the only criteria in selecting the best protocol when an adaptation is necessary. Using the T-Test with confidence of 95%, there are no significant differences between TCP and TCP+TLS and HTTPS and HTTP2. Hence, if a secure adaptation becomes necessary, HTTPS should be the only possibility as it is secure while performing similar to non-secure alternatives.

The results of this experiment show that different transport mechanisms used in gMidArch have very close performance. This fact indicates that middleware developers can also use further criteria (in addition to performance and security) while selecting the best middleware transport mechanism, e.g., design facility.

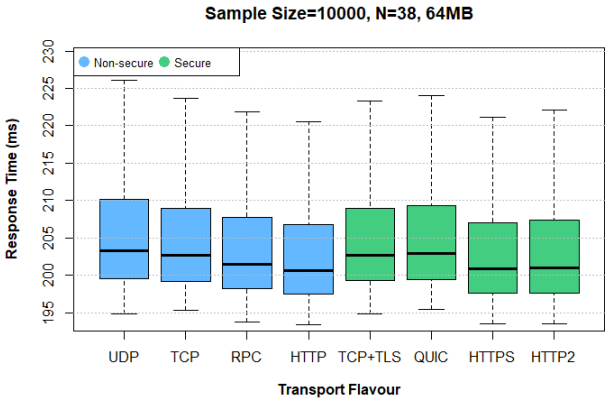

Figure 3. Performance of gMidArch transport flavours (N=38)

4.2. Comparing gMidArch versus commercial middleware systems

As mentioned before, the second objective of the evaluation was to compare the performance of gMidArch with existing commercial middleware systems, namely Go RPC, gRPC and RabbitMQ.

Commercial middleware systems were used without TLS and were compared with the non-secure extensions. HTTP that was the gMidArch flavour with the best performance (see Figure 2). Even with more variation than HTTPS, the HTTP extension has its third quartile smaller than the second quartile of any other protocol.

[1] Available at https://github.com/gfads/midarch/tree/master/evaluation

Hence, HTTP was chosen to be used in this comparison with commercial middleware.

Figure 4 shows the results of gMidArch atop HTTP (HTTP), Go RPC (E_RPC), gRPC (E_GRPC) and RabbitMQ (E_RMQ) for an application with low processing demand (*N=2*). As can be observed, gMidArch HTTP performs better than gRPC and RabbitMQ. Even though gRPC has a lower first quartile, HTTP *mean response time* is 0.9907 ms while gRPC is 1.1811 ms and RabbitMQ is 2.9421 ms. Meanwhile, a T-Test was applied to give confidence that the performance of gMidArch is also faster than the Go RPC that has a *mean response time* 2.5% higher (1.0161 ms).

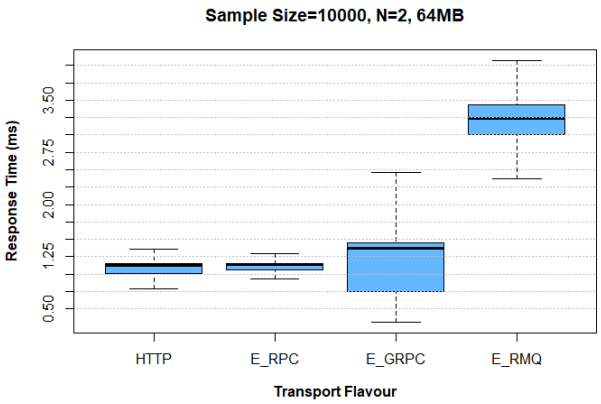

Figure 4. gMidArch versus commercial middleware systems (N=2)

Figure 5 depicts the results for $fibonacci(38)$, i.e., an application with high processing demand. In this case, a T-Test showed that gMidArch HTTP has the best performance compared with the commercial middleware systems.

5. Related Works

The design and implementation of middleware frameworks are not novel. The abstractions needed to implement a framework, especially a middleware one, comes with many challenges, e.g., the complexity of middleware functionalities, serialisation strategies, transport mechanisms. Quarterware [12], PolyORB [13] and Arcademis [14] stand as pioneer projects on this area. However, these classic middleware frameworks neither provide adaptive capabilities nor multiple transport protocols.

Man4Ware [15] is a middleware framework based on service-oriented architecture. It follows the idea of a modular middleware composed of several services integrated. Also, Man4Ware lets developers

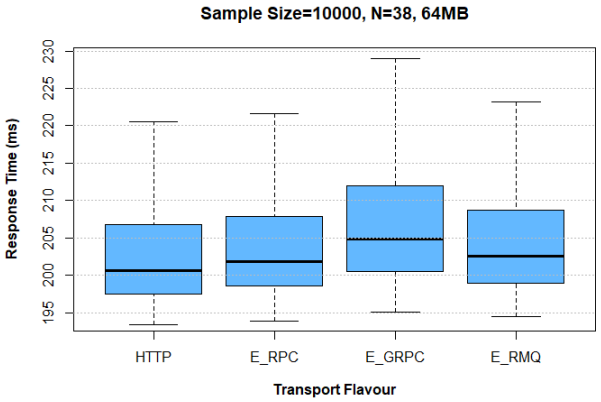

Figure 5. gMidArch versus commercial middleware systems (N=38)

implement the business code, similar to gMidArch, leaving the rest of the code for the framework. Man4Ware, however, does not allow selecting the transport protocol. Developers have neither control over how the middleware communicates nor can change the communication protocol at runtime.

Cilia [16] is an Autonomic Mediation Middleware that uses specific components to address different communication protocols and provide the possibility of adaptation like gMidArch. Nevertheless, Cilia only provides facilities like a knowledge base with runtime information and touchpoints to code. Developers need to implement the desired adaptation mechanisms. Cilia is not a general-purpose middleware like gMidArch, and it is focused on integrating cyber-physical systems into the management of smart industries.

Having the focus on supporting multiple transport protocols, CoServices [17] is a web service-based middleware framework designed to provide typical features of cooperative applications. It uses standard modules to deliver functionalities like session management and shared data management and the possibility of developing extra domain-specific modules. Although it uses web services for communication, CoServices may transport messages through UDP or HTTP. gMidArch, however, provides more additional protocols and adaptation facilities.

The matter of multiple transport protocols on adaptive middleware frameworks are discussed in [18]. It presents an adaptive middleware architecture for cyber-physical networks, where the importance of multiple transport protocols is justified to enforce quality of service constraints. Like gMidArch, the architecture introduced by Brinkschulte is based on the feedback control loop MAPE-K. However,

gMidArch uses lightweight formalisation to provide additional guarantees at development time and when adaptations occur at runtime.

Finally, even not being a framework, package RPC of Go [11] allows programmers to decide between two different transport protocols: HTTP and TCP. However, this selection is only possible at development time. Furthermore, support is limited to only two protocols.

6. Conclusion and Future Works

The unique contribution of this paper is to make available an adaptive middleware framework with a broad range of transport mechanisms. Having these flavours at hand, middleware developers can easily configure the gMidArch instance to use a particular flavour according to the application requirements. In some circumstances, it is also possible to dynamically replace the transport mechanism at runtime. Another contribution is related to the new gMidArch components' excellent performance compared to existing commercial middleware systems.

As an initial future work, new kinds of adaptation mechanisms should be introduced, especially regarding security. The developer will be able to opt for a secure evolutive adaptation, where only secure protocols would be used or will be able to choose which protocols the middleware should adopt. New protocols are also planned to be incorporated into gMidArch such as HTTP/3 and Bluetooth. As a new protocol that is still under development, HTTP/3 has excellent potential since its transport layer protocol is QUIC and has many features of HTTP/2. On the other hand, Bluetooth will open new perspectives of using gMidArch in IoT environments.

Finally, it is also necessary to enhance the experimental evaluation. The current version of gMidArch uses a Fibonacci application that enable to scale the processing time from the business logic but has a small payload size. It is planned to use different kinds of messages with distinct and more significant payloads. In this case, the size of messages will become a factor to be controlled in the experiments. It will also enable the comparison of multiple transport protocols from the perspective of message processing overhead and transmission delay.

References

[1] N. Rosa, D. Cavalcanti, G. Campos, and A. Silva, "Adaptive middleware in Go - a software architecture-based approach," *Journal of Internet Services and Applications*, 2020.

[2] "gRPC-Go package documentation." `https://github.com/grpc/grpc-go`. Accessed: 2021-04-23.

[3] "Go RabbitMQ package documentation." `https://www.rabbitmq.com`. Accessed: 2021-04-23.

[4] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, Inc., 1985.

[5] IBM, "An architectural blueprint for autonomic computing," tech. rep., IBM, June 2005.

[6] M. Volter, M. Kircher, and U. Zdun, *Remoting Patterns: Foundations of Enterprise, Internet and Real Time Distributed Object Middleware*. John Wiley & Sons Ltd, 2005.

[7] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3." RFC 8446, Aug. 2018.

[8] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, (New York, NY, USA), p. 183–196, Association for Computing Machinery, 2017.

[9] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport." RFC 9000, May 2021.

[10] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, p. 39–59, Feb. 1984.

[11] "Go RPC package documentation." `https://pkg.go.dev/net/rpc`. Accessed: 2021-04-23.

[12] A. Singhai, A. Sane, and R. H. Campbell, "Quarterware for middleware," in *Proceedings. 18th International Conference on Distributed Computing Systems (Cat. No.98CB36183)*, pp. 192–201, 1998.

[13] T. Vergnaud, J. Hugues, L. Pautet, and F. Kordon, "Polyorb: A schizophrenic middleware to build versatile reliable distributed applications," in *Reliable Software Technologies - Ada-Europe 2004* (A. Llamosí and A. Strohmeier, eds.), (Berlin, Heidelberg), pp. 106–119, Springer Berlin Heidelberg, 2004.

[14] F. Pereira, M. Valente, R. Bigonha, and M. Bigonha, "Arcademis: A framework for object-oriented communication middleware development," *Software: Practice and Experience*, vol. 36, pp. 495 – 512, 04 2006.

[15] J. Al-Jaroodi, N. Mohamed, and I. Jawhar, "A service-oriented middleware framework for manufacturing industry 4.0," *SIGBED Rev.*, vol. 15, p. 29–36, Nov. 2018.

[16] P. Lalanda, D. Morand, and S. Chollet, "Autonomic mediation middleware for smart manufacturing," *IEEE Internet Computing*, vol. 21, no. 1, pp. 32–39, 2017.

[17] W. Xie, Z. Li, and Y. Zhao, "Coservices: A web service based middleware framework for interactive cooperative applications," in *2013 Third International Conference on Intelligent System Design and Engineering Applications*, pp. 507–513, 2013.

[18] M. Brinkschulte, "Self-organizing middleware for cyber-physical networks," in *Proceedings of the 20th International Middleware Conference Doctoral Symposium*, Middleware '19, (New York, NY, USA), p. 14–16, Association for Computing Machinery, 2019.