UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DANIEL DA CRUZ BRANDÃO

Adaptação Dinâmica de Protocolos de Transporte em Sistemas de Middleware Baseados em RPC

2025

DANIEL DA CRUZ BRANDÃO

Adaptação Dinâmica de Protocolos de Transporte em Sistemas de Middleware Baseados em RPC

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Sistemas Distribuídos

Orientador (a): Nelson Souto Rosa

Recife

2025

.Catalogação de Publicação na Fonte. UFPE - Biblioteca Central

Brandão, Daniel da Cruz.
Adaptação dinâmica de protocolos de transporte em sistemas de middleware baseados em RPC / Daniel da Cruz Brandão. - Recife, 2025.
88f.: il.

Dissertação (Mestrado)- Universidade Federal de Pernambuco, Centro de Informática, Programa de Pós-Graduação em Ciência da Computação, 2025.
Orientação: Nelson Souto Rosa.

1. Middleware adaptativo; 2. Framework de middleware; 3. Protocolos de transporte. I. Rosa, Nelson Souto. II. Título.

UFPE-Biblioteca Central

Daniel da Cruz Brandão

“Adaptação Dinâmica de Protocolos de Transporte em Sistemas de Middleware Baseados em RPC”

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Redes de Computadores e Sistemas Distribuídos.

Aprovado em: 31/07/2025.

BANCA EXAMINADORA

_____

Prof. Dr. David Júnio Mota Cavalcanti
Centro de Informática / UFPE

_____

Prof. Dr. Fernando Antonio Aires Lins
Departamento de Computação / UFRPE

_____

Prof. Dr. Nelson Souto Rosa
Centro de Informática/UFPE
(**orientador**)

Dedico este trabalho à minha família. Aos meus pais, Ione e Ramiro, pelos sacrifícios que fizeram para que eu pudesse chegar até aqui. À minha esposa, Stela, pela paciência, pelo apoio e por me lembrar de sorrir nos momentos difíceis. Aos meus filhos, Miguel e Sofia, que são a razão do meu esforço e pela compreensão em momentos de ausência.

AGRADECIMENTOS

Primeiramente gostaria de agradecer ao meu orientador, Prof. Dr. Nelson Souto Rosa, por sua orientação, sua paciência, sua confiança, sua disponibilidade e por tudo que me ensinou ao longo dos anos dedicados a este trabalho. Sua orientação e suas críticas construtivas foram fundamentais para conclusão desta dissertação.

Agradeço aos membros da banca examinadora, Prof. Dr. Fernando Antônio Aires Lins e Prof. Dr. David Júnio Mota Cavalcanti, por terem aceitado o convite para avaliar esta dissertação e por dedicarem seu tempo e conhecimento para melhorar a qualidade deste trabalho com suas contribuições.

Gostaria de agradecer à equipe da Secretaria de Pós-Graduação por toda atenção e apoio ao longo destes anos.

Agradeço aos meus colegas do Centro de Informática que contribuíram mesmo que indiretamente com a minha dissertação.

Aos professores do Centro de Informática, agradeço por compartilharem seu conhecimento e experiência, que foram fundamentais para o meu aprendizado e desenvolvimento durante o mestrado.

Por fim, agradeço à Universidade Federal de Pernambuco (UFPE) por proporcionar um ambiente acadêmico rico e desafiador, que me permitiu crescer como pesquisador e profissional.

"A frase mais perigosa em qualquer língua é: 'Nós sempre fizemos as coisas desse jeito'." (Grace Hopper, 1976).

RESUMO

Um sistema distribuído adaptativo é capaz de ajustar dinamicamente (em tempo de execução) e autonomamente (sem intervenção humana) seu comportamento ou estrutura enquanto executa. Sistemas de middleware têm sido particularmente desenvolvidos para apoiar a implementação deste tipo de sistema. No entanto, middlewares existentes frequentemente não permitem a adaptação dinâmica dos protocolos de comunicação, fixando-os em tempo de desenvolvimento, não permitindo trocas dos protocolos e, como consequência, engessando a comunicação entre sistemas. Esta dissertação propõe um mecanismo de adaptação, denominado ***Protocol Adaptation* (pAdapt)**, contendo componentes de middleware que podem ser ajustados em tempo de execução e permitindo a troca do protocolo de comunicação de acordo com critérios implementados pelo desenvolvedor, como mudanças do contexto de execução da aplicação, e.g., o aumento da vulnerabilidade de segurança da rede leva à troca do protocolo de transporte da aplicação por um protocolo mais seguro. A solução proposta implementa componentes de oito protocolos de comunicação (UDP, TCP, TCP sobre TLS, RPC, QUIC, HTTP/1.1, HTTPS e HTTP/2) e introduz um mecanismo de adaptação síncrona. Este mecanismo, orquestrado pelo servidor e baseado no MAPE-K (*Monitor*, *Analyser*, *Planner*, *Executor and Knowledge*), garante a troca de protocolos em tempo de execução de forma coordenada entre o servidor e todos os clientes conectados, preservando o estado da comunicação e sem perda de mensagens. Ao mesmo tempo, estes novos componentes são incorporados a um *framework* de desenvolvimento de middleware adaptativo já existente, chamado gMidArch. Uma avaliação experimental foi realizada para comparar o desempenho da solução adaptativa proposta com middlewares comerciais como gRPC e RabbitMQ. A avaliação mostra que a sobrecarga do mecanismo de adaptação tem pouco impacto sobre o desempenho da aplicação. Ao mesmo tempo, os resultados indicam que o **pAdapt** com os novos componentes apresenta menor consumo de CPU no cliente em cenários de baixa carga e desempenho superior em transferências de arquivos grandes. Como principal contribuição, este trabalho permite que desenvolvedores de middleware selecionem e reconfigurem dinamicamente o protocolo de comunicação mais adequado para diferentes requisitos da aplicação, sem comprometer o desempenho das aplicações.

Palavras-chaves: Middleware Adaptativo, Framework de Middleware, Protocolos de Transporte.

ABSTRACT

An adaptive distributed system is capable of dynamically (at runtime) and autonomously (without human intervention) adjusting its behavior or structure while executing. Middleware systems have been particularly developed to support the implementation of this type of system. However, existing middleware often does not allow dynamic adaptation of communication protocols, fixing them at development time, not allowing protocol changes and, as a consequence, hindering communication between systems. This dissertation proposes an adaptation mechanism, named ***Protocol Adaptation*** (**pAdapt**), containing middleware components that can be adjusted at runtime and allowing the exchange of the communication protocol according to criteria implemented by the developer, such as changes in the application execution context, e.g., the increase in network security vulnerability leads to the exchange of the application transport protocol for a more secure protocol. The proposed solution implements components of eight communication protocols (UDP, TCP, TCP over TLS, RPC, QUIC, HTTP/1.1, HTTPS and HTTP/2) and introduces a synchronous adaptation mechanism. This mechanism, orchestrated by the server and based on MAPE-K (*Monitor*, *Analyser*, *Planner*, *Executor and Knowledge*), ensures the coordinated exchange of protocols at runtime between the server and all connected clients, preserving the communication state and without message loss. At the same time, these new components are incorporated into an existing adaptive middleware development framework called gMidArch. An experimental evaluation was performed to compare the performance of the proposed adaptive solution with commercial middleware such as gRPC and RabbitMQ. The evaluation shows that the overhead of the adaptation mechanism has little impact on the application performance. At the same time, the results indicate that **pAdapt** with the new components presents lower CPU consumption on the client in low-load scenarios and superior performance in large file transfers. As a main contribution, this work allows middleware developers to dynamically select and reconfigure the most suitable communication protocol for different application requirements, without compromising application performance.

Keywords: Adaptive Middleware, Middleware Framework, Transport Protocols.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ADL	*Architecture Description Language*
AMQP	*Advanced Message Queuing Protocol*
API	*Application Programming Interface*
AUM	*Adaptive Ubiquitous Middleware*
CoAP	*Constrained Application Protocol*
CSV	*Comma-Separated Values*
DDS	*Data Distribution Service*
gMidArch	*go adaptive Middleware aid by software Architecture*
Golang	*Go language*
gRPC	*gRPC Remote Procedure Calls*
HTTP	*HyperText Transfer Protocol*
HTTP/2	*HyperText Transfer Protocol version 2*
HTTP/3	*HyperText Transfer Protocol version 3*
HTTPS	*Hypertext Transfer Protocol Secure*
IoT	*Internet of Things*
JSON	*JavaScript Object Notation*
mADL	*middleware Architecture Description Language*
MAPE-K	*Monitor, Analyze, Plan, Execute and Knowledge*
MEx	*Middleware Extendify*
MQTT	*Message Queuing Telemetry Transport*
MTU	*Maximum Transmission Unit*
OSI	*Open Systems Interconnection*
pAdapt	*Protocol Adaptation*
pADL	*Python-based Architecture Description Language*
QoS	*Quality of Service*

QUIC	*Quick UDP Internet Connections*
REST	*Representational State Transfer*
RPC	*Remote Procedure Call*
RTT	*Round-Trip Time*
TCP	*Transmission Control Protocol*
TLS	*Transport Layer Security*
UDP	*User Datagram Protocol*

SUMÁRIO

1 INTRODUÇÃO

Este capítulo introduz o trabalho desenvolvido nesta dissertação, iniciando pela Seção 1.1 que estabelece o contexto e a motivação da pesquisa, e analisa como as tendências em sistemas distribuídos demandam flexibilidade na comunicação. Passando então para a Seção 1.2, que aborda o problema da rigidez nos sistemas de middleware convencionais, seguindo para a Seção 1.3, que expõe as soluções existentes e as lacunas identificadas na literatura. Com esta base, na Seção 1.4 são apresentados os objetivos desta dissertação, que visam abordar as lacunas identificadas e propor uma solução inovadora. A seção 1.5 é então dedicada a introduzir a solução proposta, o mecanismo ***Protocol Adaptation*** (**pAdapt**), e seus princípios. Finalmente, a Seção 1.6 descreve a estrutura do documento, detalhando a organização dos capítulos subsequentes.

1.1 CONTEXTO E MOTIVAÇÃO

Sistemas distribuídos são utilizados em diversos domínios, que vão desde aplicações Web (ABGAZ et al., 2023) (CHEN et al., 2018) e microsserviços (FOWLER, 2014) (SöYLEMEZ; TEKINERDOGAN; KOLUKıSA, 2022), até a *Internet of Things* (IoT) (KHEZEMI et al., 2024) e a computação de borda (*edge computing*) (SHI et al., 2016) e operam em ambientes cada vez mais heterogêneos e dinâmicos. A heterogeneidade é vista na diversidade de regras de negócio, de dispositivos, sistemas operacionais, frameworks e principalmente nos protocolos de comunicação. Já a dinamicidade é percebida na variação da carga de trabalho, na disponibilidade de recursos computacionais e nas condições instáveis da rede. Sistemas adaptativos são temas recorrentes e já estabelecidos (DA; DALMAU; ROOSE, 2011) (WEYNS, 2021), mas, independentemente do suporte oferecido aos desenvolvedores de sistemas, as questões de heterogeneidade e adaptabilidade geralmente são tratadas por uma camada de software intermediária, o middleware.

Um Middleware adaptativo (ROSA et al., 2020) gerencia este dinamismo, reconfigurando-se automaticamente para atender a requisitos não funcionais, como desempenho, segurança e confiabilidade, ou mesmo requisitos funcionais, como uma opção para poupar energia e manter o dispositivo funcionando por mais tempo. Contudo, apesar dos avanços na área, uma limitação crítica persiste: não foram identificados sistemas de middleware capazes de ajustar

dinamicamente o protocolo de comunicação. Ou seja, por padrão, o protocolo usado pelo middleware é fixado em tempo de desenvolvimento e depois não pode ser ajustado durante a execução.

O dinamismo percebido nas aplicações contrasta com a natureza estática das arquiteturas de middleware. Em *clusters* de microsserviços, por exemplo, a comunicação entre serviços internos pode exigir um middleware de alto desempenho, como o *gRPC Remote Procedure Calls* (gRPC) (GRPC, 2025). Já a comunicação com clientes externos, como navegadores Web e aplicações móveis, demanda pela construção de uma *Application Programming Interface* (API) com o uso de protocolos amplamente adotados e interoperáveis, como *HyperText Transfer Protocol* (HTTP) (FIELDING; NOTTINGHAM; RESCHKE, 2022) com *Representational State Transfer* (REST) (FIELDING, 2000) e *JavaScript Object Notation* (JSON). Por outro lado, para aplicações que toleram perdas de pacotes, como telemetria em tempo real, protocolos leves baseados em *User Datagram Protocol* (UDP) são mais adequados. Além disso, em cenários onde os serviços estão hospedados em nuvens pagas, também é necessário considerar o custo de operação, onde o uso de protocolos que exijam mais recursos computacionais, como memória, CPU e tráfego de rede, pode impactar diretamente no valor a ser pago.

Um middleware que utiliza um único protocolo de transporte obriga as equipes de desenvolvimento a fazerem concessões de arquitetura. Ou seja, os desenvolvedores devem optar pela adoção do protocolo pré-estabelecido, ou então gerenciar múltiplos conjuntos de tecnologia de comunicação, como diferentes middleware, diferentes protocolos de comunicação e encapsulamento e diferentes formas de implementação para o envio de uma mesma mensagem, o que eleva a complexidade de desenvolvimento, monitoramento e manutenção.

A proliferação da Internet das Coisas com a explosão de dispositivos IoT introduz outro conjunto de desafios. Tais dispositivos operam sob restrições severas de energia, poder de processamento e largura de banda de rede. Nesses cenários, a escolha do protocolo é crítica.

Um destes cenários de IoT pode ser exemplificado com o uso de protocolos leves como o UDP, ideal para o envio de pequenas leituras de sensores, onde o *overhead* de uma conexão *Transmission Control Protocol* (TCP) é proibitivo. Em cenários onde a confiabilidade e a segurança são necessárias, é indicado o uso de protocolos que garantam a entrega das mensagens, como TCP, e também de um protocolo que melhore a segurança, como o *Transport Layer Security* (TLS) (RESCORLA, 2018). Um cenário, que se enquadra nestas condições, é o envio de atualizações de firmware para dispositivos IoT, onde a perda de pacotes pode ocasionar falhas no dispositivo e um firmware comprometido pode ser um risco de segurança.

Um outro cenário é onde um dispositivo pode precisar alternar entre um modo de baixa energia (usando UDP) e um modo de alta confiabilidade (usando TCP sobre TLS), dependendo da tarefa a ser executada ou das condições da rede. A ausência de um mecanismo de adaptação de protocolo nesses ambientes resulta em sistemas que, ou desperdiçam recursos, ou falham em garantir a confiabilidade quando ela é mais necessária.

Esta dissertação é, portanto, motivada pela necessidade de evoluir o middleware para além de sua concepção de protocolos de comunicação estáticos. A proposta é tratar o protocolo de transporte não como um detalhe de implementação fixo, mas como uma dimensão estratégica e dinâmica do sistema, que pode e deve ser adaptada em tempo de execução para responder às demandas da aplicação e do ambiente, otimizando assim, seu desempenho, segurança, eficiência e custo.

1.2 PROBLEMA

O problema central abordado nesta dissertação é a rigidez da camada de transporte dos sistemas de middleware convencionais. Esta limitação compromete não somente a eficiência, a flexibilidade e a capacidade de evolução de aplicações distribuídas, mas também os custos envolvidos na manutenção e execução destas aplicações. A prática de implementar o middleware com um único protocolo de comunicação em tempo de desenvolvimento, tornando-o imutável em tempo de execução, gera uma série de desafios técnicos e operacionais.

O primeiro destes desafios é a inflexibilidade em tempo de execução. A escolha de um protocolo durante a fase de projeto é baseada em suposições sobre o ambiente operacional e o perfil de uso da aplicação. Na prática, esses fatores são dinâmicos e podem mudar ao longo do tempo. Por exemplo, a carga de trabalho pode variar sazonalmente, as condições da rede podem se degradar, novos requisitos de segurança podem ser impostos (MITRE, 2013) e novas tecnologias de protocolo mais eficientes podem emergir (BISHOP, 2022). Um sistema com um protocolo fixo é incapaz de reagir a essas mudanças, ficando restrito a operar em um estado inseguro ou operando de forma ineficiente, até que um ajuste seja realizado manualmente para atualizar o sistema.

Outro desafio é a complexidade e o custo de manutenção. A falta de um middleware adaptativo e multiprotocolo nativo impõe uma carga de complexidade sobre as equipes de desenvolvimento. Para contornar esta rigidez, os desenvolvedores são forçados a adotar soluções alternativas, como gerenciar múltiplas pilhas de middleware distintos em paralelo ou construir

gateways de tradução de protocolo. Essas soluções são frágeis, introduzem latência adicional, aumentam a superfície de ataque e tornam o sistema como um todo mais difícil de entender, depurar e manter.

Por outro lado, para evitar a complexidade, o custo de manutenção e os problemas mencionados acima, é possível que os desenvolvedores optem pela utilização de um protocolo genérico para todas as aplicações e cenários. No entanto, a utilização de um protocolo inadequado para uma determinada tarefa leva a um desperdício de recursos. Por exemplo, forçar o uso de TCP para uma aplicação de telemetria em tempo real, que poderia tolerar pequenas perdas de pacotes, introduz *overhead* de latência com *handshakes* e retransmissões. Inversamente, usar UDP para uma transação financeira sem uma camada de confiabilidade robusta implementada na aplicação é inviável. Nestas situações, a diferença entre o protocolo ideal e o utilizado resulta em maior consumo de CPU, maior uso de memória e possivelmente um aumento de custo de operação, especialmente em ambientes de nuvem onde o uso de recursos é cobrado.

Um outro desafio é a adoção de um novo protocolo de transporte. A introdução de um novo protocolo exigiria a substituição completa do middleware, em vez de uma atualização incremental. Isso inibe a capacidade do sistema de evoluir e se beneficiar dos avanços contínuos na tecnologia de comunicação.

1.3 SOLUÇÕES EXISTENTES E LACUNAS IDENTIFICADAS

As soluções existentes frequentemente apresentam uma ou mais limitações. Uma primeira limitação encontrada nas soluções atuais é o escopo de domínio específico. Muitos frameworks adaptativos, como Cilia (Lalanda; Morand; Chollet, 2017), AUM (PRADEEP; KRISHNAMOORTHY; VASILAKOS, 2021) e MEx (CAVALCANTI; ROSA, 2024), são projetados com foco em domínios específicos, como IoT ou redes ciberfísicas. Suas arquiteturas são otimizadas para os requisitos desses domínios (e.g., protocolos como MQTT e CoAP, e baixo consumo de energia), o que limita sua aplicabilidade em cenários de propósito geral, como aplicações Web de alto desempenho.

Outra limitação é a adaptação inexistente ou limitada. Outras soluções oferecem suporte limitado a múltiplos protocolos, mas carecem de mecanismos para adaptação em tempo de execução entre eles. O pacote RPC nativo do Go (RPC-GO, 2025), por exemplo, permite escolher entre TCP e HTTP, mas essa escolha é estática. O framework Gorilla (GORILLA, 2025) também se concentra em HTTP e WebSocket sem prover meios para reconfiguração

dinâmica.

Portanto, fica evidente a lacuna nas soluções existentes, a falta de um middleware adaptativo de propósito geral que suporte a adaptação de protocolos de comunicação em tempo de execução. Com isso, a pergunta central que esta dissertação busca responder: *Como superar a rigidez da camada de transporte dos sistemas de middleware convencionais?*

1.4 OBJETIVOS

O objetivo geral desta dissertação é projetar, implementar e avaliar um middleware adaptativo, de propósito geral, que supere a rigidez da camada de transporte, através da adaptação (troca) de forma autonômica e dinâmica de protocolos de comunicação em tempo de execução, sem a perda de informações e de forma síncrona entre clientes e servidores.

Para alcançar o objetivo geral, os seguintes objetivos específicos foram definidos:

1. Realizar um levantamento bibliográfico e análise comparativa dos principais frameworks e middleware adaptativos: Realizar um estudo abrangente sobre os middleware adaptativos existentes, com foco em suas capacidades de adaptação e nos seus protocolos de comunicação, identificando lacunas e limitações que justificam a necessidade de uma nova abordagem;

2. Projetar e implementar diferentes protocolos de comunicação: Projetar e implementar componentes que incorporem um conjunto diversificado de protocolos de comunicação (UDP, TCP, TCP sobre TLS (RESCORLA, 2018), *Remote Procedure Call* (RPC) (THURLOW, 2009), *Quick UDP Internet Connections* (QUIC) (IYENGAR; THOMSON, 2021), HTTP/1.1 (FIELDING; NOTTINGHAM; RESCHKE, 2022), *Hypertext Transfer Protocol Secure* (HTTPS) (FIELDING; NOTTINGHAM; RESCHKE, 2022) e *HyperText Transfer Protocol version 2* (HTTP/2) (THOMSON; BENFIELD, 2022)) como componentes modulares e intercambiáveis;

3. Projetar e implementar um mecanismo de adaptação de protocolos de comunicação: Projetar e implementar um mecanismo de adaptação em tempo de execução, capaz de orquestrar a migração síncrona de protocolo entre o servidor e seus clientes de forma *stateful*, proporcionando consistência ao sistema e integridade dos dados durante a tran-

sição, permitindo assim que o servidor e seus clientes sejam adaptados simultaneamente; e

4. Avaliar o desempenho e a eficácia da solução: Avaliar experimentalmente o desempenho da solução proposta, quantificando o impacto (*overhead*) do mecanismo de adaptação e comparando o seu desempenho com outros middleware comerciais sob diferentes condições de trabalho, a fim de validar sua viabilidade e competitividade.

1.5 SOLUÇÃO PROPOSTA

A solução proposta nesta dissertação para resolver os desafios mencionados é um mecanismo de adaptação, chamado ***Protocol Adaptation*** (**pAdapt**), capaz de realizar trocas do protocolo de comunicação enquanto a aplicação executa e sem a parada da mesma. O **pAdapt** consiste de um conjunto de novos componentes em tempo de desenvolvimento, e um novo mecanismo de adaptação com sincronização entre clientes e servidores. Os novos componentes em tempo de desenvolvimento adicionados encapsulam regras de comunicação de diversos protocolos de comunicação acrescentados ao middleware. Já o novo mecanismo de adaptação, com sincronização entre clientes e servidores, age como um orquestrador que gerencia a troca dos componentes (protocolos) dinamicamente, proporcionando uma adaptação consistente e sem perda de estado ou mensagens em trânsito. Por fim, o **pAdapt** é incorporado como uma extensão a um framework de middleware adaptativo já existente, chamado gMidArch (ROSA et al., 2020).

O mecanismo **pAdapt** foi projetado considerando dois princípios básicos:

- **Suporte Nativo e Extensível a Múltiplos Protocolos de Comunicação**: O **pAdapt** implementa um conjunto diversificado de componentes de comunicação, cada um encapsulando um protocolo de transporte distinto, UDP, TCP, TCP sobre TLS, RPC, QUIC, HTTP/1.1, HTTPS e HTTP/2. A arquitetura de componentes do gMidArch permite que esses protocolos sejam tratados como "peças" intercambiáveis e, mais importante, que novos protocolos sejam adicionados no futuro com esforço mínimo, melhorando a extensibilidade da solução; e

- **Mecanismo de Adaptação Síncrona e Stateful**: O orquestrador coordena a mudança de protocolos entre o servidor e todos os seus clientes conectados. Para isso, o

orquestrador usa o MAPE-K (IBM, 2005), i.e., monitora o sistema, analisa as informações coletadas do monitoramento, planeja e executa as adaptações conforme necessário. O foco deste novo mecanismo está em como realizar a adaptação, e não no motivo da adaptação.

1.6 ESTRUTURA DO DOCUMENTO

O restante da dissertação é organizado em mais cinco capítulos:

- **Capítulo 2**: Este capítulo introduz os conceitos básicos necessários para o entendimento deste trabalho;
- **Capítulo 3**: Este capítulo apresenta os novos componentes dos protocolos de comunicação, o novo mecanismo de adaptação de protocolos em tempo de execução, denominado **pAdapt**, e a integração deles com o gMidArch;
- **Capítulo 4**: Este capítulo apresenta a avaliação experimental da solução proposta;
- **Capítulo 5**: Este capítulo apresenta uma análise comparativa com os trabalhos existentes; e
- **Capítulo 6**: Neste último capítulo são apresentadas as conclusões e os potenciais trabalhos futuros desta dissertação.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os conceitos necessários para a compreensão desta dissertação. Primeiramente serão abordados os protocolos de comunicação na Seção 2.1. Na sequência, a Seção 2.2 apresenta os conceitos de middleware adaptativo e responde as questões básicas de um sistema adaptativo. Em seguida, na Seção 2.3 é apresentado o MAPE-K. A Seção 2.4 detalha o framework de middleware gMidArch (ROSA et al., 2020), implementado utilizando a linguagem de programação Go. Por fim, a Seção 2.5 apresenta as considerações finais deste capítulo.

2.1 PROTOCOLOS DE COMUNICAÇÃO

Os protocolos de comunicação de mensagens em sistemas distribuídos garantem que os dados transmitidos entre os componentes da aplicação cheguem corretamente ao destino, mesmo diante de falhas ou congestionamentos.

Entre os protocolos mais comuns utilizados em sistemas distribuídos estão UDP, TCP, TCP com TLS, RPC, QUIC, HTTP, HTTPS, HTTP/2, gRPC e *Advanced Message Queuing Protocol* (AMQP) (GODFREY; INGHAM; SCHLOMING, 2012). A seguir, são detalhadas as principais características destes protocolos de comunicação:

- **UDP**: O *User Datagram Protocol* (UDP) é um dos protocolos de comunicação mais simples e de menor latência. Seu bom desempenho decorre da ausência de mecanismos de controle de confiabilidade, como verificação de integridade, correção de erros ou retransmissão de pacotes perdidos. O protocolo não estabelece conexão, transmitindo dados sem garantia de entrega ou ordenação. Essa característica o torna particularmente adequado para aplicações sensíveis ao tempo, como transmissões multimídia e sistemas em tempo real, nas quais a perda ocasional de pacotes é tolerável, e preferível, a atrasos decorrentes de retransmissões;
- **TCP**: O *Transmission Control Protocol* (TCP) é um protocolo orientado à conexão, que oferece comunicação confiável entre duas extremidades. Ele implementa controle de fluxo, controle de congestionamento e garantia de entrega de pacotes na ordem correta. Por essa razão, o protocolo TCP é amplamente adotado em aplicações que exigem confiabilidade;

- **TCP com TLS**: O *Transport Layer Security* (TLS) (RESCORLA, 2018) é um protocolo criptográfico que opera sobre o TCP, adicionando uma camada de segurança à comunicação. Ele protege contra escutas e alterações indevidas nos dados transmitidos, sendo fundamental em ambientes que exigem confidencialidade e integridade. Por simplicidade, neste trabalho, o termo TLS será utilizado para se referir ao uso do protocolo TLS sobre o protocolo TCP;

- **RPC**: O *Remote Procedure Call* (RPC) (THURLOW, 2009) não é, em si, um protocolo de transporte, mas sim um paradigma de comunicação que permite a um programa executar uma função que esteja em outro de endereço, e.g., em outra máquina, como se fosse uma chamada local. O RPC abstrai a complexidade da comunicação em rede, permitindo que os desenvolvedores se concentrem na lógica de negócio do sistema;

- **QUIC**: O *Quick UDP Internet Connections* (QUIC) (IYENGAR; THOMSON, 2021) é um protocolo de transporte desenvolvido pelo Google que combina as vantagens do UDP com mecanismos de segurança e confiabilidade. Ele incorpora criptografia por padrão, multiplexação eficiente de conexões e recuperação rápida de perdas;

- **HTTP**: O *HyperText Transfer Protocol* (HTTP) (FIELDING; NOTTINGHAM; RESCHKE, 2022) é um protocolo de aplicação baseado em texto, utilizado amplamente na comunicação Web. Ele define regras para requisições e respostas entre clientes e servidores. Sua simplicidade e padronização o tornam adequado para aplicações *Representational State Transfer* (REST) e microsserviços (FOWLER, 2014). HTTP é o principal protocolo utilizado para a transferência de dados na Web, permitindo a comunicação entre navegadores e servidores. O HTTP opera sobre o TCP, garantindo a entrega confiável de mensagens;

- **HTTPS**: O *Hypertext Transfer Protocol Secure* (HTTPS) (FIELDING; NOTTINGHAM; RESCHKE, 2022) é uma versão mais segura do HTTP, sendo essencialmente HTTP operando sobre TLS. Ele garante a confidencialidade e autenticidade das comunicações, sendo o padrão adotado para transações seguras na Web;

- **HTTP/2**: A Versão 2 do HTTP, o *Hypertext Transfer Protocol Secure version 2* (HTTP/2) (THOMSON; BENFIELD, 2022), é uma evolução do protocolo HTTP, introduzindo melhorias como multiplexação de *streams*, compressão de cabeçalhos e uso mais eficiente de conexões TCP. Isso resulta em menor latência e melhor desempenho;

- **gRPC**: O *gRPC Remote Procedure Calls* (gRPC) (GRPC, 2025) é um framework de chamadas de procedimento remoto baseado em HTTP/2 e *Protocol Buffers*. Ele permite comunicação eficiente entre serviços, com suporte a *streaming*, múltiplas plataformas e linguagens de programação; e
- **AMQP**: O *Advanced Message Queuing Protocol* (AMQP) (GODFREY; INGHAM; SCHLOMING, 2012) é um protocolo que opera como um intermediário de mensagens entre sistemas distribuídos, promovendo desacoplamento e escalabilidade, também é o protocolo utilizado como base para o RabbitMQ (RabbitMQ, 2025).

A Tabela 1 resume as principais características dos protocolos de comunicação abordados, destacando aspectos como confiabilidade e segurança, assim como adicionando os principais casos de uso de cada protocolo.

Tabela 1 – Características dos Protocolos de Comunicação

Protocolo	**Confiabilidade**	**Segurança**	**Principal Caso de Uso**
UDP	Baixa	Não	Multimídia, tempo real (baixa latência)
TCP	Alta	Não	Aplicações que exigem confiabilidade
TLS	Alta	Alta	Comunicação segura (confidencialidade)
RPC	Dependente da implementação	Dependente da implementação	Chamadas de funções remotas
QUIC	Alta	Alta	Web moderna (substituto TCP+TLS)
HTTP	Alta	Não	Web, APIs REST, microsserviços
HTTPS	Alta	Alta	Web segura
HTTP/2	Alta	Alta	Web de alto desempenho (multiplexação)
gRPC	Alta	Alta	APIs, comunicação entre microsserviços
AMQP	Alta	Não	Mensageria, sistemas desacoplados

Fonte: Elaborado pelo autor (2025)

Como pode ser observado na Tabela 1, cada protocolo possui diferentes características, que os tornam mais adequados para determinados cenários. Ou seja, uma aplicação pode se beneficiar de um protocolo em determinado cenário, e.g., UDP para transmissões multimídia, mas em outro cenário, e.g., chamadas à APIs, o HTTP/2 pode ser mais apropriado.

2.2 MIDDLEWARE ADAPTATIVO

Para lidar com a complexidade dos sistemas distribuídos, os middlewares adaptativos surgem como uma solução eficaz. Um middleware adaptativo é uma camada de software intermediária que trata a heterogeneidade e adaptabilidade do sistema. Ele gerencia o dinamismo do ambiente, reconfigurando-se automaticamente para atender a diferentes requisitos. Isso inclui a capacidade de adaptar protocolos de comunicação, foco deste trabalho.

Para gerenciar a complexidade ao se desenvolver um middleware adaptativo, são enfrentados os mesmos desafios de se desenvolver um sistema adaptativo (SALEHIE; TAHVILDARI, 2009). Estes desafios nos levam a questões básicas de adaptação, como: o porquê de adaptar, onde adaptar, o que adaptar, como adaptar e quando adaptar.

Considerando que um middleware é uma camada entre diferentes sistemas distribuídos, a adaptação dos protocolos de comunicação se torna ainda mais impactante, uma vez que, se ocorrer algum problema, ambos os sistemas se comunicando podem ser afetados.

Sendo assim, respondendo as questões levantadas, começamos com o porquê de adaptar protocolos de comunicação. A adaptação de protocolos de comunicação é necessária, como esclarecido na Seção 1.2, para lidar com mudanças no ambiente, seja nas condições de rede, nos requisitos de desempenho, na segurança e ou mesmo na confiabilidade.

A segunda questão é onde adaptar. A adaptação de protocolos de comunicação deve ocorrer somente nas camadas do middleware que tratam da comunicação, i.e., onde os protocolos de comunicação são implementados e gerenciados. Isso permite que o middleware adapte os protocolos utilizados, sem impactar as camadas superiores, nem a aplicação que utiliza o middleware.

A terceira questão é o que adaptar. Para efetuar a adaptação dos protocolos de comunicação, os componentes responsáveis pela comunicação devem ser substituídos por outros componentes que implementem o novo protocolo desejado. Com isso é necessário que os componentes de comunicação sejam desenvolvidos de forma que sejam compatíveis entre si, ou seja, que possuam a mesma interface de comunicação.

A quarta questão é como adaptar. Esta questão é o foco deste trabalho, projetar e implementar um middleware adaptativo que permita a adaptação de protocolos de comunicação em tempo de execução, sem a necessidade de reiniciar o sistema ou interromper o serviço e sem perda de informações, ainda de forma síncrona entre clientes e servidores. Para isso, o middleware deve seguir um fluxo de adaptação que permita a troca dos componentes de

comunicação de forma sincronizada.

A última questão é quando adaptar. A adaptação dos protocolos de comunicação deve ocorrer quando houver uma mudança nas condições do ambiente ou nos requisitos da aplicação que justifique a troca do protocolo utilizado. Este é um ponto onde o desenvolvedor do middleware deve definir e implementar os critérios que serão utilizados para disparar a adaptação. A solução proposta neste trabalho não aborda a definição destes critérios, mas sim o mecanismo de adaptação em si, deixando a cargo do desenvolvedor do middleware a definição de quando adaptar.

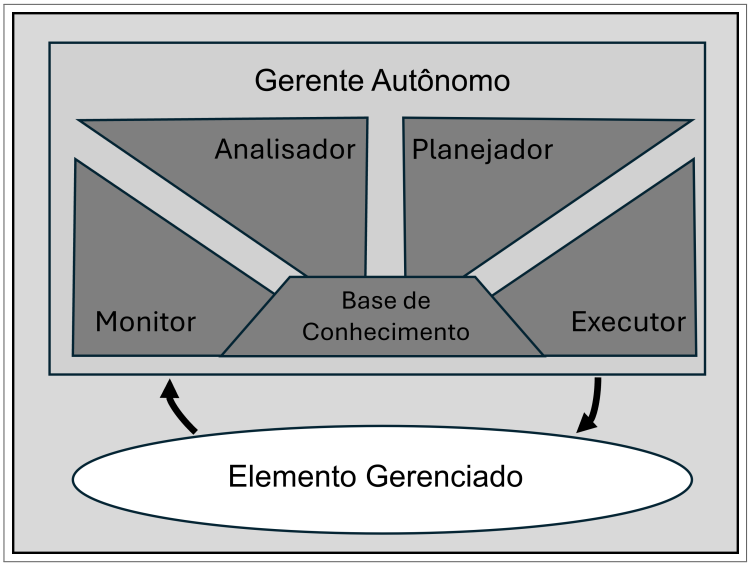
2.3 FEEDBACK LOOP

Uma abordagem tradicional para estruturar sistemas adaptativos é utilizar o conceito de *feedback loop* para gerenciar a adaptação destes sistemas (KEPHART; CHESS, 2003), especialmente em ambientes distribuídos e dinâmicos. Trata-se de um mecanismo de gerenciamento baseado na coleta contínua de dados do sistema, análise do seu comportamento atual e tomada de decisões para ajustes ou reconfigurações em tempo de execução.

O modelo de *feedback loop* MAPE-K (KEPHART; CHESS, 2003) (IBM, 2005) é comumente utilizado (BRINKSCHULTE, 2019) (ROSA; CAMPOS; CAVALCANTI, 2019) (CAVALCANTI; ROSA, 2024). A arquitetura do MAPE-K pode ser vista na Figura 1. Nela é possível observar o *Gerente Autônomo* e o *Elemento Gerenciado*. O *Elemento Gerenciado* é o objeto que se pretende adaptar, ele que é monitorado para verificar se há necessidade de adaptação ou não. O *Gerente Autônomo* coordena todo o ciclo de adaptação. O *Gerente Autônomo* é dividido em quatro componentes principais e unido por uma base de conhecimento:

- **Monitor:** coleta informações do ambiente e do próprio sistema, como desempenho da rede, disponibilidade de recursos e estado dos componentes;
- **Analisador:** interpreta os dados monitorados para identificar padrões, anomalias ou a necessidade de adaptação;
- **Planejador:** define ações corretivas ou estratégias de reconfiguração para alcançar os objetivos definidos;
- **Executor:** aplica as mudanças planejadas no sistema, como trocar componentes, migrar serviços ou ajustar parâmetros; e

Figura 1 – MAPE-K



Fonte: KEPHART; CHESS (2003)

- **Base de Conhecimento:** atua como um repositório central de informações históricas e regras, que podem ser usadas por todos os componentes do MAPE-K.

A adoção de *feedback loops* permite que os middleware sejam mais resilientes, eficientes e autônomos, adaptando-se dinamicamente a mudanças nas condições de operação ou nos requisitos das aplicações (KEPHART; CHESS, 2003). Essa abordagem é particularmente relevante em ambientes de sistemas distribuídos, e.g., computação em nuvem, Internet das Coisas (IoT) e sistemas orientados a serviços.

2.4 GMIDARCH FRAMEWORK

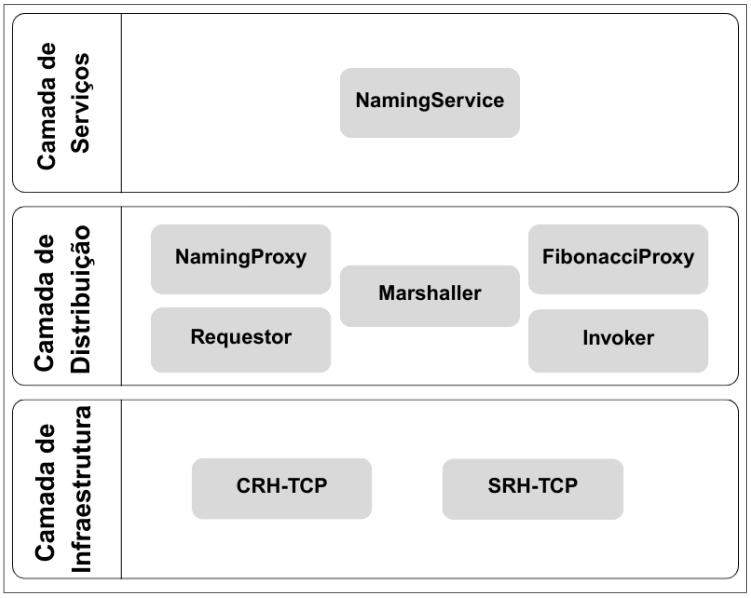
O gMidArch (*go adaptive Middleware aid by software Architecture*) (ROSA; CAMPOS; CAVALCANTI, 2019) (ROSA et al., 2020) é um framework de middleware adaptativo desenvolvido em Go que facilita o projeto, a implementação e a execução de sistemas de middleware adaptativos. As seções seguintes descrevem os principais conceitos do gMidArch.

2.4.1 Componentes de Middleware

Para facilitar o desenvolvimento de sistemas de middleware adaptativos, o gMidArch oferece um conjunto de componentes reutilizáveis para implementar funcionalidades de middleware. Esses componentes são organizados em três categorias de acordo com seu papel

nas camadas do middleware, como pode ser observado na Figura 2. Sendo elas, *Camada de Serviços*, *Camada de Distribuição*, e *Camada de Infraestrutura*.

Figura 2 – Componentes do gMidArch



Fonte: Elaborado pelo autor (2025)

A *Camada de Serviços* fornece serviços para as aplicações. Esta camada inclui tanto serviços que podem ser reutilizados por várias aplicações, e.g., o serviço de nomes, quanto serviços utilizados apenas por algumas aplicações, e.g., serviço de sincronização de edição.

A *Camada de Distribuição* esconde a complexidade dos sistemas distribuídos, ou seja, é a camada responsável pela criação dos pacotes, serialização e codificação dos dados enviados.

A *Camada de Infraestrutura* é responsável pela comunicação dentro do middleware, como envio/recebimento de dados e estabelecimento de conexões. Ela se comunica diretamente com a API de comunicação do sistema operacional. Esta camada também pode implementar transparências de concorrência, tecnologia e falhas. A transparência de concorrência permite que múltiplas operações sejam realizadas simultaneamente, sem interferência entre elas. A transparência de tecnologia permite que diferentes tecnologias de comunicação sejam utilizadas sem que o desenvolvedor precise se preocupar com os detalhes técnicos de cada protocolo. Já a transparência de falhas garante que o sistema continue funcionando mesmo em caso de falhas no envio de mensagens, e.g., perda de mensagens.

2.4.2 Adaptação em tempo de execução

A adaptação em tempo de execução é um dos principais recursos do gMidArch e permite que o middleware se adapte a mudanças nas condições de operação ou nos requisitos das aplicações. Isso é feito através da troca de componentes, sem a necessidade de reiniciar o sistema ou interromper o serviço.

A adaptação em tempo de execução é realizada através do mecanismo de adaptação. Esse mecanismo é responsável por monitorar o estado do middleware e decidir quando e como realizar adaptações. O desenvolvedor pode optar por um mecanismo de adaptação evolutivo ou por não utilizar mecanismos de adaptação. A estratégia de adaptação evolutiva se baseia em monitorar novas versões para os componentes utilizados. Sendo assim, cada vez que uma nova versão é publicada, o middleware automaticamente troca o componente antigo pela nova versão.

2.4.3 Ambiente de execução

O gMidArch possui um ambiente de execução responsável por executar os componentes e conectores e realizar as adaptações. A *unit* é a unidade de execução que gerencia o ciclo de vida dos componentes. O gMidArch cria grafos, que são máquinas de estado, onde cada nó do grafo é uma *unit* que representa o seu componente.

A *unit* é uma abstração que permite que o middleware execute os componentes de forma independente e paralela, garantindo que cada componente possa ser executado em seu próprio contexto. Ela também é responsável por permitir a adaptação dos componentes em tempo de execução.

Quando uma adaptação é iniciada, a *unit* responsável pelo componente a ser trocado encerra a execução do componente depreciado e inicia a execução do novo componente. Caso tenha alguma chamada a ser feita para o componente que está sendo trocado, a *unit* suspende a comunicação até que a troca tenha sido finalizada. Ao finalizar a troca para o novo componente, a comunicação é reestabelecida já para o componente atualizado. Como são ações realizadas em paralelo com a execução do sistema, o impacto da troca é reduzido.

2.5 CONSIDERAÇÕES FINAIS

Este capítulo apresentou a base teórica deste trabalho. Foram apresentados os protocolos de comunicação utilizados e os conceitos de middleware adaptativo e de *Feedback Loop*. Por fim, foi apresentado o framework gMidArch, sua arquitetura e seus componentes e também como ele pode ser utilizado para implementar um middleware adaptativo.

3 pAdapt (*PROTOCOL ADAPTATION*)

Neste capítulo será apresentado em detalhes a solução proposta, o mecanismo ***Protocol Adaptation*** (**pAdapt**) [1]. Primeiramente será apresentada na Seção 3.1 uma visão geral das extensões propostas. Na sequência, a Seção 3.2 detalha o novo mecanismo de adaptação adicionado. Em seguida, na Seção 3.3 serão apresentadas as novas extensões de protocolos de comunicação, contendo os protocolos UDP, TCP, TLS, RPC, QUIC, HTTP, HTTPS e HTTP/2. Por fim, a Seção 3.4 traz as considerações finais deste capítulo.

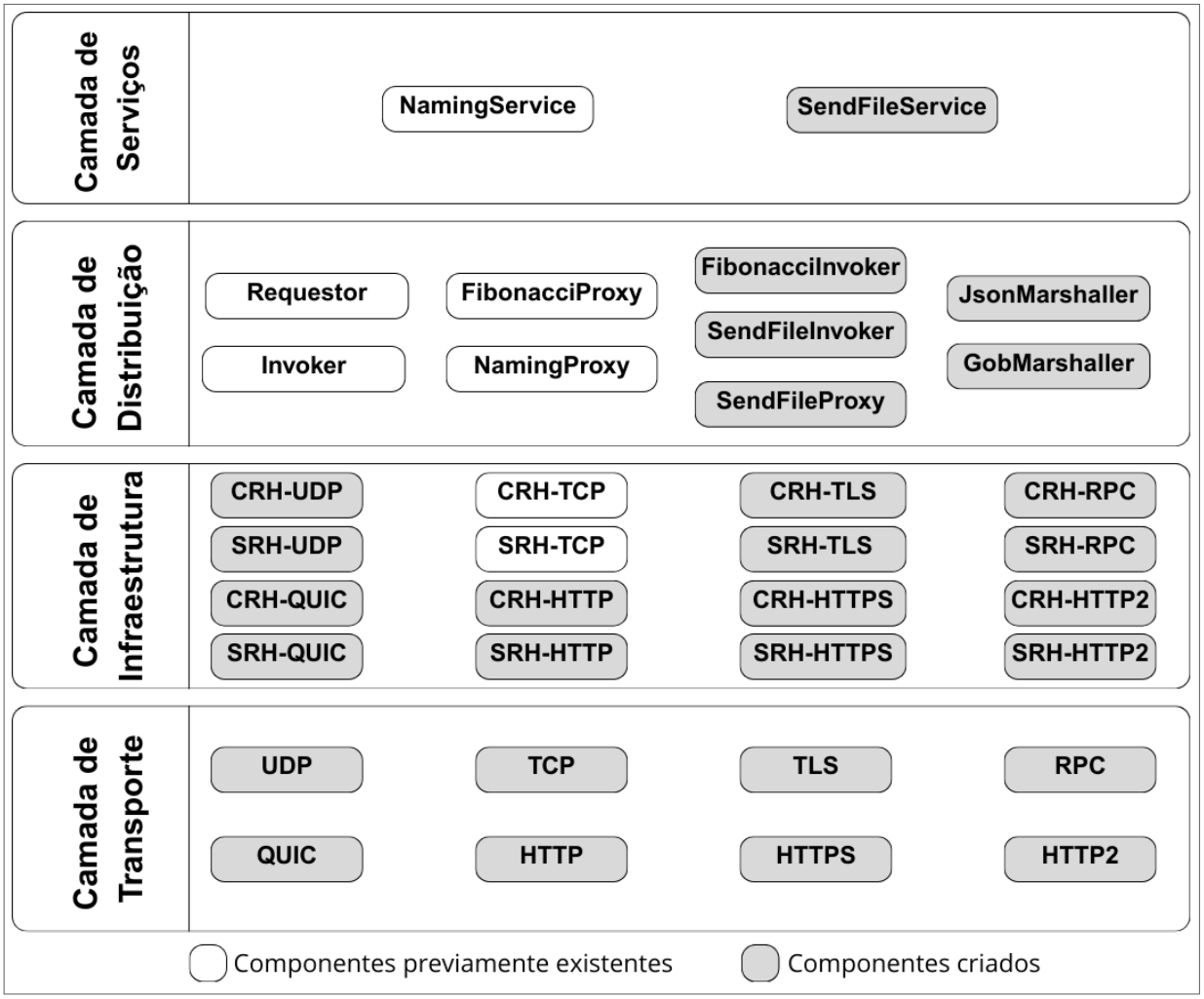
3.1 VISÃO GERAL

A proposta do **pAdapt** consiste de duas partes. A primeira é a adição de novos componentes de transporte, que implementam protocolos de comunicação amplamente utilizados em sistemas distribuídos. A segunda é a implementação de mecanismos que possibilitem que o middleware suporte a adaptação evolutiva de protocolos de comunicação em tempo de execução. A Figura 3 apresenta uma visão geral do gMidArch com as novas extensões (cinza claro), em comparação aos componentes previamente existentes (branco). Além disso, a figura também exibe a adição de uma nova camada, a *Camada de Transporte*, que tem por objetivo agregar as implementações dos protocolos através de uma interface comum a todos. A figura apresenta somente os componentes de middleware, sem os componentes responsáveis pela adaptação e execução, que serão abordados na Seção 3.2.

A maior parte dos novos componentes implementados está na *Camada de Infraestrutura*. Estes componentes são responsáveis pelas transparências de concorrência, tecnologias e falhas. A transparência de concorrência ocorre, e.g., quando diferentes clientes acessam o mesmo serviço no mesmo servidor sem que isso altere o retorno esperado do serviço. Quem faz esta separação da concorrência é o *Server Request Handler*, com seu *pool de conexões*. Já a transparência de tecnologias ocorre ao se utilizar diferentes protocolos independente de qualquer implementação externa ao middleware, i.e., a única coisa que um desenvolvedor de sistemas utilizando o gMidArch com **pAdapt** precisa fazer para escolher um protocolo é configurar um arquivo para utilizar o protocolo desejado. Qualquer diferença entre as diversas tecnologias dos protocolos é tratada internamente pela *Camada de Infraestrutura* e não muda a implementação efetuada pelo desenvolvedor. A transparência de falhas também é implementada,

[1] Código Fonte disponível em: https://github.com/gfads/midarch

Figura 3 – Componentes do gMidArch



Fonte: Elaborado pelo autor (2025)

e.g, na conexão do cliente com o servidor, onde o cliente tenta reconectar automaticamente caso ocorra algum erro na conexão.

Para permitir a adaptação de diferentes protocolos de comunicação, foi criada uma nova camada no middleware, a *Camada de Transporte*, onde de fato é implementado o acesso aos protocolos de comunicação. Esta nova camada fornece serviços à *Camada de Infraestrutura*, através da abstração das chamadas à API do sistema operacional, e.g., para enviar mensagens de um cliente para o servidor. A *Camada de Transporte* serve para padronizar o acesso aos diferentes protocolos de comunicação, e consequentemente viabilizar a troca (adaptação) entre eles em tempo de execução. Para isso, foi criada uma interface que deve ser implementada por cada protocolo. É através desta interface que os componentes da *Camada de Infraestrutura* se comunicam utilizando os protocolos de comunicação. Novos *proxies* e *marshallers*, foram também adicionados à *Camada de Serviços* e à *Camada de Distribuição*.

3.2 MECANISMO DE ADAPTAÇÃO

Um ponto importante no novo mecanismo de adaptação do **pAdapt**, e que o diferencia da adaptação evolutiva padrão previamente existente no gMidArch, é que a adaptação anterior visava somente a alteração de componentes únicos. Com o **pAdapt**, a adaptação (troca de componentes) pode ocorrer de maneira orquestrada no Cliente e no Servidor simultaneamente. Além disso, a adaptação deve ser realizada em todos os clientes que estiverem acessando o servidor. Para possibilitar esta nova adaptação síncrona, foi criado como parte do novo mecanismo, um fluxo de adaptação de protocolos de comunicação, que será abordado na sequência.

3.2.1 Fluxo de Adaptação no pAdapt

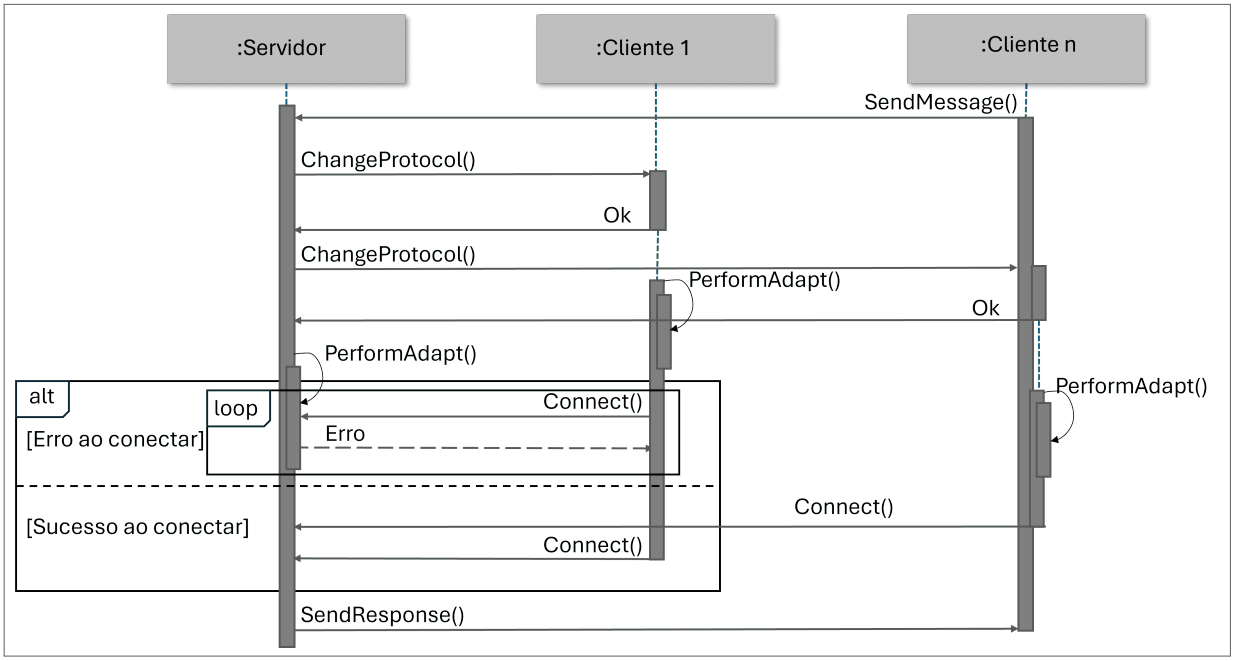
A troca (adaptação) de um protocolo de transporte por outro em tempo de execução, como proposto no **pAdapt**, sincroniza esta ação no cliente e no servidor. A Figura 4 mostra a sequência de ações executadas pelo **pAdapt** para realizar a troca de protocolos.

Para que a adaptação de protocolos de comunicação ocorra, o servidor é o responsável por iniciar o processo de adaptação. Isto é necessário para evitar que um protocolo seja trocado por conta da demanda de um único cliente.

Sendo assim, o Fluxo de Adaptação de Protocolos de Comunicação do **pAdapt**, definido na Figura 4, inicia com o *Servidor* recebendo uma mensagem comum (*SendMessage*) de um *Cliente*. Mas antes de processar a mensagem, o *Servidor* identifica a necessidade de uma troca (adaptação) do seu protocolo. A necessidade de adaptação pode ser através da inclusão de uma nova extensão no middleware, i.e., colocando o código fonte de um novo componente na pasta de extensões do middleware, ou através de uma solicitação de adaptação gerada internamente, baseada no monitoramento do middleware, e.g., pode ser criada uma implementação para que, quando o servidor identificar que o protocolo de transporte atual não é mais o adequado para a comunicação com os clientes conectados, seja 33gerada uma solicitação de adaptação.

Após identificar a necessidade de adaptação, o *Servidor* começa a reter as mensagens (*SendMessage*) recebidas, ou seja, não as processa mais. Isso é necessário para que o *Servidor* não perca nenhuma mensagem durante a adaptação. A partir daí, como pode ser visualizado no diagrama, o *Servidor* envia uma mensagem (*ChangeProtocol*) para cada *Cliente* conectado, informando que ocorrerá uma adaptação de protocolo e para qual protocolo deve ser adaptado.

Figura 4 – Fluxo de Adaptação de Protocolos de Comunicação do **pAdapt**: Sequência de passos



Fonte: Elaborado pelo autor (2025)

Após receber a mensagem de adaptação, o *Cliente* deve responder com uma mensagem de *Ok* para o *Servidor*, confirmando que está pronto para a adaptação. Na sequência, o próprio *Cliente* inicia a adaptação para o novo protocolo (*PerformAdapt*). O *Servidor* aguarda a confirmação através da mensagem de *Ok* de todos os *Clientes* conectados, e somente após receber essa confirmação de todos os *Clientes* é que ele inicia a sua adaptação. Assim que cada *Cliente* finaliza a adaptação, ele tenta se conectar ao *Servidor* novamente (*Connect*), já com o novo protocolo. Caso o *Servidor* já esteja adaptado, ele aceita as novas conexões com *Sucesso ao conectar*, finalizando assim o fluxo de adaptação. Caso o *Servidor* ainda esteja se adaptando, o *Cliente* recebe um *Erro* ao tentar conectar, e continua a tentar a conexão em um *loop* com o novo protocolo até obter sucesso.

Por fim, a comunicação é reestabelecida e o *Servidor* retoma o processamento das mensagens recebidas, agora com o novo protocolo de transporte, enviando o retorno das mensagens retidas com o *SendResponse*.

3.2.2 Extensões de Adaptação

A implementação do gMidArch utiliza uma arquitetura de software baseada em componentes, que permite a construção de sistemas distribuídos de forma modular e flexível. Essa

abordagem facilita a reutilização de componentes existentes e a adição de novos componentes. Com a finalidade de prover estas vantagens, o gMidArch disponibiliza uma *Architecture Description Language* (ADL), ou seja, uma linguagem que descreve arquiteturas, denominada *middleware Architecture Description Language* (mADL) (ROSA et al., 2020), que permite a definição da arquitetura do middleware através de um artefato. Essa arquitetura é composta por componentes, conectores e estratégias de adaptação e representa a estrutura do middleware e como os componentes interagem entre si (MEDVIDOVIC; TAYLOR, 2000).

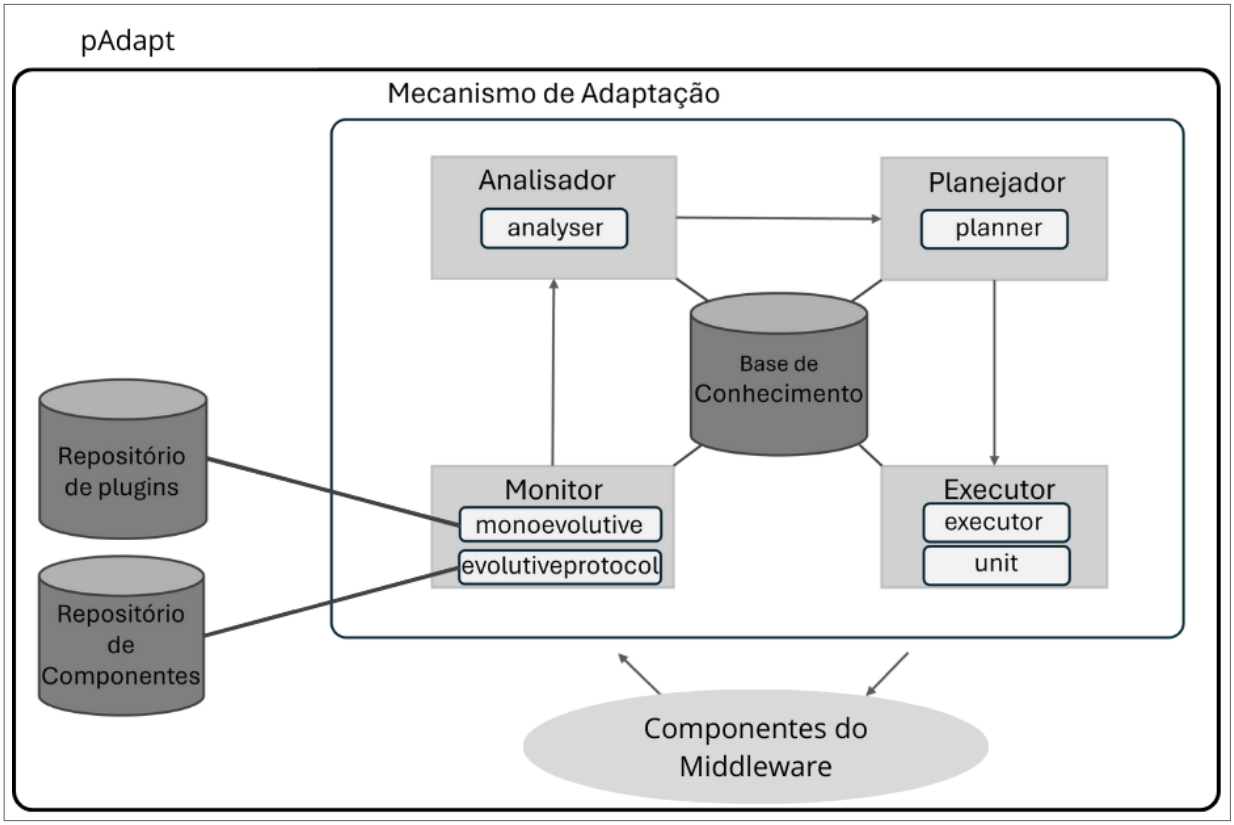
Para indicar o tipo de adaptação do middleware é necessário configurar o mADL. Na seção de estratégias de adaptação do mADL existem três opções, *none* para informar que não se deseja realizar adaptações no middleware, *Evolutive* para o método de Adaptação Evolutiva através de *plugins*, ou *EvolutiveProtocol* para o novo método de Adaptação Evolutiva de Protocolos. Também é possível utilizar os dois métodos ao mesmo tempo. Ao iniciar a aplicação servidora marcando o middleware como utilizando a adaptação evolutiva de protocolos, o middleware passa a monitorar uma lista de componentes a serem adaptados. Caso seja identificada alguma necessidade de adaptação, em qualquer um dos métodos, o componente atual é substituído pelo desejado. Se o componente a ser substituído for um componente de comunicação, então o Fluxo de Adaptação de Protocolos de Comunicação do **pAdapt** mostrado na Figura 4 é seguido.

O gMidArch utiliza o *feedback loop* MAPE-K para realizar as adaptações. Os componentes do MAPE-K foram alterados para implementação da nova adaptação de protocolos. Os componentes do MAPE-K podem ser vistos na Figura 5. Iniciando a sequência do *feedback loop* está o grupo *Monitor* do MAPE-K, que monitora solicitações de adaptação.

O monitoramento já contava com o componente *monoevolutive*, que é a implementação de monitoramento do método de adaptação *Evolutive*. Este componente é o responsável por detectar novas versões de componentes já em uso. Posteriormente nesta seção serão detalhadas as alterações que foram feitas neste componente para também permitir a compilação de códigos fonte em tempo de execução.

Para o novo método de adaptação, *EvolutiveProtocol*, foi implementado um novo componente de monitoramento, onde qualquer necessidade de adaptação para componentes já existentes realiza a inclusão do novo componente ou grupo de componentes a ser adaptado em uma lista. O novo componente adicionado, chamado *evolutiveprotocol*, realiza o monitoramento destes componentes e encaminha a lista com os componentes ao *Analisador*. Além disso, o *evolutiveprotocol* também é responsável por verificar se o componente adicionado na

Figura 5 – Componentes de adaptação do **pAdapt**



Fonte: Elaborado pelo autor (2025)

lista está presente no sistema. Dessa forma, utilizando o método de adaptação *EvolutiveProtocol* componentes do middleware podem ser trocados, mas não é possível adicionar novos componentes ao middleware.

Além do *Monitor*, o *Executor* foi também estendido para suportar a adaptação dinâmica dos protocolos de comunicação. Como o foco do projeto é em como a adaptação é realizada, e não em quando a adaptação deve ser realizada, então os componentes *Analisador* e *Planejador* não precisaram ser estendidos.

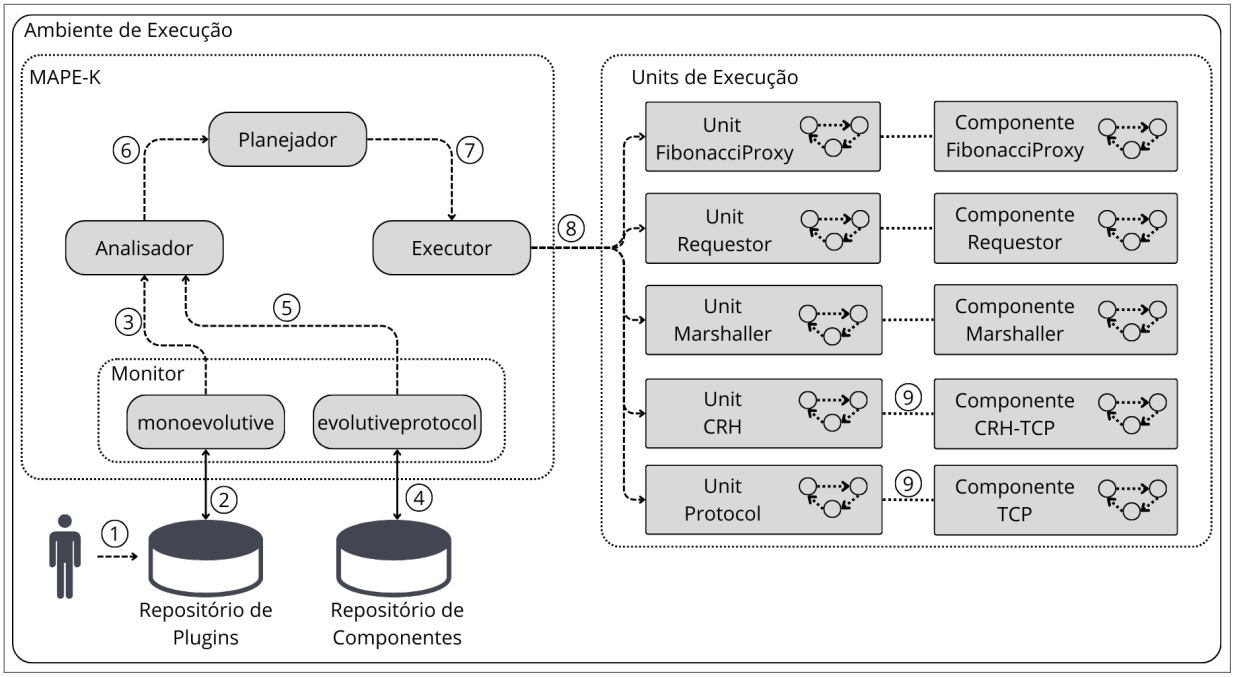
No gMidArch, *Unit* é o componente de apoio tanto à adaptação quanto à execução, um dos componentes mais importantes do **pAdapt**. No gMidArch, os componentes do middleware são executados por uma *Unit*. A *Unit* é basicamente uma unidade de execução de componente, i.e., é um componente que gerencia o ciclo de vida das demais extensões em execução do gMidArch. Ao iniciar uma *Unit*, ele inicia o seu componente, e caso algum componente sofra uma adaptação, a *Unit* finaliza a instância anterior do componente e passa a apontar para uma nova instância do componente atualizado.

3.2.3 Implementação do mecanismo de adaptação do pAdapt

A Figura 6 apresenta o *Ambiente de Execução do gMidArch*. Este ambiente é onde são executados os componentes do middleware, e onde o **pAdapt** realiza as adaptações de protocolos de comunicação. O ambiente de execução é dividido entre o MAPE-K e as *Units* de execução do middleware. A adaptação pode ser iniciada através de dois fluxos possíveis.

O primeiro caminho para adaptação é através do método de adaptação evolutiva, onde um novo plugin é colocado no repositório de plugins do middleware (1), então o *Monitor* evolutivo (*monoevolutive*) detecta sua presença (2) e encaminha para o *Analisador* (3). O segundo caminho para iniciar a adaptação é através do novo método de adaptação evolutiva de protocolos, onde o *Monitor* evolutivo de protocolos (*evolutiveprotocol*) monitora um repositório de componentes (4). Quando o *Monitor* identifica uma necessidade de adaptação, ele encaminha para o *Analisador* (5).

Figura 6 – Ambiente de Execução do **pAdapt**



Fonte: Elaborado pelo autor (2025)

A partir deste ponto, ambos os caminhos seguem o mesmo fluxo. O *Analisador* examina se os componentes a serem adaptados são compatíveis e encaminha para o *Planejador* (6). O *Planejador* então verifica o melhor momento para ser realizada a troca dos componentes, e também é o responsável por indicar ao *Executor* (7) o que deve ser feito durante a adaptação. Na sequência, o *Executor* inicia os procedimentos para a troca dos componentes. Ele gera os

comandos para a troca, compila plugins caso seja necessário, carrega os novos componentes e encaminha para cada *Unit* responsável pelo componente a ser adaptado (8).

Qualquer componente do middleware pode ser adaptado, no entanto, caso o componente a ser adaptado seja relacionado aos protocolos de comunicação, então a *Unit* inicia o Fluxo de Adaptação de Protocolos de Comunicação do **pAdapt**. Vale observar que, como cada componente (incluindo a própria *Unit*) é regida por uma máquina de estados, para realizar a troca é necessário que o componente esteja em seu estado inicial, garantindo assim que os componentes concluirão suas atividades e que podem ser substituídos sem afetar o sistema. Esta funcionalidade também garante que os componentes sejam trocados somente quando não estiverem em execução, i.e., paralelizando a adaptação e gerando menor impacto no desempenho. Por fim, a *Unit* realiza a troca do componente (9), e o novo componente passa a ser executado no lugar do antigo, com uma nova máquina de estados e novas funcionalidades, completando assim o ciclo de adaptação.

Entrando mais em detalhes sobre o funcionamento do *Executor*, quando ele recebe uma solicitação de adaptação, ele carrega a nova extensão e então encaminha para a *Unit* responsável pelo componente a ser adaptado, conforme pode ser observado no fragmento de Código Fonte 1. As Linhas 2 a 4 representam a geração do comando para a Adaptação Evolutiva de Protocolos. Neste ponto, é carregado o novo componente da lista de componentes já existentes dentro do middleware.

Código Fonte 1 – executor.go

```
1  [...]
   if shared.Contains(shared.Adaptability, shared.EVOLUTIVE_PROTOCOL_ADAPTATION) {
3      unitCommand.Cmd = shared.REPLACE_COMPONENT
       unitCommand.Type = shared.GetComponentTypeByNameFromRAM(componentName)
5  } else if shared.Contains(shared.Adaptability, shared.EVOLUTIVE_ADAPTATION) {
       if strings.Contains(pluginName, ".go") {
7          pluginUtils.GeneratePlugin(componentName, versionedPluginName)
           pluginName = versionedPluginName + ".so"
9      }
       plg := pluginUtils.LoadPlugin(pluginName)
11     getType, _ := plg.Lookup("GetType")
       elemType := getType.(func() interface{})()
13     unitCommand.Cmd = shared.REPLACE_COMPONENT
       unitCommand.Params = plg
15     unitCommand.Type = elemType
   }
17 [...]
```

Fonte: Elaborado pelo autor (2025)

Em seguida, as Linhas 5 a 16 representam a geração do comando para a Adaptação Evolutiva padrão. Entre as Linhas 6 a 9, caso a nova extensão seja um código fonte, e não um *plugin*, então o middleware compila o *plugin*. Para isso, na Linha 7, o *Executor* lê o código fonte do novo componente e interpreta o tipo do componente através da modelagem de arquitetura do gMidArch para gerar o novo *plugin*.

Caso não seja um código fonte, seja um *plugin* já compilado previamente, então ele será utilizado sem alterações. Na sequência, o *plugin* é carregado nas Linhas 10 a 12. Por fim, o *plugin* é enviado, juntamente com seu tipo e comando, para a *Unit* nas Linhas 13 a 15.

A *Unit* então utiliza o novo componente para terminar o ciclo de vida do componente anterior e carregar a nova extensão, completando assim a adaptação. No Código Fonte 2 é possível observar o trecho exato de código da *Unit* que realiza a adaptação de um componente.

A princípio, todos os componentes executam infinitamente, até que o sistema se encerre. Como o middleware se baseia na execução de componentes através de máquinas de estado, então cada componente tem sua máquina de estado, e fica em *loop* infinito. Sendo assim, para o gerenciamento do ciclo de vida e a adaptação das extensões, foi criado um atributo em cada componente indicando se é para executar infinitamente ou não. A fim de controlar o ciclo de vida dos componentes, a *Unit* faz a manipulação deste atributo, ativando a execução infinita ao iniciar um componente, e desativando-a quando for necessária uma adaptação.

Código Fonte 2 – unit.go

```
1 [...]
  *elementComponent.ExecuteForever = false
3 for *elementComponent.Executing == true {
      time.Sleep(200 * time.Millisecond)
5 }
  elementComponent.Type = cmd.Type
7 elementComponent.TypeName = cmdElemType
  [...]
```

Fonte: Elaborado pelo autor (2025)

Na Linha 2, do Código Fonte 2, o componente tem sua execução infinita removida, i.e., na próxima vez que a máquina de estado atingir o estado inicial, o componente irá finalizar automaticamente. Enquanto isso não ocorre, a *Unit* aguarda a finalização das atividades do componente a ser substituído (Linhas 3-5).

Na sequência, ou seja, após o componente voltar ao ponto inicial da máquina de estado e finalizar suas atividades (Linhas 6 e 7), o novo componente é carregado no lugar do anterior, e então já passa a executar no próximo ciclo de execução da *Unit*. Como os componentes trocados precisam ser compatíveis, i.e., ter a mesma máquina de estado, os demais componentes que não participam da adaptação não são afetados. A adaptação ocorre em um momento em que não há nada a ser executado pelo componente a ser trocado. Isso é realizado desta forma para garantir que nada se perca durante a execução, tornando a adaptação um processo transparente e dinâmico, aguardando o melhor momento para executar a adaptação.

O **pAdapt** contém um artefato para a geração automática do *plugin*, o *pluginBuild.model*, que é um *boilerplate*, i.e., um fragmento de código sem lógica de aplicação. Este modelo tem a função de complementar um código fonte externo, de forma que ele possa ser compilado em um novo *plugin*. Este *boilerplate* é usado como base, e então modificado dinamicamente em tempo de execução a partir da leitura de informações do código fonte da nova extensão a ser incorporada no gMidArch.

No Código Fonte 3 é possível observar o *pluginBuild.model*. O *Executor* utiliza os dados obtidos da leitura da nova extensão e substitui tanto o <*pluginName*> da Linha 4, quanto o <*pluginType*> da Linha 8, gerando um novo arquivo, o *pluginBuild.go*, que é compilado em um *plugin* e carregado no projeto em tempo de execução.

Código Fonte 3 – pluginBuild.model

```
1 package main
2
3 import (
4     "<pluginName>"
5 )
6
7 func GetType() interface{} {
8     return &<pluginType>
9 }
```

Fonte: Elaborado pelo autor (2025)

A geração de *plugins*, juntamente com as adequações no *Executor*, permitiram a adaptação para qualquer componente pré-existente no middleware e sem a necessidade de duplicação de código. Estas melhorias na adaptação evolutiva padrão foram muito importantes para a adaptação de protocolos de comunicação.

3.2.4 Controle de Estado dos Componentes

A *Unit* do gMidArch, antes do **pAdapt**, considera que os componentes são, *stateless*, i.e., caso haja adaptação, o estado do componente anterior não era repassado ao novo componente. Com o **pAdapt**, os componentes *Client Request Handlers* e *Server Request Handlers* precisam ser *stateful*. Isso é necessário, pois, o estado da conexão entre clientes e servidores precisa ser preservado, i.e., para que a comunicação entre clientes e servidores não seja perdida devido à adaptação. Sendo assim, quando um destes dois componentes são adaptados, o componente antigo encerra seu ciclo de vida, mas suas informações de conexão devem ser salvas e atualizadas no novo componente.

Para adaptar um *Client Request Handler*, é necessário saber os dados de conexão com o servidor mesmo após sua adaptação. E para adaptar um *Server Request Handler*, de acordo com a Figura 4, o **pAdapt** precisa comunicar a alteração a cada cliente conectado e aguardar o *ok* de todos para iniciar a mudança. Para esta implementação, e para permitir múltiplas conexões clientes no mesmo servidor, foi criado um *pool* de conexões, e os clientes conectados são salvos no estado do *Server Request Handler*, em seguida repassados para o novo componente que irá assumir seu lugar após a adaptação. Assim, é possível saber para qual cliente uma mensagem deve ser enviada, mesmo que o protocolo de transporte tenha sido alterado.

Cada *Server Request Handler* implementa seu próprio *pool* de conexões. A fim de padronizar as diferentes implementações de protocolos, foi criada uma interface que deve ser implementada por cada nova implementação de protocolo, seja no servidor ou no cliente. Esta nova interface é apresentada na próxima seção, que apresenta os novos componentes adicionados pelo **pAdapt**.

3.3 NOVOS COMPONENTES

Como apresentado na Seção 3.1, o **pAdapt** inclui um novo mecanismo de adaptação ao gMidArch e um conjunto de novos componentes que implementam diversos protocolos de comunicação. As seções a seguir apresentam detalhes de como este protocolos foram definidos e implementados.

3.3.1 Escolha dos protocolos

A escolha dos novos protocolos de comunicação iniciou considerando protocolos de menor complexidade e avançando para os mais atuais e complexos. A seleção dos protocolos foi baseada em dois critérios principais: o estilo cliente/servidor e a funcionalidade oferecida. Com relação ao primeiro critério, um protocolo de transporte adequado deve suportar naturalmente o modelo de interação cliente/servidor. Quanto à funcionalidade, é esperado que os novos protocolos agreguem valor ao middleware, como novas características, e.g., segurança, eficiência e flexibilidade.

Sendo assim, foram escolhidos primeiro os protocolos mais básicos. Iniciando com UDP, por sua velocidade, seguindo para o TCP, pela confiabilidade e finalizando com TLS para adicionar segurança. Na sequência foi adicionado o próprio RPC como um método clássico de comunicação cliente/servidor e no qual o gMidArch tem sua arquitetura baseada. O próximo protocolo é o QUIC como um protocolo mais moderno e inovador, com a principal vantagem de ser utilizado para streaming de dados.

Por fim, também foram adicionados os protocolos HTTP, HTTPS e HTTP/2 por serem populares e flexíveis. Com estes protocolos é possível criar aplicações Web, e também realizar comunicação com navegadores, ou seja, com a implementação destes protocolos é possível efetuar chamadas ao gMidArch diretamente dos navegadores. O uso destes protocolos, aliado ao componente *Jsonmarshaller*, que empacota mensagens com o uso de JSON, permite que sistemas de middleware implementados com o gMidArch possam ser adotados no desenvolvimento de microsserviços que usam REST (FOWLER, 2014).

Os protocolos HTTP, HTTPS e HTTP/2 utilizam TCP e são protocolos da camada de aplicação, de acordo com o modelo *Open Systems Interconnection* (OSI) (ISO/IEC, 1994). No gMidArch, os protocolos HTTP, HTTPS e HTTP/2 são utilizados somente para transporte e comunicação entre sistemas, da mesma forma que os demais, e.g., TCP. Todos os componentes de protocolos de comunicação do **pAdapt** implementam a mesma interface, e fornecem os mesmos serviços à camada superior. Sendo assim, apesar de HTTP, HTTPS e HTTP/2 serem protocolos da camada de aplicação, no gMidArch eles são tratados como protocolos de transporte para comunicação.

3.3.2 Extensões de Protocolos de Comunicação

No **pAdapt** foram implementados os protocolos de comunicação UDP, TCP, TLS, RPC, QUIC, HTTP, HTTPS e HTTP/2. É possível efetuar a adaptação de qualquer protocolo para qualquer protocolo. Para que eles sejam compatíveis foi criado a interface *Protocol*. A interface *Protocol* é uma interface composta, ou seja, ela contém outra interface, a interface *Client*, que define os métodos de comunicação do servidor para o cliente. Estas interfaces devem ser implementadas por cada novo protocolo adicionado ao middleware. Isso é necessário pois a *Unit* utiliza os protocolos através das interfaces para implementar a sincronização entre clientes e servidores e garantir o fluxo apresentado na Figura 4.

O Código Fonte 4, *protocol.go*, apresenta estas duas interfaces. A primeira define os métodos disponíveis para o protocolo em si (Linhas 1-17), e a outra representa os clientes conectados ao pool de conexões do servidor (Linhas 18-29), e define as assinaturas de seus métodos.

Código Fonte 4 – protocol.go

```
1  type Protocol interface {
       StartServer(ip, port string, initialConnections int)
3      StopServer()
       AvailableConnectionFromPool() (available bool, idx int)
5      WaitForConnection(cliIdx int) (cl *Client)
       GetClients() (clients []*Client)
7      GetClient(id int) (client Client)
       GetClientFromAddr(addr string) (client Client)
9      AddClient(client Client, id int)
       ResetClients()
11     ConnectToServer(ip, port string)
       CloseConnection()
13     ReadString() string
       WriteString(message string)
15     Receive() ([]byte, error)
       Send(msgToServer []byte) error
17 }
   type Client interface {
19     AdaptId() int
       SetAdaptId(adaptId int)
21     Address() string
       Connection() (conn interface{})
23     CloseConnection()
       Read(b []byte) (n int, err error)
25     ReadString() (message string)
       WriteString(message string)
27     Receive() ([]byte, error)
```

```
    Send(msgToServer []byte) error
}
```

Fonte: Elaborado pelo autor (2025)

A primeira interface, *Protocol*, contém tanto métodos utilizados pelo servidor (Linhas 2-10), quanto métodos utilizados pelo cliente (Linhas 11-16). Os métodos utilizados pelo servidor gerenciam tanto o ciclo de vida do protocolo, i.e., iniciar e finalizar o servidor (Linhas 2 e 3), quanto o *pool* de conexões (Linhas 4-10). Para o *pool* de conexões, a interface define assinaturas para aguardar novas conexões de clientes, obter conexões disponíveis do *pool*, obter os clientes conectados e remover clientes do *pool*. Por outro lado, os métodos definidos para o lado do cliente (Linhas 11-16) são utilizados para conectar ao servidor, fechar a conexão, enviar mensagens e receber mensagens.

Já a interface *Client*, ela define como o servidor configura e obtém as identificações de cada cliente (Linhas 19 e 20), através do *AdaptId*, utilizado para identificar o cliente nas adaptações. Esta interface também contém os métodos utilizados pelo servidor para gerenciar a conexão dos clientes conectados (Linhas 21-23). Outra funcionalidade importante é a de enviar e receber mensagens do cliente (Linhas 24-28).

O Código Fonte 5 executa durante a adaptação de componentes efetuada pela *Unit*, e mostra não só como a *Unit* trata o Fluxo de Adaptação de Protocolos de Comunicação do **pAdapt**, mas também como é a utilização dos protocolos através da interface *Protocol*. Nas Linhas 2 a 7, a *Unit* percorre todos os clientes conectados ao servidor, e envia uma mensagem de adaptação para cada um deles, informando o novo protocolo. Nas Linhas 9 a 12, a *Unit* finaliza o ciclo de vida do componente anterior, e nas Linhas 13 e 14 a *Unit* inicia o novo componente. Por fim, as Linhas 16 a 19 param o servidor, que será iniciado novamente com o novo protocolo de transporte quando o novo componente executar pela primeira vez.

Código Fonte 5 – unit.go

```
[...]
    for idx, client := range srhInfo.Protocol.GetClients() {
        (*client).SetAdaptId(idx)
        miopPacket := miop.CreateReqPacket("ChangeProtocol", []interface{}{
            adaptTo, (*client).AdaptId()}, (*client).AdaptId())
        msg := &messages.SAMessage{ToAddr: (*client).Address(), Payload:
            marshaller{}.Marshall(miopPacket)}
        shared.MyInvoke(elementComponent.Type, elementComponent.Id, "I_Send", msg
            , &elementComponent.Info, &reset)
```

```
7       }
        [...]
9       *elementComponent.ExecuteForever = false
        for *elementComponent.Executing == true {
11          time.Sleep(200 * time.Millisecond)
        }
13      elementComponent.Type = cmd.Type
        elementComponent.TypeName = cmdElemType
15      [...]
        srhInfo := elementComponent.Info.(*messages.SRHInfo)
17      if srhInfo.Protocol != nil {
            srhInfo.Protocol.StopServer()
19          srhInfo.Protocol = nil
        }
21  [...]
```

Fonte: Elaborado pelo autor (2025)

As próximas seções apresentam detalhes de cada protocolo implementado.

3.3.3 UDP

Apesar da diferença em como funcionam, os componentes TCP e UDP têm comportamentos de alto nível semelhantes no gMidArch. O fluxo padrão de um protocolo UDP externo ao middleware, do lado do cliente, é: enviar uma mensagem para o servidor, aguardar que o servidor processe a mensagem, e então receber uma resposta do servidor, sem a necessidade de estabelecer uma conexão prévia. Já no fluxo padrão de um protocolo TCP, existe uma etapa adicional para criar uma conexão antes de enviar a mensagem. No entanto, no **pAdapt**, ambos os protocolos são implementados para funcionar de forma semelhante, i.e., ambos estabelecem uma conexão antes de enviar a mensagem. Isso se faz necessário tanto para que o componente *UDP* possa implementar a interface *Protocol*, quanto para que o **pAdapt** possa garantir a entrega de mensagens fracionadas em pacotes, e.g., no envio de arquivos. Além disso, também é necessário para que o **pAdapt** possa efetuar a adaptação entre diferentes protocolos de comunicação, de forma que o servidor possa identificar seus clientes e efetuar troca de mensagens durante a adaptação.

Como clientes e servidores precisam usar o mesmo protocolo de transporte, o suporte ao UDP inclui três novos componentes: *Client Request Handler UDP* (*CRH-UDP*), *Server Request Handler UDP* (*SRH-UDP*) e a implementação UDP da interface *Protocol*.

Neste ponto, vale observar que *Client* e *Server Request Handlers* são padrões arquiteturais de middleware amplamente adotados (VOLTER; KIRCHER; ZDUN, 2005), cujas responsabilidades são gerenciar todos os aspectos da comunicação em clientes e servidores, respectivamente. Eles abrem/fecham conexões em protocolos orientados a conexão, funcionam como o único ponto de contato com as APIs de *sockets* dos sistemas operacionais e isolam outros componentes de middleware da complexidade de lidar com problemas de transporte de mensagens. Estes padrões arquiteturais estarão presentes em todos os protocolos de comunicação implementados no gMidArch.

Apesar de a especificação do UDP não conter conexões ativas (POSTEL, 1980), no **pAdapt**, esta conexão é implementada por software. Em outras palavras, antes de enviar uma mensagem, o *Client Request Handler UDP* estabelece uma conexão com o servidor. Isso é realizado através de troca de mensagens, de forma transparente entre o *CRH-UDP* e o *SRH-UDP*. O *CRH-UDP* envia uma mensagem *"Connect"* para o servidor, que responde com uma mensagem *"Ok"*, caso a conexão tenha sido estabelecida com sucesso. Caso não consiga conectar, o *CRH-UDP* tenta novamente.

Como a adaptação é *stateful*, é necessário que o servidor conheça todos os clientes conectados para solicitar as adaptações. Isso também é necessário para efetuar a garantia de entrega das mensagens, especialmente em mensagens grandes, e.g., no envio de arquivos. Para garantir a entrega das mensagens, o **pAdapt** implementa um mecanismo de confirmação de entrega de mensagens, onde o cliente envia uma mensagem e aguarda a confirmação de entrega do servidor. Caso o servidor não confirme a entrega, o cliente reenvia a mensagem.

Não é possível enviar uma mensagem grande em um único pacote UDP, pois o tamanho máximo de um pacote é de 65.507 bytes. Para mensagens maiores que ultrapassam o tamanho máximo de pacotes UDP, e.g., no envio de arquivos, o UDP também implementa o fracionamento da mensagem em pacotes. Além de garantir a entrega de pacotes, é necessário também garantir a sequência dos mesmos, uma vez que o protocolo UDP não garante nem a entrega de pacotes, nem a ordem de entrega dos pacotes.

Para garantir a entrega da mensagem completa no UDP, mesmo em mensagens grandes, foi implementado uma sequência de algorítmos: o fracionamento da mensagem de acordo com o tamanho máximo configurado, o envio de pacotes ordenados, o reenvio de pacotes perdidos, a confirmação de entrega dos pacotes e a reconstrução da mensagem fracionada.

Para melhorar o desempenho do envio de mensagens grandes, o UDP implementa o envio de pacotes em paralelo, onde o cliente envia vários pacotes ao mesmo tempo, juntamente com

um sequencial, e no fim aguarda a confirmação. Caso o servidor não tenha recebido algum pacote, ele solicita o reenvio dos pacotes faltantes. Evitando a confirmação em cada pacote, o envio de pacotes em paralelo melhora o desempenho do envio de arquivos. Para que o servidor saiba quantas mensagens esperar, a primeira mensagem trocada entre cliente e servidor é um cabeçalho que contém o tamanho total da mensagem.

Do lado do servidor, cada mensagem é esperada juntamente com o seu respectivo sequencial. O servidor ao final informa se recebeu todas as mensagens com sucesso, ou se faltou alguma mensagem enviando o sequencial da mensagem faltante. O Código Fonte 6 implementa como um cliente UDP envia um arquivo.

Código Fonte 6 – ClientRequestHandlerUDP Send File

```
1  func (st *UDP) Send(msgToServer []byte) error {
     _, err := st.serverConnection.Write(len(msgToServer))
3    if err != nil {...}
     bufferSize, payloadSize, packetsQuantity := calcSizes(len(msgToServer))
5
     for seq := 0; seq < packetsQuantity; seq++ {
7      packet := getPacket(msgToServer, seq, payloadSize)
       st.serverConnection.Write(packet)
9      time.Sleep(5 * time.Microsecond)
     }
11
     for {
13     st.serverConnection.SetReadDeadline(time.Now().Add(1 * time.Second))
       n, err := st.serverConnection.Read(ackBuffer)
15     if err != nil {...}

17     ack := strings.TrimSpace(string(ackBuffer[:n]))
       if ack == "ack" { break }
19
       seqInt64, err := strconv.ParseUint(ack, 10, 32)
21     st.serverConnection.Write(getPacket(msgToServer, int(seqInt64), payloadSize))
     }
23   return nil
   }
```

Fonte: Elaborado pelo autor (2025)

O Código 6 começa nas Linhas 2-4 enviando o tamanho da mensagem ao servidor, e calculando quantos pacotes serão necessários para enviar o arquivo completo. Na sequência, nas Linhas 6-8, o cliente envia os pacotes, com um *delay* de 5 microssegundos na Linha 9

para evitar congestionamento, e consequente perda de pacotes. O cliente envia os pacotes em paralelo, ou seja, não aguarda a confirmação do servidor antes de enviar o próximo pacote.

As Linhas 12-23 enviam a confirmação de entrega dos pacotes, que é repetido até que a mensagem tenha sido enviada com sucesso. Nas Linhas 13-15, o cliente aguarda uma resposta do servidor, que pode ser um *ok*(Linhas 17-18), ou um número de pacote. Caso receba um número de pacote, o cliente reenvia o pacote correspondente, conforme Linhas 20-21.

Na Figura 4, é possível observar que o Cliente pode finalizar sua adaptação antes mesmo do servidor. Sendo assim, pode ocorrer um erro de conexão ao tentar enviar mensagens para um protocolo que ainda não está online. Para evitar este problema e auxiliar no *pool* de conexões no lado do *Server Request Handler* UDP, o Código Fonte 7 apresenta como o *Client Request Handler* realiza uma sequência de tentativas de conexões, enviando uma mensagem com operação *"Connect"*, Linhas 3 a 5, e verificando o retorno do servidor com *"Ok"* na Linha 10.

Código Fonte 7 – ClientRequestHandler UDP Connection

```
for {
  time.Sleep(200 * time.Millisecond)
  pck := miop.CreateReqPacket("Connect", []interface{}{adaptId}, adaptId)
  msgPayload := marshaller{}.Marshall(miopPacket)
  err = c.send(sizeOfMsgSize, msgPayload, crhInfo.Conns[addr])
  if err != nil {...}
  msgFromServer, err := c.read(crhInfo.Conns[addr], sizeOfMsgSize)
  if err != nil {...}
  if isNewConnection, miopPacket := c.isNewConnection(msgFromServer);
      isNewConnection {
      if miopPacket.Bd.ReqBody.Body[1] == "Ok" {
          break
      }
  }
}
```

Fonte: Elaborado pelo autor (2025)

3.3.4 TCP

Os componentes *Server Request Handler TCP* (*SRH-TCP*) e *Client Request Handler TCP* (*CRH-TCP*) pertenciam ao único protocolo implementado na versão anterior do gMidArch, o TCP. No entanto, foram necessárias alterações para permitir a adaptação: a implementação do

TCP foi desacoplada, i.e., foi gerado um novo componente TCP aderente à interface *Protocol*, foi criado um *pool* de conexões para suportar vários clientes e os componentes *Server Request Handler TCP* e *Client Request Handler TCP* foram alterados para tratar conexões como *stateful*, ou seja, persistindo e utilizando dados de clientes conectados para identificar cada conexão e permitir a adaptação. Para poder utilizar o *pool* de conexões, o *SRH-TCP* teve que ser alterado pois anteriormente, ao aguardar uma conexão, ele ficava bloqueado até que um cliente se conectasse.

Agora, o *SRH-TCP* faz uso de *goroutines*, um recurso da *Go language* (Golang) para concorrência, similar às *threads*, no entanto, enquanto as *threads* são gerenciadas pelo sistema operacional e consomem mais memória para salvar seus estados ao alternar de uma *thread* para outra, as *goroutines* são gerenciadas pela própria Golang, e consomem menos memória pois não necessitam salvar os estados a cada mudança de rotina. As *goroutines* são utilizadas no *SRH-TCP* para aceitar múltiplas conexões de clientes, e assim não ficar bloqueado aguardando uma conexão. Este recurso permite que um cliente possa estar conectando enquanto outros clientes estão enviando e recebendo mensagens simultaneamente.

3.3.5 TCP+TLS

A criação dos três componentes para TCP+TLS foram baseadas nos componentes TCP. Foram adicionados os componentes: *Client Request Handler TLS* (*CRH-TLS*) e *Server Request Handler TLS* (*SRH-TLS*) e a implementação da interface *Protocol* TCP+TLS. A principal diferença do TCP+TLS para a extensão TCP é que os componentes TLS geram a configuração TLS baseada no TLS 1.3, conforme especificado no RFC 8446 (RESCORLA, 2018). Sendo assim, *CRH-TLS* e *SRH-TLS* precisam de certificados para gerar as configurações TLS. Para isso é necessário configurar o caminho para os certificados através das variáveis de ambiente conforme a Tabela 2.

Tabela 2 – Variáveis de ambiente para configuração de certificados para utilização com protocolos seguros

Variável	Descrição
CA_PATH	Caminho para o certificado da autoridade certificadora.
CRT_PATH	Caminho para o certificado da aplicação/servidor.
KEY_PATH	Caminho para a chave do certificado da aplicação/servidor.

Fonte: Elaborado pelo autor (2025)

3.3.6 QUIC

O QUIC (LANGLEY et al., 2017) é um protocolo de transporte implementado sobre o UDP e projetado pelo Google. O objetivo inicial do QUIC era melhorar o tráfego na Internet. Portanto, ele foi projetado sobre o UDP como uma camada de transporte de uso geral para reduzir a latência em comparação ao TCP. As características essenciais do QUIC incluem redução no tempo de estabelecimento de conexão, controle de congestionamento aprimorado, multiplexação sem bloqueio e migração de conexão. Assim, o QUIC usa a simplicidade e a velocidade do UDP, mas implementa confiabilidade e segurança sobre ele. O QUIC é um protocolo amplamente utilizado, pois é a base do *HyperText Transfer Protocol version 3* (HTTP/3) (BISHOP, 2022), presente nos sites mais modernos e nas *Big Techs*.

Apesar de usar UDP, o QUIC é semelhante à extensão TCP+TLS, pois eles usam a mesma configuração do TLS 1.3, i.e., as mesmas variáveis de ambiente do Quadro 2. Diferentemente do TCP+TLS, a implementação do QUIC precisa implementar recursos como controle de conexões através de *stream* (forma que o QUIC usa para a multiplexação), que funciona como uma conexão adicional para o mesmo cliente. Utilizando o QUIC, os clientes podem estabelecer uma conexão e usar a mesma conexão para se comunicar em um ou mais *streams*. Vale notar que essa característica aumenta a velocidade das transferências, permitindo várias solicitações simultâneas do mesmo cliente.

Semelhante às extensões anteriores, três novos componentes foram implementados usando o protocolo QUIC baseado no RFC 9000 (IYENGAR; THOMSON, 2021): o *Client Request Handler QUIC* (*CRH-QUIC*), o *Server Request Handler QUIC* (*SRH-QUIC*) e a implementação da interface *Protocol* para QUIC. Como o Go não possui uma implementação nativa do protocolo QUIC, o pacote usado nos novos componentes é o *quic-go* (quic-go Contributors, 2024), um pacote Go construído pela comunidade que ainda não está na versão estável, mas é o pacote QUIC mais funcional disponível para Go.

O Código Fonte 8 faz parte da implementação do *Server Request Handler QUIC* (*SRH-QUIC*) e é necessário para aceitar uma conexão, aceitar um *stream*, e receber mensagens do cliente.

Código Fonte 8 – Conexão *Server Request Handler* do QUIC

```
func (st *QUIC) WaitForConnection(cliIdx int) (cl *generic.Client) {
    conn, err := st.listener.Accept(context.Background())
    if err != nil {
        [...]
```

```
    }
6   if len(st.clients) > cliIdx {
        (*st.clients[cliIdx]).(*QUICClient).connection = conn
8       (*st.clients[cliIdx]).(*QUICClient).Ip = conn.RemoteAddr().String()
        (*st.clients[cliIdx]).(*QUICClient).stream, err = conn.AcceptStream(
            context.Background())
10  [...]
```

Fonte: Elaborado pelo autor (2025)

As Linhas 2 a 5 implementam os passos para aceitar uma conexão, similar ao *Server Request Handler TCP* (*SRH-TCP*). As Linhas 7 a 9 explicitam a diferença entre QUIC e TCP, pois o QUIC precisa de um controle extra sobre a comunicação através de *streams*, quase como uma conexão adicional.

3.3.7 RPC

A implementação do RPC foi construída sobre o Go RPC, que é uma biblioteca nativa da linguagem Go. Os novos componentes utilizam o Go RPC como um mecanismo de transporte de *request-reply*, i.e., somente para enviar e receber mensagens.

O RPC se baseia na chamada de funções remotas, onde a implementação da função remota está associada à regra de negócio, e.g., para calcular o Fibonacci utilizando RPC, o esperado é que a função registrada no servidor RPC tenha o nome e os parâmetros da função que faz o cálculo do Fibonacci. No entanto, para criar a abstração da comunicação, os novos componentes RPC implementam a interface de Protocolos, onde o padrão para envio e resposta de mensagens é através de *arrays* de bytes. Sendo assim, por mais que o usuário do gMidArch implemente diferentes regras de negócio, com funções e parâmetros diferentes, o RPC sempre irá enviar e receber mensagens através de *arrays* de bytes. Em outras palavras, o RPC implementado no gMidArch é um RPC genérico, onde o usuário pode implementar qualquer regra de negócio, mas a comunicação sempre será feita através de *arrays* de bytes, i.e., implementação da função do RPC nos novos componentes registram uma função chamada *Request*, que é responsável somente pelo envio da mensagem como um *array* de bytes. Com isso, o RPC se torna um protocolo de transporte de acordo com o modelo OSI.

Para atingir este objetivo, foram implementados os componentes *Client Request Handler RPC* (*CRH-RPC*) e *Server Request Handler RPC* (*SRH-RPC*), além da implementação pro-

priamente dita da interface de Protocolo para RPC.

Como o servidor no Go RPC fica escutando por conexões ativamente, e só responde na implementação do objeto registrado, foi necessário adaptar seu funcionamento, de forma a quebrar sua execução nas funções da interface de Protocolo. Para isso, foi criado um objeto remoto que divide a execução do servidor em funções, fazendo uso dos canais do Go para enviar e receber mensagens.

O Código Fonte 9 mostra como este objeto remoto e sua função remota são implementados no servidor. Este objeto é responsável por receber as mensagens do cliente, processá-las e enviar a resposta. As Linhas 1 a 4 definem a *struct* do objeto remoto, que contém o canal de mensagens recebidas e o canal de respostas. As Linhas 6 a 12 implementam a função remota que é chamada a cada requisição. A função recebe a mensagem do cliente como parâmetro na Linha 6. Em seguida, nas Linhas 7 a 9, esta função coloca a mensagem em um canal para ser processada e aguarda a resposta no canal de retorno na Linha 10. Por fim, a função envia a resposta para o cliente na Linha 11.

Código Fonte 9 – Servidor RPC

```
type RPCRequest struct {
    msgChan   chan []byte
    replyChan chan []byte
}

func (rq RPCRequest) Request(request []byte, reply *[]byte) error {
    go func() {
        rq.msgChan <- request
    }()
    replyMsg := <-rq.replyChan
    *reply = replyMsg
    return nil
}
```

Fonte: Elaborado pelo autor (2025)

3.3.8 HTTP/1.1

Os componentes HTTP/1.1 trazem várias novas possibilidades para o gMidArch, uma vez que seu componente *Server Request Handler HTTP* (*SRH-HTTP*) pode atuar como um servidor Web e interagir com clientes através de navegadores ou qualquer outro cliente HTTP.

Apesar de HTTP não ser um protocolo da camada de transporte segundo o modelo OSI, todos os componentes HTTP foram implementados de forma a serem compatíveis com os demais protocolos de comunicação. Para isso, estes componentes no gMidArch atuam somente na *Camada de Infraestrutura* com o *Client Request Handler HTTP* e *Server Request Handler HTTP* e na *Camada de Transporte*, com a implementação do protocolo HTTP/1.1, servindo como uma forma de enviar e receber *stream* de bytes, e controlando o sucesso das requisições com os códigos de status HTTP, assim como é feito ao se implementar uma API REST.

O componente HTTP implementa a especificação HTTP/1.1 (FIELDING; NOTTINGHAM; RESCHKE, 2022), mas não inclui TLS. Essa extensão foi implementada usando o pacote *net/HTTP* da linguagem Go. Para que o componente HTTP seja compatível com os demais protocolos de comunicação, a interface *Protocol* foi implementada. O método *StartServer* definido na interface *Protocol* e implementado no componente HTTP prepara um servidor HTTP/1.1, que passa a aceitar conexões no método *WaitForConnection* (também da interface *Protocol*). O servidor HTTP então fica executando em *background*.

Ao contrário dos demais protocolos, que devem explicitamente ler mensagens, um servidor HTTP fica esperando ativamente por uma mensagem. Sendo assim, o que o método *Receive* faz é ficar observando um canal, como visto nas Linhas 2 e 3 no Código Fonte 10.

Código Fonte 10 – HTTP Receive

```
1 func (cl *HTTPClient) Receive() (msg []byte, err error) {
      msg = <-cl.msgChan
3 }
```

Fonte: Elaborado pelo autor (2025)

Por sua vez, este canal é carregado quando o servidor HTTP recebe a mensagem, conforme o Código Fonte 11. Esta função é executada em cada recebimento de mensagem no servidor HTTP. A Linha 1 é a assinatura padrão de uma função HTTP, onde é passado um *response writer* (contendo a mensagem a ser respondida ao cliente) e um *request* (a mensagem que o cliente enviou ao servidor). As Linhas 2 e 3 leem a mensagem recebida, e as Linhas 4 a 6 são responsáveis por enviar a mensagem para o canal observado pelo método *Receive*. Na sequência, o gMidArch processa a mensagem recebida de forma padrão, como qualquer outro protocolo, e aguarda a resposta na Linha 7.

Código Fonte 11 – HTTP Serve

```
1 func (rq HTTPRequest) ServeHTTP(w http.ResponseWriter, r *http.Request) {
```

```
    msg, _ := io.ReadAll(r.Body)
    r.Body.Close()
    go func() {
        rq.msgChan <- msg
    }()
    replyMsg := <-rq.replyChan
    w.Write(replyMsg)
}
```

Fonte: Elaborado pelo autor (2025)

Ao fim do processamento da mensagem, o *Server Request Handler HTTP* chama a função *Send* do componente HTTP, e assim como o *Receive*, se comunica com o servidor HTTP através de canais. Conforme Código Fonte 12, pode ser observado nas Linhas 2 a 4 que ele somente coloca a mensagem de retorno no canal do servidor HTTP, que é o mesmo canal que ficou aguardando na Linha 7 no Código Fonte 11.

Código Fonte 12 – HTTP Send

```
1 func (cl *HTTPClient) Send(msg []byte) error {
      go func() {
3         cl.replyChan <- msg
      }()
5 [...]
```

Fonte: Elaborado pelo autor (2025)

Como o servidor HTTP deve fornecer uma resposta para o cliente, a Linha 7 do Código Fonte 11 lê a mensagem de resposta do canal e a envia para o cliente na Linha 8, finalizando o ciclo de vida da mensagem HTTP/1.1 no servidor.

3.3.9 HTTPS

Um novo protocolo de transporte foi adicionado ao gMidArch para melhorar a segurança na comunicação como um servidor Web, o HTTPS. Conforme mencionado na extensão TCP+TLS (Seção 3.3.5), o TLS 1.3 melhora tanto a confiança sobre a identidade do servidor, quanto dificulta que alguém seja capaz de ler as mensagens trocadas entre cliente e servidores.

Três novos componentes foram adicionados para suportar o HTTPS: *CRH-HTTPS* e *SRH-HTTPS*, além da própria implementação do protocolo. Essa nova extensão HTTPS é semelhante à do HTTP, no entanto, ela utiliza o TLS como camada de segurança para transportar

mensagens. Para isso, além do pacote *net/HTTP* da linguagem Go, foi utilizado o pacote *crypto/tls* para implementar a criptografia. Como o TLS se baseia na utilização de certificados para a geração da criptografia, foram utilizadas as mesmas variáveis de ambiente para configuração de certificados da Tabela 2.

3.3.10 HTTP/2

A implementação dos componentes HTTP/2 é similar à dos componentes HTTPS, diferenciando basicamente nos pacotes utilizados para implementação. O componente HTTP/2 utiliza o pacote *golang.org/x/net/http2* da linguagem Go, que implementa a especificação HTTP/2 (BELSHE; PEON; THOMSON, 2015). Apesar de o HTTP/2 não exigir criptografia, muitas implementações importantes só suportam HTTP/2 sobre TLS, e.g., Chrome, Firefox e Safari (NAZIRIDIS, 2018). Portanto, os componentes HTTP/2 do **pAdapt** também suportam somente HTTP/2 sobre TLS 1.3 e consequentemente utilizam as mesmas variáveis de ambiente da Tabela 2.

3.4 CONSIDERAÇÕES FINAIS

Este capítulo iniciou apresentando a visão geral da solução proposta, o **pAdapt**. Em seguida, foi apresentado o mecanismo de adaptação de protocolos de comunicação em tempo de execução, abordando o Fluxo de Adaptação de Protocolos de Comunicação, as extensões de adaptação, a implementação do novo mecanismo e o novo controle de estado dos componentes. Finalmente, na última parte do capítulo, foram apresentados os novos componentes de comunicação, a interface *Protocol*, como foram escolhidos os protocolos adicionados e os detalhes de implementação para cada conjunto de componentes de cada protocolo.

4 AVALIAÇÃO EXPERIMENTAL

Neste capítulo são apresentados os experimentos realizados para avaliar o desempenho do **pAdapt**, tanto com relação ao desempenho dos componentes de transporte do próprio middleware, quanto em comparação com sistemas de middleware comerciais. A Seção 4.1 apresenta os objetivos da avaliação, enquanto a Seção 4.2 descreve as métricas, parâmetros e a carga de trabalho utilizados nos experimentos. A Seção 4.3 descreve os fatores e o projeto dos experimentos. Na sequência, a Seção 4.4 apresenta a solução em ação, detalhando como o **pAdapt** foi implementado e utilizado nos experimentos. A Seção 4.5 apresenta os resultados e a análise dos resultados. Por fim, a Seção 4.6 apresenta as considerações finais.

4.1 OBJETIVOS DA AVALIAÇÃO

A avaliação experimental tem cinco objetivos principais:

- Analisar o desempenho dos componentes de transporte adicionados e modificados no **pAdapt**;
- Analisar o impacto da adaptação no desempenho dos componentes de transporte;
- Comparar o desempenho entre os diferentes protocolos de comunicação do **pAdapt** (inclusive com adaptação) e sistemas de middleware comerciais (gRPC (GO, 2025), Go RPC (RPC-GO, 2025) e RabbitMQ (GO-RABBITMQ, 2025));
- Analisar o impacto da demanda de processamento no **pAdapt** e nos sistemas de middleware comerciais; e
- Analisar o impacto do tamanho do pacote no **pAdapt** e nos sistemas de middleware comerciais.

Para todos os objetivos, as mesmas aplicações cliente-servidor foram implementadas sobre as diferentes configurações do **pAdapt**, bem como dos sistemas de middleware comerciais, i.e., do gRPC, do Go RPC e do RabbitMQ.

O ponto principal da avaliação é o protocolo de transporte e a sua adaptação automática para outros protocolos em tempo de execução. Foram construídas duas aplicações cliente-servidor, que são analisadas nas próximas seções. O estudo foca somente no serviço de trans-

porte do middleware, ou seja na comunicação entre o cliente e o servidor. Todas as aplicações foram implementadas na linguagem Go, e os experimentos foram executados em um *cluster* Docker Swarm de nó único.

4.2 MÉTRICAS, PARÂMETROS E CARGA DE TRABALHO

Uma das métricas utilizadas ao longo dos experimentos foi o tempo de resposta, ou *Round-Trip Time* (RTT), que é medido no lado do cliente e se refere ao tempo decorrido do momento em que o cliente faz uma solicitação e o instante em que ele recebe uma resposta. Para esta métrica, foram medidos o RTT de cada requisição.

Para verificar a utilização dos recursos pelo sistema para suportar os diferentes sistemas de middleware, outras métricas utilizadas são a *utilização de CPU* e a *utilização de memória*. Estas métricas foram medidas a cada 1 segundo.

Os parâmetros do sistema estão definidos na Tabela 3. Como parâmetros do sistema estão as especificações da máquina utilizada para a execução dos experimentos, bem como da plataforma utilizada na conteinerização dos serviços.

Tabela 3 – Parâmetros do Sistema

Parâmetros do sistema	**Valor**
Processador	AMD® Ryzen 7 5800H
Memória	16,0 GiB
Wifi	Off
Sistema Operacional	Ubuntu 24.04
Plataforma de Conteinerização	Docker swarm
Memória reservada para cada contêiner	256M
CPU reservado para cada contêiner	0.4 CPU
Limite de Memória para cada contêiner	256M
Limite de CPU para cada contêiner	0.4 CPU
Sistema Operacional dos contêineres	Debian Buster 11
Versão do go	go1.22.2 linux/amd64
Go Modules	on

Fonte: Elaborado pelo autor (2025)

O ambiente usado para a avaliação é um *cluster* Docker Swarm no qual cada componente (cliente e servidor) é executado em um contêiner separado. Cada contêiner é executado a partir de uma imagem Debian Buster 11 com restrições de limites de memória e memórias

reservadas pré-alocadas em 256 MB de RAM, i.e., os contêineres já iniciam com memórias pré-alocadas, e também iniciam com um limite de memória igual aos valores pré-alocados, evitando assim que sejam alocadas dinamicamente e também que ultrapassem o máximo de RAM, o que pode impactar no desempenho e avaliação dos serviços. Além da memória, os contêineres também contam com quantidade de CPUs limitadas e pré-alocadas em 40% de um core da CPU da máquina hospedeira, também evitando que aloquem mais recursos do que o necessário e impactem na avaliação de desempenho dos serviços.

O *cluster* Docker foi executado em um computador com CPU AMD® Ryzen 7 5800H de 16 GB de memória RAM e utilizando sistema operacional Ubuntu 24.04 LTS. Além disto, a versão da linguagem Go utilizada foi a 1.22, e com a diretiva Go Modules ativada.

Os parâmetros da carga de trabalho escolhidos para os experimentos estão definidos na Tabela 4. Foi criado um tempo entre invocações que obedece uma distribuição normal com média de 200 ms e desvio padrão de 20 ms (10%), isso se faz necessário, pois sem um intervalo entre as invocações o experimento passaria a ser um experimento de carga, e não das funcionalidades e do desempenho. Além disso, esta distribuição visa simular um cenário mais próximo do real onde as requisições não são feitas em intervalos regulares. *Warm-up requests* foram definidas em 100, isso representa o número de requisições que o cliente faz antes de considerarmos que o sistema está em estado estável, i.e., um estado onde o sistema já foi carregado e não está mais alocando recursos para a inicialização. O número de invocações foi definido em 10.000, para que possamos ter uma quantidade suficiente de requisições para avaliar o desempenho do sistema. O tamanho máximo do pacote foi definido em 65500 bytes, para evitar ultrapassar o limite do *Maximum Transmission Unit* (MTU). Por fim, os *Power Settings* da máquina utilizada para execução dos experimentos foram definidos em *Performance*, para que possamos ter o máximo de desempenho.

Tabela 4 – Parâmetros da Carga de Trabalho

Parâmetros da Carga de Trabalho	**Valor**
Tempo entre invocações	Distribuição normal com média 200 ms e desvio padrão de 20 ms (10%)
Número de invocações	10000
Warm-up requests	100
Tamanho máximo do pacote	65500 bytes
Power Settings	*Performance*

Fonte: Elaborado pelo autor (2025)

Todos os experimentos executados seguiram os mesmos parâmetros de sistema e de carga de trabalho, variando apenas nos fatores, que serão descritos na próxima seção.

4.3 FATORES E PROJETO DOS EXPERIMENTOS

Duas aplicações foram escolhidas para a avaliação experimental: uma com alta variação na demanda de processamento mas tamanho de pacote pequeno (*Fibonacci*), e outra com baixa demanda de processamento mas com grande variação de tamanho do pacote (*SendFile*).

Tendo como foco o middleware, na aplicação de Fibonacci, o procedimento remoto invocado pelo cliente ($fibonacci(N)$), calcula recursivamente um número *N* da sequência de Fibonacci passado como parâmetro pelo cliente. Na prática, cada solicitação passa pelo middleware do cliente e do servidor antes de ser executada remotamente. Apesar de simples, a aplicação Fibonacci é de fácil implantação e utiliza todos os componentes de middleware (semelhantes a aplicações mais complexas), requisito fundamental na avaliação. Esta aplicação visa avaliar o desempenho do middleware e seus componentes, sem muita interferência do tamanho do pacote ou do estado da rede. A aplicação Fibonacci foi escolhida por ser uma aplicação simples, que no entanto, demanda grande utilização de processamento para números grandes, simulando uma aplicação com alta demanda de processamento em sua lógica de negócios.

A aplicação *SendFile* é uma aplicação que envia um arquivo de um cliente para um servidor, e foi escolhida por ser uma aplicação que, além de utilizar todos os componentes do middleware, envolve a transferência de arquivos, de diferentes tamanhos, e que pode ser utilizada para avaliar o desempenho do middleware em relação ao tamanho do pacote, e o envio dos pacotes pela rede de forma fragmentada.

Os fatores dos experimentos foram definidos de acordo com a Tabela 5. Como Camada de Transporte foram colocados todos os protocolos de comunicação disponíveis no **pAdapt** (TCP, TLS, go RPC, QUIC, HTTP, HTTPS, HTTP/2) e também os sistemas de middleware comerciais Go RPC, gRPC e RabbitMQ, identificados na tabela com o prefixo "E_" de "Externo", para indicar que não fazem parte do **pAdapt**.

A adaptação também foi definida como um fator, podendo ser ativada ou não. Para os experimentos, a adaptação é realizada sempre a cada intervalo de tempo, independente do motivo, o foco do experimento é em como adaptar e se esta adaptação compromete o desempenho do middleware, com isso, outro fator é o tempo de adaptação, que pode ser

ativado a cada 2 minutos ou 5 minutos. A combinação de protocolos da adaptação é um outro fator, podendo ser TCP+TLS, RPC+HTTP, TLS+HTTP2.

Por fim, o último fator é o *InputSet*, que é a junção da aplicação executada com sua variação, podendo ser F2, F11 e F38 para a aplicação de Fibonacci, e I36, I2k, I4k para a aplicação *SendFile*.

Os números do *InputSet* do sistema de Fibonacci representam a sequência de Fibonacci que o cliente envia para o servidor calcular. Estes números podem assumir os níveis F2 com número de Fibonacci 2 (baixa demanda de processamento), F11 para o número de Fibonacci 11 (uma demanda com um custo um pouco maior mas sem exigir muito dos processadores) ou F38, representando o número de Fibonacci 38 (alta demanda de processamento), em todos os casos com um pequeno tamanho de carga útil. Na prática, quando *N=2*, o tempo de negócio (tempo para calcular o Fibonacci) é menor que o tempo do middleware (tempo que a requisição/resposta passa dentro do middleware). No caso *N=38*, o tempo do middleware torna-se menor que o tempo do negócio.

Já os números do fator *InputSet* do sistema *SendFile* representam a resolução da imagem enviada. Este fator pode assumir os valores I36, representando uma imagem de 36x36 pixels, resolução típica de pequenos ícones, enviados em um único pacote. O valor I2k, representando uma imagem de resolução 2k, i.e., 2048x1080 pixels, típico de uma foto de boa qualidade. E o valor I4k representando uma imagem de resolução 4k, que possui 3840x2160 pixels e é utilizado em imagens de ótima resolução.

Tabela 5 – Fatores

Fatores	**Níveis**
Camada de Transporte	UDP / TCP / TLS / Go RPC / QUIC / HTTP / HTTPS / HTTP2 / E_RPC / E_RabbitMQ / E_gRPC
Adaptação	On / Off
Tempo de adaptação	2m / 5m
Combinação de Protocolos	TCP+TLS / RPC+HTTP / TLS+HTTP2
InputSet	F2 / F11 / F38 / I36 / I2k / I4k

Fonte: Elaborado pelo autor (2025)

Para executar os experimentos foram criadas imagens Docker com o *build* realizado a partir de arquivos de configuração Dockerfile e armazenadas no Docker Hub, uma imagem para a aplicação servidora e outra imagem para a aplicação cliente. Para garantir a integração correta

entre os contêineres, foi criado também um arquivo de *docker-compose*[1] (Apêndice A), que é um arquivo no formato YAML que define uma *Stack* no Docker Swarm. Uma *Stack* é um conjunto de serviços que são implantados juntos, e que compartilham configurações, como redes e volumes. O arquivo de *docker-compose* foi utilizado para definir os serviços, as redes e valores dos fatores que seriam utilizados nos experimentos.

Para monitorar os experimentos foram criados *Scripts* de Monitoramento e *Profiling* baseadas na API do Docker. Estes *scripts* foram executados em paralelo com os experimentos, sem necessidade de instrumentação do código, que poderia afetar o desempenho, e são os responsáveis pela coleta das métricas de utilização de CPU e de memória dos contêineres.

Todo o processo de avaliação experimental ocorreu sem intervenção manual, i.e., os experimentos foram executados de forma automatizada, e os resultados foram coletados e armazenados em arquivos *Comma-Separated Values* (CSV) para posterior análise. O algoritmo dos *Scripts* de Automação se baseou em um *loop* pelos diferentes níveis dos fatores, e para cada combinação de fatores foi executado um experimento, totalizando 102 experimentos.

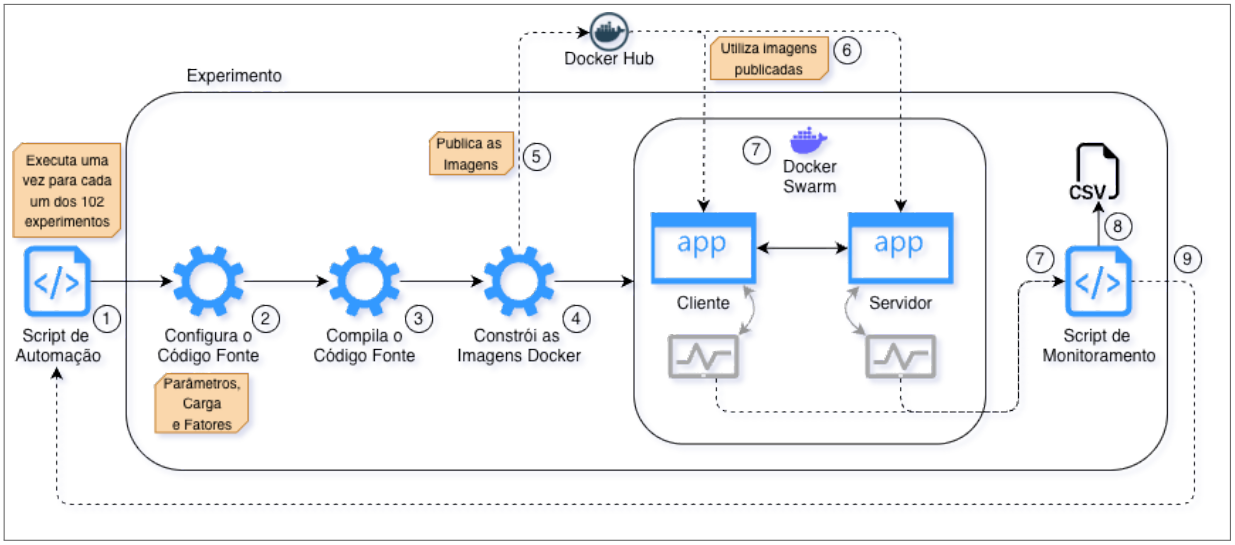
Como parte de cada experimento, foram construídas as imagens Docker e o *docker-compose*, o arquivo de configuração da *Stack*, que continha os fatores a serem analisados no experimento em específico. Na sequência, as imagens foram enviadas para o *Docker Hub* a fim de possibilitar a reprodução dos experimentos. Em seguida, os contêineres foram iniciados no *cluster* Docker Swarm, e os *scripts* de monitoramento e *profiling* foram executados em paralelo. Por fim, com o término da execução das aplicações, os resultados foram coletados e armazenados em arquivos CSV em uma pasta específica do experimento. Todas as informações do experimento, como serviços, contêineres, imagens, redes e fatores são então removidas do *cluster* Docker Swarm para possibilitar a execução de novos experimentos.

A Figura 7 representa a arquitetura e o fluxo de execução dos experimentos.

Conforme Figura 7, os experimentos inciam com a execução do *Script* de Automação no passo 1. Este *script* é responsável então por configurar o Código Fonte de acordo com os parâmetros, carga e fatores do experimento a ser executado no passo 2. Em seguida, no passo 3, o *script* efetua a compilação do Código Fonte. O passo 4 é a construção das imagens Docker, seguidas da publicação das imagens no *Docker Hub* no passo 5. Com as imagens publicadas, o passo 6 é a utilização destas imagens enviadas iniciar o experimento. No passo 7, o *Docker Swarm* inicia os contêineres com as imagens publicadas. Com isso, o experimento começa. Paralelamente ao experimento, ainda no passo 7, o *Script* de Monitoramento e *Profiling* é

[1] Código Fonte dos experimentos disponível em: https://github.com/gfads/midarch

Figura 7 – Arquitetura dos Experimentos



Fonte: Elaborado pelo autor (2025)

iniciado para coletar as métricas do experimento. Ao fim da execução do experimento, no passo 8, o *Script* de Monitoramento e *Profiling* gera planilhas em formato CSV com os dados coletados. Por fim, no passo 9, o *Script* de Automação remove todos os serviços, contêineres, imagens e redes do experimento do *Docker Swarm*, dando início ao próximo ciclo de experimento.

4.4 SOLUÇÃO EM AÇÃO

Para os experimentos foram criadas duas aplicações cliente-servidor: uma aplicação de Fibonacci e uma aplicação de envio de arquivos (*SendFile*). Cada uma contendo dois executáveis, um para o cliente e outro para o servidor. Ambas as aplicações utilizam o **pAdapt** como middleware de comunicação entre o cliente e o servidor.

O Código Fonte 13 demonstra a implementação do servidor de Fibonacci.

Código Fonte 13 – Fibonacci Server

```
func main() {
    args := make(map[string]messages.EndPoint)
    args["srh"] = messages.EndPoint{Host: "0.0.0.0", Port: "1314"}
    fe := frontend.NewFrontend()
    fe.Deploy(frontend.DeployOptions{FileName: "FibonacciDistributedServerMid.
        madl", Args: args, Components: map[string]interface{}{
        "FibonacciInvoker": &middleware.FibonacciInvoker{},
    }})
```

```
    intervalBetweenInjections, _ := strconv.Atoi(shared.
        EnvironmentVariableValueWithDefault("INJECTION_INTERVAL", "45"))
    evolutive.EvolutiveInjector{}.StartEvolutiveProtocolInjection("srhhttp2", "
        srhtls", time.Duration(intervalBetweenInjections)*time.Second)
}
```

Fonte: Elaborado pelo autor (2025)

As linhas 2 e 3 definem os parâmetros que são levados em consideração ao se subir o servidor do **pAdapt**, como o endereço IP, representado por "0.0.0.0" para indicar que recebe requisições de qualquer endereço e a porta onde a aplicação irá escutar as requisições. As linhas 4 a 7 inicializam o middleware do **pAdapt** com os parâmetros definidos anteriormente. A inicialização também indica como é o deploy do middleware, informando a configuração do middleware (arquivo mADL), e adicionando um componente novo ao middleware, que não está implementado no **pAdapt**, o FibonacciInvoker. Para que o **pAdapt** encontre o novo componente a ser extendido no middleware, é necessário que a variável de ambiente MIDARCH_BUSINESS_COMPONENTS_PATH aponte para o diretório onde o componente está implementado.

Por fim, as linhas 8 e 9 iniciam um injetor de adaptações. O objetivo deste injetor é simular uma necessidade de adaptação, forçando o **pAdapt** a adaptar os protocolos de comunicação a cada intervalo de tempo definido na variável de ambiente INJECTION_INTERVAL. Este injetor também é quem define quais os protocolos de comunicação que serão utilizados na adaptação.

O Código Fonte 14 demonstra a implementação do cliente de Fibonacci.

Código Fonte 14 – Fibonacci Client

```
func main() {
    args := make(map[string]messages.EndPoint)
    args["crh"] = messages.EndPoint{Host: shared.CALCULATOR_HOST, Port: shared.
        CALCULATOR_PORT}
    fe := frontend.NewFrontend()
    fe.Deploy(frontend.DeployOptions{
        FileName: "FibonacciDistributedClientMid.madl",
        Args:     args,
        Components: map[string]interface{}{
            "FibonacciProxy": &fibonacciProxy.FibonacciProxy{},
        }})
    proxyConfig := generic.ProxyConfig{Host: shared.CALCULATOR_HOST, Port: shared
        .CALCULATOR_PORT}
    fibonacci := &fibonacciProxy.FibonacciProxy{}
```

```
    fibonacci.Configure(proxyConfig)
14  for x := 0; x < SAMPLE_SIZE; x++ {
        ok := false
16      for !ok {
            t1 := time.Now()
18          r := fibonacci.F(shared.FIBONACCI_PLACE)
            t2 := time.Now()
20          duration := t2.Sub(t1)
            if r != 0 {
22              ok = true
                log.Printf(";ok;%d;%f;%d\n", x+1, duration, r)
24          } else {
                log.Printf(";error;%d;%f;%d\n", x+1, duration, r)
26          }
            time.Sleep(shared.GetIntervalBetweenInvocations())
28      }
    }
30  }
```

Fonte: Elaborado pelo autor (2025)

As linhas 2 e 3 definem os parâmetros que são levados em consideração ao inicializar o middleware com **pAdapt**, como o endereço IP e a porta do servidor. As linhas 4 a 10 inicializam o middleware do **pAdapt** com os parâmetros definidos anteriormente. A inicialização também indica como é o deploy do middleware, informando a configuração do middleware (arquivo mADL), e adicionando um componente novo ao middleware, que não está implementado no **pAdapt**, o FibonacciProxy. Assim como no servidor, para que o **pAdapt** encontre o novo componente a ser extendido no middleware, é necessário que a variável de ambiente MIDARCH_BUSINESS_COMPONENTS_PATH aponte para o diretório onde o novo componente está implementado.

As linhas 11 a 13 iniciam o proxy de fibonacci, que é responsável por fazer a comunicação com o servidor. O proxy encapsula a lógica de comunicação, permitindo chamadas RPC para o servidor através do middleware do **pAdapt**.

Por fim, as linhas 14 a 29 implementam a lógica do experimento no lado do cliente. O cliente percorre um *loop* de acordo com número definido na variável de ambiente SAMPLE_SIZE. Para cada iteração do *loop*, as linhas 17 a 20 realizam uma requisição ao servidor para calcular o número Fibonacci, passando como parâmetro o número definido na variável de ambiente FIBONACCI_PLACE, e calculando o tempo de resposta da requisição. Na sequência, as linhas

21 a 26 armazenam o tempo de resposta em um arquivo CSV. Finalizando então na linha 27 com um *sleep* do tempo entre invocações, antes que se inicie a próxima iteração do *loop*.

A aplicação *SendFile* segue a mesma estrutura da aplicação Fibonacci, mas com a lógica de negócio diferente, i.e., com um proxy e um invoker específicos para o envio de arquivos.

4.5 RESULTADOS E ANÁLISE DOS RESULTADOS

Conforme mencionado anteriormente, o primeiro objetivo da avaliação experimental foi comparar o desempenho dos diferentes componentes adicionados e modificados do **pAdapt**. Para isso, as aplicações cliente-servidor foram executadas sobre instâncias do **pAdapt** configuradas com diferentes mecanismos de transporte, aqui classificados em protocolos seguros (TCP+TLS, QUIC, HTTPS e HTTP/2) e não seguros (UDP, TCP, RPC e HTTP).

Os dados obtidos através do *profiling* e monitoramento dos experimentos geraram CSVs com os dados ao longo do tempo, e a partir destes dados foram gerados *boxplots* para cada uma das diferentes métricas de desempenho, como RTT, utilização de CPU e memória. A partir destes *boxplots* foi possível realizar a comparação entre os diferentes protocolos de comunicação do **pAdapt**, e também entre os protocolos do **pAdapt** e os middleware comerciais gRPC, Go RPC e RabbitMQ.

Para a geração dos *boxplots*, foi utilizado a linguagem Python, a biblioteca *pandas* para manipulação dos dados, e as bibliotecas *matplotlib* e *seaborn* para a geração dos gráficos. Para análise dos resultados foram aplicados testes T-Test para verificar diferenças entre os resultados. Nas figuras dos *boxplots* abaixo, os protocolos do **pAdapt** sem adaptação são representados pelos seus nomes e na cor azul, os protocolos do **pAdapt** com adaptação são representados pelos nomes concatenados de ambos os protocolos utilizados na adaptação, seguidos do tempo entre cada adaptação, e estão na cor verde. Já os sistemas que utilizam middleware comerciais são representados com o prefixo "E_" de "Externo", e estão representados na cor cinza.

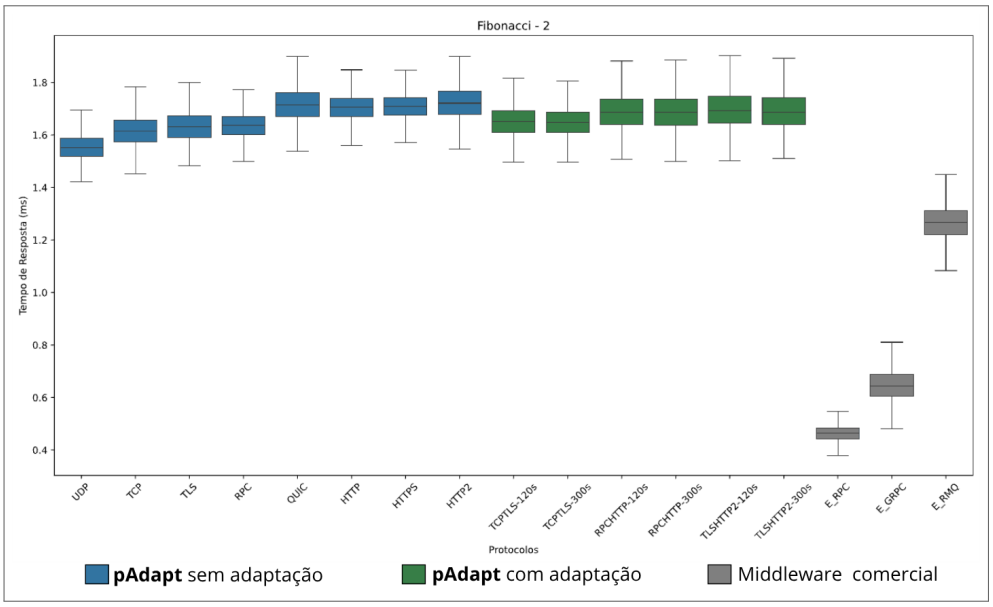
A seguir, são discutidos os principais achados e suas implicações para o desempenho do sistema.

4.5.1 Desempenho dos protocolos de comunicação do pAdapt

Analisando os tempos de resposta das aplicações Fibonacci de 2 e *SendFile* com arquivo de tamanho 36x36, foi possível observar que: quando o tempo da regra de negócio é baixo, a quantidade dos pacotes, a utilização de CPU e a utilização de memória são similares entre diferentes regras de negócio. Em outras palavras, o RTT é influenciado pelo tamanho do pacote, pela utilização de memória e pela utilização de CPU gerados em decorrência das necessidades e demandas das regras de negócio.

A Figura 8 mostra o RTT dos diferentes protocolos de comunicação para a aplicação Fibonacci com número 2. A Figura 9 mostra o RTT dos diferentes protocolos de comunicação para a aplicação *SendFile* com arquivo de 36x36 pixels. O tamanho do pacote do envio de imagens é ligeiramente maior que o tamanho do pacote do envio de números da sequência de Fibonacci. Além disso, no envio de imagens ainda há o tempo necessário para salvar a imagem em disco após o recebimento da imagem e isso pode ser observado nos *boxplots*, onde o RTT para a aplicação *SendFile* é ligeiramente maior que o RTT para a aplicação Fibonacci. No entanto, pode ser verificado que a consistência dos resultados é similar, com os diferentes protocolos se mantendo em posições similares nos *boxplots*.

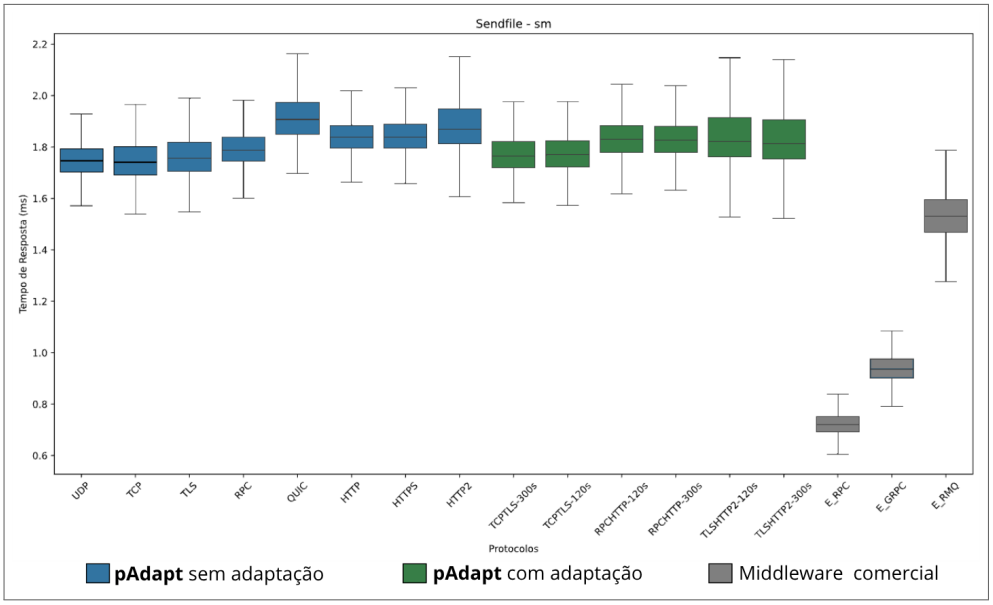
Figura 8 – Experimento Fibonacci 2: Boxplot Protocolos vs RTT



Fonte: Elaborado pelo autor (2025)

Outro ponto importante a ser destacado é a magnitude dos tempos de resposta, medida em milisegundos. O protocolo mais lento faz o RTT do *Experimento SendFile 36x36* com

Figura 9 – Experimento SendFile 36x36: Boxplot Protocolos vs RTT



Fonte: Elaborado pelo autor (2025)

uma mediana de apenas 1.9 ms. Neste mesmo experimento, a diferença entre as médias dos tempos de resposta do protocolo mais lento para o protocolo mais rápido é de 63%, i.e, de apenas 1.2 ms, expressa pela diferença entre o QUIC e a implementação RPC comercial.

4.5.2 Impacto da adaptação no pAdapt

Outro resultado que fica evidente nas Figuras 8 e 9, é que não há diferença de desempenho entre os protocolos de comunicação do **pAdapt** com e sem adaptação.

Analisando estatisticamente os resultados do *Experimento Fibonacci 2*, não foi encontrada uma diferença estatisticamente significativa entre os tempos de resposta da execução do TLS sem adaptação e a execução do TLS com adaptação (TCPTLS-300s). A análise do teste T indicou que a diferença não foi significativa, com um valor do *p-value* de 0.73 e intervalo de confiança de 95%.

Analisando estatisticamente os resultados do *Experimento SendFile 36x36*, não foi encontrada uma diferença estatisticamente significativa entre os tempos de resposta da execução do TLS sem adaptação e a execução do TLS com adaptação (TCPTLS-300s). A análise do teste T indicou que a diferença não foi significativa, com um valor do *p-value* de 0.23 e intervalo de confiança de 95%.

Consequentemente é possível chegar a conclusão de que a adaptação automática de pro-

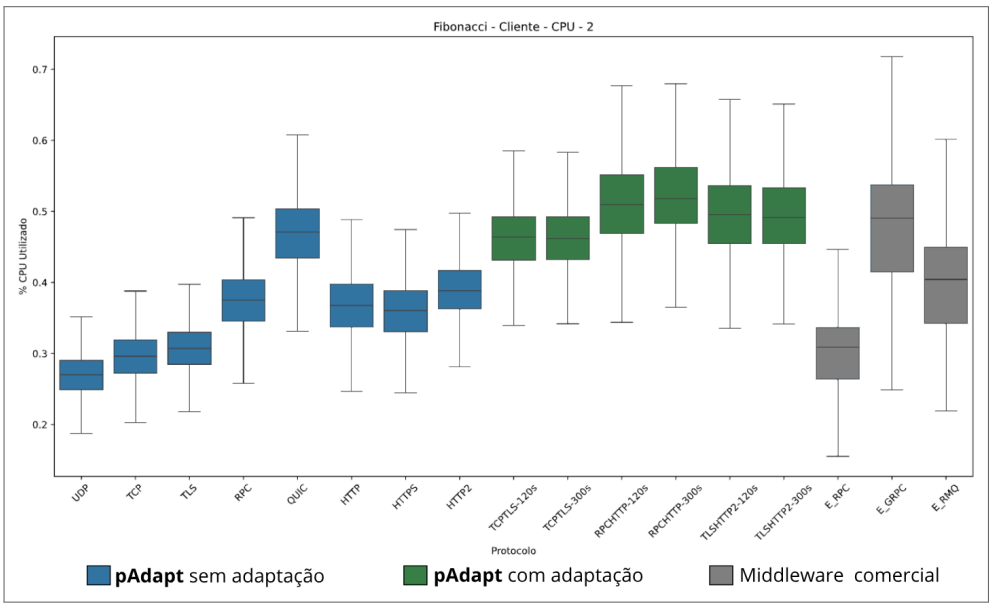
tocolos de comunicação do **pAdapt** não impacta significativamente no RTT, e os protocolos de comunicação com adaptação mantém RTTs similares aos protocolos sem adaptação. Evidenciando assim que a adaptação automática de protocolos de comunicação do **pAdapt** é eficiente com relação ao impacto causado no RTT.

4.5.3 Desempenho dos protocolos de comunicação do pAdapt e sistemas de middleware comerciais

Ainda com relação ao RTT para aplicações com pouco uso de memória e CPU, e pacotes de tamanho pequeno, observados nas Figuras 8 e 9, é possível perceber que os sistemas de middleware comerciais (E_RPC, E_GRPC e E_RMQ) têm melhor desempenho do que os protocolos de comunicação do **pAdapt**. No entanto, esta diferença é de, no máximo, aproximadamente 1 ms.

Analisando por outras perspectivas, nestes mesmos experimentos, na Figura 10, é possível perceber que no **pAdapt** a utilização de CPU para UDP, TCP e TLS têm um desempenho melhor que os sistemas de middleware comerciais. E que mesmo com a adaptação do **pAdapt** em funcionamento, o desempenho do **pAdapt** ainda supera o middleware comercial gRPC.

Figura 10 – Experimento Fibonacci 2: Boxplot da CPU do Cliente x Protocolos



Fonte: Elaborado pelo autor (2025)

Com isso podemos concluir que, tanto os protocolos implementados, quanto o **pAdapt** são eficientes em termos de utilização de CPU no cliente, mesmo utilizando adaptação. Sendo

assim, para clientes que tenham restrição de CPU, o **pAdapt** se mostra como uma opção interessante quando comparado aos sistemas de middleware comerciais analisados.

4.5.4 Aplicações com diferentes demandas de processamento

Aplicações com alta demanda de processamento, como no experimento de Fibonacci de posição 38, apresentado na Figura 11, têm resultados de RTT muito estáveis para diferentes protocolos, onde o tempo gasto de execução da regra de negócio é muito maior que o necessário para envio das mensagens.

Analisando estatisticamente os resultados do *Experimento Fibonacci 38*, não foi encontrada uma diferença estatisticamente significativa entre os tempos de resposta da execução do RPC sem adaptação e a execução do RPC com adaptação para HTTP (RPCHTTP-120s). A análise do teste T indicou que a diferença não foi significativa, com um valor do *p-value* de 0.21 e intervalo de confiança de 95%.

Também não foi encontrada uma diferença estatisticamente significativa entre os tempos de resposta da execução do TLS sem adaptação e a execução do TLS com adaptação para HTTP2 (TLSHTTP2-120s). A análise do teste T indicou que a diferença não foi significativa, com um valor do *p-value* de 0.78 e intervalo de confiança de 95%.

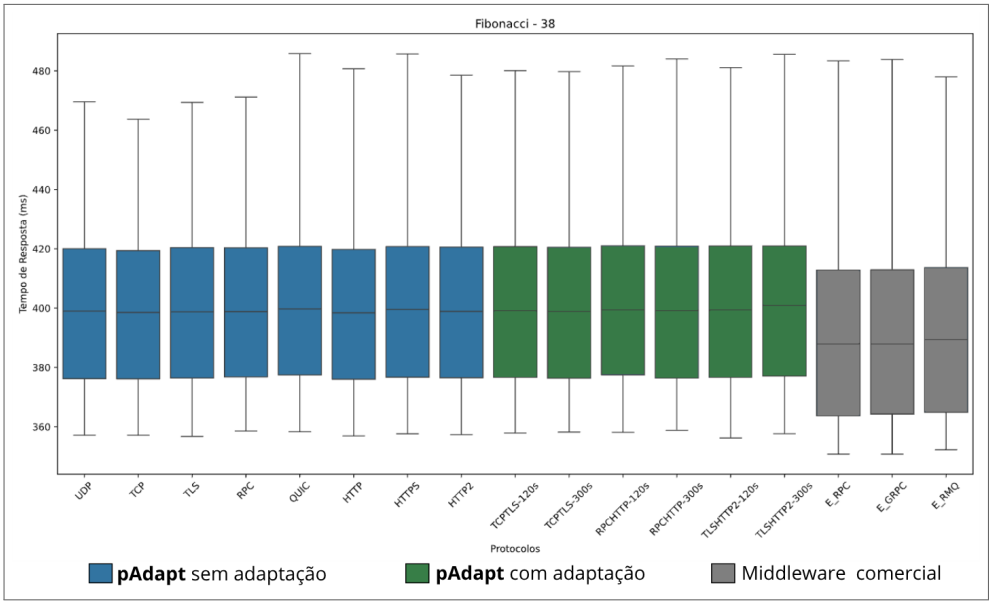
Neste caso, em aplicações com alta demanda de processamento, não há diferença significativa no RTT entre os protocolos de comunicação sem adaptação e os protocolos com adaptação automática do **pAdapt**.

Já aplicações com baixa demanda de processamento (Figura 8), têm resultados de RTT bem variados para os diferentes protocolos, onde o tempo de execução da regra de negócio é muito menor que o necessário para envio das mensagens. Este comportamento torna evidente o impacto do protocolo de transporte no RTT. No entanto, apesar da variação, a diferença máxima entre a média dos RTT dos protocolos do **pAdapt** é de apenas 0.2 ms (11%) de diferença por mensagem.

4.5.5 Aplicações com diferentes tamanhos de pacote

Aplicações com tamanho de pacote grandes fazem uso de muitos recursos do middleware. Isso se deve ao fato de que este tipo de aplicação exige que as mensagens sejam enviadas via rede de forma fragmentada, ou através de *stream* de dados. O envio de mensagens desta

Figura 11 – Experimento Fibonacci 38: Boxplot Protocolos vs RTT



Fonte: Elaborado pelo autor (2025)

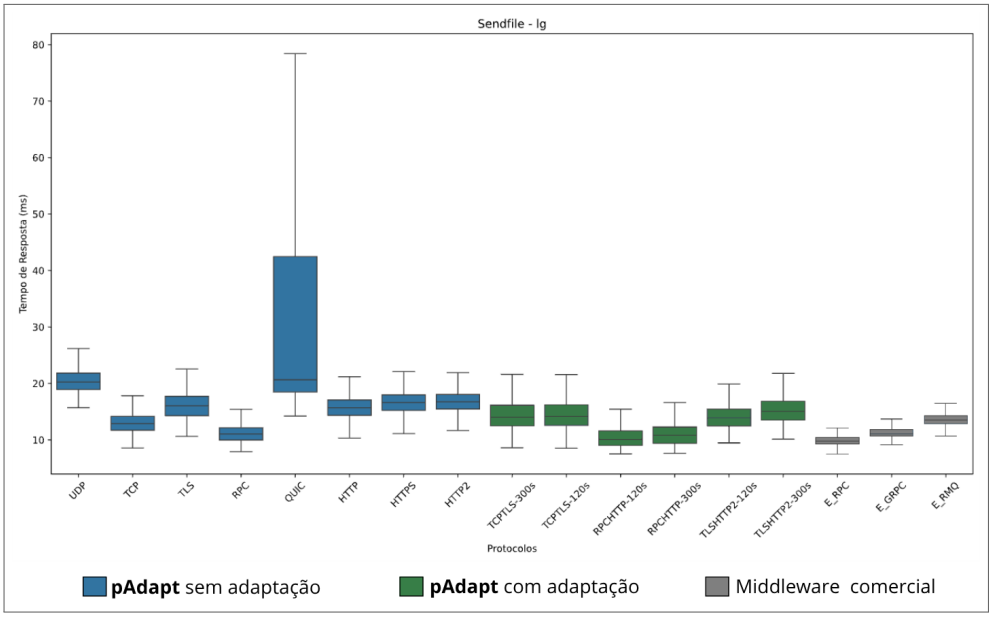
forma implica em chamadas iterativas, o que impacta no RTT. O experimento *SendFile* com arquivo de 4k é um exemplo deste cenário, e seus resultados de RTT por protocolo podem ser vistos na Figura 12.

Para este tipo de aplicação, o **pAdapt** conseguiu se mostrar eficiente, em especial para os protocolos RPC e TCP, onde o RPC teve um desempenho similar ao gRPC e melhor que o RabbitMQ, e o TCP conseguiu se sair melhor que o RabbitMQ.

É possível perceber também que o protocolo QUIC não teve um bom *Round-Trip Time*. Como o QUIC é considerado um protocolo mais eficiente que o HTTP/2, então o problema des desempenho possivelmente se deve à implementação efetuada para o protocolo em questão, que não fez proveito da multiplexação provida pelo QUIC, o que poderia melhorar o desempenho do protocolo.

Outro ponto importante a ser destacado é que o **pAdapt** se mostrou eficiente na adaptação automática de protocolos de comunicação, mesmo para aplicações com tamanho de pacote grande. A Figura 12 mostra que o RTT dos protocolos com adaptação é muito próximo dos protocolos sem adaptação, i.e., a escolha do protocolo de transporte impacta mais no desempenho do que a própria adaptação. Isso indica não somente que o **pAdapt** é eficiente, mas também que seus mecanismos de adaptação são efetivos para aplicações que buscam aproveitar a vantagem de cada protocolo em diferentes cenários, e.g., um sistema pode começar utilizando o protocolo TCP, que tem um RTT baixo, e depois por alguma necessidade adaptar

Figura 12 – Experimento SendFile 4k: Boxplot Protocolos vs RTT



Fonte: Elaborado pelo autor (2025)

para o protocolo TCP com TLS, que é mais seguro, mas com um RTT maior, sem impactar significativamente no desempenho do sistema.

4.6 CONSIDERAÇÕES FINAIS

Este capítulo apresentou a avaliação experimental da solução proposta, o **pAdapt**, aplicado ao gMidArch. Inicialmente foram apresentados os objetivos da avaliação experimental. Em seguida, ele mostrou a metodologia utilizada para a realização dos experimentos, incluindo a definição de métricas, parâmetros, carga de trabalho, fatores e o projeto dos experimentos. Foram apresentados como as aplicações de servidores e clientes foram implementados. E finalmente, foram apresentados e analisados os resultados obtidos de acordo com cada objetivo.

5 TRABALHOS RELACIONADOS

A construção de frameworks de middleware é uma área consolidada, mas ainda cercada por desafios significativos, especialmente quando se busca oferecer suporte à adaptação dinâmica e a múltiplos protocolos de comunicação. Implementar esse tipo de solução requer abstrações complexas, que envolvem desde funcionalidades internas do middleware até estratégias de serialização e mecanismos de comunicação. Projetos pioneiros de frameworks de middleware, como Quarterware (Singhai; Sane; Campbell, 1998), PolyORB (VERGNAUD et al., 2004) e Arcademis (PEREIRA et al., 2006), foram trabalhos seminais, embora não tenham incorporado capacidades adaptativas nem compatibilidade com múltiplos protocolos.

Outros trabalhos mais recentes buscaram evoluir estas soluções em diferentes direções. O Man4Ware (AL-JAROODI; MOHAMED; JAWHAR, 2018), por exemplo, é um framework de middleware baseado em uma arquitetura orientada a serviços. Ele adota uma abordagem modular, composta por diversos serviços integrados. Assim como o gMidArch, o Man4Ware permite que os desenvolvedores foquem na implementação do código de negócio, delegando as demais funcionalidades ao próprio framework. No entanto, o Man4Ware não oferece suporte à seleção do protocolo de transporte, ao contrário do **pAdapt**, que disponibiliza diversos protocolos, adaptáveis, e facilmente configuráveis. Isso significa que os desenvolvedores não têm controle sobre como o middleware realiza a comunicação, tampouco podem modificar o protocolo utilizado em tempo de execução.

Já o Cilia (Lalanda; Morand; Chollet, 2017) é um Middleware de Mediação Autônoma que utiliza componentes específicos para lidar com diferentes protocolos de comunicação e possibilitar adaptações, de maneira semelhante ao gMidArch. Contudo, o Cilia se limita a fornecer estruturas auxiliares, como uma base de conhecimento com informações em tempo de execução e pontos de extensão para inclusão de código. Cabe aos desenvolvedores que utilizam o middleware a implementação dos mecanismos de adaptação desejados. Estes mecanismos não estão disponíveis previamente no Cilia, ao contrário do **pAdapt**, que já implementa o mecanismo de adaptação entre diferentes protocolos. Além disso, diferentemente do gMidArch com **pAdapt**, o Cilia não é um middleware de uso geral, sendo projetado com foco na integração de sistemas ciber-físicos na gestão de indústrias inteligentes.

Projetado para atender aos requisitos típicos de aplicações cooperativas, o CoServices (Xie; Li; Zhao, 2013) é um framework de middleware baseado em *Web Service*. Sua arquitetura é

composta por módulos padronizados que fornecem funcionalidades como gerenciamento de sessões e de dados compartilhados, além da possibilidade de desenvolvimento de módulos específicos. Embora utilize *Web Service* como base para comunicação, o CoServices também permite o transporte de mensagens por meio dos protocolos UDP ou HTTP. O **pAdapt**, por sua vez, amplia esse suporte ao incluir protocolos adicionais e mecanismos de adaptação destes protocolos. O CoServices também não conta com adaptação em tempo de execução, seu foco é disponibilizar um conjunto de serviços para o desenvolvedor utilizar.

A discussão sobre o uso de múltiplos protocolos de comunicação em frameworks de middleware adaptativos também é abordada por Brinkschulte et al. (BRINKSCHULTE, 2019), que propõe uma arquitetura de middleware adaptativa voltada a redes ciber-físicas. Nesse trabalho, os autores justificam a importância do suporte a diferentes protocolos como forma de atender aos requisitos de qualidade de serviço. Assim como o gMidArch, essa arquitetura é baseada no modelo MAPE-K. No entanto, o **pAdapt** se diferencia da arquitetura de Brinkschulte, pois Brinkschulte não realiza adaptações de protocolos de comunicação, mas sim disponibiliza diferentes protocolos ao mesmo tempo. Já no **pAdapt** os protocolos são trocados conforme necessidade, e sincronizados de forma orquestrada entre servidor e clientes.

Sangeeta et al. apresentam um middleware adaptativo voltado à integração de sistemas legados em redes elétricas inteligentes (*smart grids*) (SANGEETA et al., 2023). O framework propõe a interoperabilidade entre protocolos tradicionais de automação e novas tecnologias de comunicação, com suporte a múltiplos protocolos e adaptação conforme mudanças na rede elétrica. Os autores enfatizam a integração transparente (*"seamless integration"*). No entanto, o middleware proposto por Sangeeta foca somente em tradução de protocolos legados para protocolos mais atuais, gerando um *overhead* da utilização de dois protocolos adicionada a uma tradução a cada envio de mensagem. Enquanto isso, o **pAdapt** permite a utilização do protocolo desejado, e a sua troca, sem necessidade de tradução, ou seja, cliente e servidor conversando no mesmo protocolo.

O *Adaptive Ubiquitous Middleware* (AUM) (PRADEEP; KRISHNAMOORTHY; VASILAKOS, 2021) é uma proposta voltada a ambientes IoT, com suporte explícito a múltiplos protocolos de comunicação e foco em adaptação consciente de contexto. Ele atua como um ponto de integração entre dispositivos e aplicações, utilizando uma ponte de múltiplos protocolos, e.g., TCP, HTTP e *Constrained Application Protocol* (CoAP), que podem ser escolhidos dinamicamente conforme o contexto do sistema. AUM adapta sua configuração em tempo de execução, levando em consideração fatores como localização, tipo de dispositivo, restrições de

rede e requisitos da aplicação. Embora sua aplicação principal esteja em ambientes ubíquos e IoT, ele compartilha com o gMidArch a capacidade adaptativa em tempo de execução e a interoperabilidade com diferentes protocolos. No entanto, AUM não faz uso de métodos formais para garantir uma integração mais confiável entre seus componentes.

O Hetero-Genius (ELHABBASH et al., 2023) é uma arquitetura de middleware voltada à composição automática e mediação entre sistemas IoT heterogêneos. Sua proposta visa conectar, em tempo de execução, diversos dispositivos e serviços com diferentes protocolos de comunicação, oferecendo um mecanismo de composição baseado em contexto. A arquitetura permite ao desenvolvedor definir fluxos abstratos de tarefas, sendo responsável por localizar, selecionar e integrar dinamicamente serviços concretos disponíveis na rede. Embora seu foco principal seja em aplicações IoT, como veículos conectados, seu suporte a múltiplos protocolos e adaptação dinâmica de fluxos o tornam relevante para comparação com o gMidArch. No entanto, o Hetero-Genius não mantêm sistemas distribuídos se comunicando e adaptando protocolos de comunicação sem paradas, ele simplesmente trabalha com fluxos pré-definidos, que a depender do fluxo podem enviar mensagens com diferentes protocolos.

Cavalcanti e Rosa propõem o *Middleware Extendify* (MEx) (CAVALCANTI; ROSA, 2024) como uma plataforma para construção de middleware IoT adaptáveis. O MEx oferece suporte ao protocolo de comunicação *Message Queuing Telemetry Transport* (MQTT), além de mecanismos para adaptação em tempo real. Seu diferencial está na possibilidade de ajustes dinâmicos de funcionalidades e parâmetros do sistema com base em condições contextuais, utilizando estratégias de adaptação como ajustes reativos e evolutivos. O middleware também introduz uma linguagem de descrição própria, denominada *Python-based Architecture Description Language* (pADL), para definir seus componentes, similar ao mADL do gMidArch. No entanto o MEx é limitado ao protocolo MQTT, enquanto o **pAdapt** habilita o suporte a diversos protocolos de comunicação no gMidArch, além de permitir a adaptação entre eles.

O PolyglIoT (CABRAL et al., 2024), é uma arquitetura de middleware que permite tradução entre múltiplos protocolos de comunicação em ambientes heterogêneos. A arquitetura implementa um serviço dinâmico de tradução entre protocolos, permitindo interações entre tecnologias como MQTT, AMQP, Kafka e *Data Distribution Service* (DDS). O PolyglIoT permite ativação e desativação de tradutores em tempo de execução e garante propriedades de *Quality of Service* (QoS) durante a tradução. Seu foco está em promover interoperabilidade em sistemas distribuídos heterogêneos, com aplicação especial em contextos de IoT e automação. Apesar de utilizar propriedades de QoS para tentar melhorar o desempenho, o uso de

traduções de protocolos gera um *overhead* da utilização dos protocolos traduzidos somada ao processamento necessário para a tradução a cada envio de mensagem. Por outro lado, o **pAdapt** pode habilitar que qualquer sistema faça a alteração do protocolo de transporte utilizado na comunicação apenas mudando a sua configuração, i.e., eliminando qualquer necessidade de tradução e consequentemente seu *overhead*.

Outro exemplo de framework de middleware com mais de um protocolo de transporte é o Gorilla (GORILLA, 2025), um framework para a linguagem Go que, apesar de oferecer suporte somente a comunicação via HTTP e *WebSocket*, também permite utilizar os fundamentos do RPC para o desenvolvimento de sistemas, mas trafegando os dados através do HTTP. Embora o Gorilla permita a escolha do protocolo de transporte, ele não possui mecanismos de adaptação em tempo de execução, e nem permite que os protocolos sejam alterados sem mudanças no código fonte, assim como é no **pAdapt**. Isso significa que, uma vez escolhido o protocolo, não há suporte para mudanças dinâmicas durante a execução do sistema. O Gorilla, apesar de ter o código aberto, também não disponibiliza pontos de extensão para que os desenvolvedores possam implementar novos protocolos de comunicação ou adaptação de protocolos.

Por fim, embora não constitua um framework completo, o pacote RPC do Go (RPC-GO, 2025) permite ao programador escolher entre dois protocolos de comunicação (TCP e HTTP), mas seu escopo limitado a essas duas opções compromete a flexibilidade para aplicações que demandem protocolos adicionais ou comportamentos mais sofisticados de adaptação, pontos que são características do **pAdapt**.

A Tabela 6 resume e compara os frameworks de middleware mencionados com base em características fundamentais como suporte a múltiplos protocolos, adaptação em tempo de execução, uso da arquitetura MAPE-K, utilização de métodos formais, extensibilidade e aplicação de escopo geral ou específica.

Tabela 6 – Comparação entre o gMidArch e frameworks relacionados

Framework	**Múltiplos protocolos**	**Adaptação em *runtime***	**MAPE-K**	**Extensível**	**Adaptação de Protocolos**
Quarterware	Não	Não	Não	Sim	Não
PolyORB	Não	Não	Não	Sim	Não
Arcademis	Não	Não	Não	Sim	Não
Man4Ware	Não	Não	Não	Sim	Não
Cilia	Sim	Sim	Não	Sim	Não
CoServices	Sim (UDP, HTTP)	Não	Não	Sim	Não
Brinkschulte et al.	Sim	Sim	Sim	Sim	Não
Sangeeta et al.	Sim (protocolos legados/modernos)	Sim	Não	Sim	Não
AUM	Sim (TCP, HTTP, CoAP)	Sim	Não	Sim	Não
Hetero-Genius	Sim (implícito via composição)	Sim	Não	Sim	Não
MEx	Não (MQTT)	Sim	Sim	Sim	Não
PolygIIoT	Sim (MQTT, AMQP, Kafka, DDS)	Sim	Não	Sim	Não
Gorilla	Sim (HTTP, *WebSocket*)	Não	Não	Não	Não
RPC do Go	Sim (TCP, HTTP)	Não	Não	Não	Não
gMidArch+pAdapt	**Sim (UDP, TCP, TCP sobre TLS, RPC, QUIC, HTTP/1.1, HTTPS e HTTP/2)**	**Sim**	**Sim**	**Sim**	**Sim**

Nota: O uso de negrito na última linha enfatiza as características do gMidArch com **pAdapt** em comparação com os outros frameworks.

Fonte: Elaborado pelo autor (2025)

Como pode ser observado na Tabela 6, o gMidArch, aliado às novas funcionalidades do **pAdapt**, se destaca por reunir em um único framework características fundamentais que aparecem de forma isolada em outras propostas. Sua combinação de suporte a múltiplos protocolos, adaptação em tempo de execução, extensibilidade e principalmente, a adaptação de protocolos em tempo de execução, o posiciona como uma solução robusta e versátil frente aos trabalhos relacionados discutidos neste capítulo.

5.1 CONSIDERAÇÕES FINAIS

Este capítulo apresentou os trabalhos relacionados ao gMidArch com o novo mecanismo de adaptação **pAdapt**. Para isso foi realizada uma análise comparativa entre cada trabalho, identificando as diferenças entre as características de cada um e o gMidArch com **pAdapt**. O capítulo conclui com uma tabela comparativa entre todos os trabalhos relacionados apresentados, resumindo as principais características de cada trabalho relacionado.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este capítulo apresenta as conclusões e contribuições do trabalho, bem como as limitações da solução proposta, e finaliza com sugestões de trabalhos futuros.

6.1 CONCLUSÕES

A revisão dos trabalhos relacionados mostrou que a adaptação em middleware é um tema amplamente discutido e atual, assim como a utilização de múltiplos protocolos de comunicação. No entanto, poucos trabalhos conseguem reunir essas características de maneira integrada. Este trabalho propôs o **pAdapt** como solução para essa lacuna.

O **pAdapt** trouxe um reforço ao gMidArch que possui uma abordagem abrangente, capaz de preencher lacunas deixadas por middleware anteriores, tanto conceituais quanto práticos.

A implementação e a avaliação experimental indicam que o **pAdapt** é aplicável a diferentes cenários, especialmente em cenários onde a flexibilidade de comunicação e a adaptabilidade em tempo de execução são requisitos críticos, como integração entre aplicações distribuídas em ambientes heterogêneos.

O **pAdapt** se traduz nas duas principais contribuições deste trabalho: a primeira sendo apresentar e tornar disponível um mecanismo de adaptação de protocolos de comunicação e a segunda sendo a disponibilização de um framework de middleware adaptativo com vários protocolos de comunicação.

O mecanismo de adaptação criado se destaca pela capacidade de suportar a troca entre múltiplos protocolos de comunicação em tempo de execução, adaptando tanto clientes quanto servidores de maneira orquestrada e sem a perda de informações durante sua adaptação. Isso permite que desenvolvedores de middleware reconfigurem dinamicamente o protocolo de comunicação mais adequado para diferentes cenários da aplicação, sem comprometer o desempenho dos sistemas.

Como uma terceira contribuição foi criado o Fluxo de Adaptação de Protocolos de Comunicação entre o servidor e seus clientes. Um fluxo que permite a adaptação de protocolos em tempo de execução sem a perda de mensagens, e que pode ser implementado por outros middleware para permitir integração entre diferentes tecnologias e linguagens.

Outro ponto importante está relacionado à avaliação do desempenho do middleware, que,

a depender do cenário, demonstrou melhor desempenho quando comparado a sistemas de middleware comerciais existentes.

Estas características promovem a flexibilidade e permitem que sistemas distribuídos sejam reconfigurados dinamicamente. A proposta combina os conceitos da arquitetura MAPE-K com a extensibiliade de um framework aberto e com implementações de diversos protocolos de comunicação, viabilizando a construção de sistemas distribuídos flexíveis e extensíveis.

Por fim, a última contribuição são os *scripts* de monitoramento e *profilling* da avaliação experimental, que foram desenvolvidos para avaliar o desempenho do **pAdapt** em conjunto com o gMidArch e também compará-lo com outros sistemas de middleware amplamente utilizados. Apesar de não estar ligado diretamente ao objetivo do trabalho, os *scripts* de monitoramento e *profilling* são executados em paralelo aos experimentos, e permitem a coleta de dados de memória, processamento, e tempo de execução sem necessidade de instrumentação do código, mesmo em ambientes distribuídos. Com os *scripts* ainda é possível executar grandes sequências de experimentos sem a intervenção humana, o que poderia gerar dados inconsistentes nos experimentos. Estes *scripts* podem ser utilizados em outros experimentos, sendo assim uma contribuição adicional.

6.2 LIMITAÇÕES

Apesar de o **pAdapt** atingir seus objetivos no estado atual, algumas limitações foram identificadas:

- **Consumo de recursos**: devido a natureza do gMidArch, que fica em constante estado de monitoramento, o consumo de CPU e memória do middleware pode ser elevado em alguns cenários, se comparado a sistemas de middleware não adaptativos. No entanto, o monitoramento e as adaptações de protocolo mostraram não ter um impacto significativo no desempenho do sistema, mesmo em cenários com múltiplas adaptações;
- **Variedade de protocolos**: o **pAdapt** suporta diversos protocolos de comunicação, e também é facilmente extensível, mas a falta de protocolos como AMQP, MQTT, CoAP e bluetooth limita a utilização do **pAdapt**em alguns cenários;
- **Motivo da adaptação**: o **pAdapt** atual não considera o motivo da adaptação, ou seja, quando e porquê uma adaptação será iniciada, apenas a forma em que será adaptado e

o melhor momento para que a adaptação identificada seja executada. Isso significa que a decisão de adaptação é baseada em condições predefinidas, sem levar em conta fatores contextuais ou específicos do aplicativo, e.g., memória, consumo de energia e segurança na comunicação, o que melhoraria a efetividade da adaptação; e

- **Adaptação pelo cliente**: o **pAdapt** não permite que o cliente solicite a adaptação do protocolo de transporte. A adaptação é iniciada somente pelo servidor, o que pode limitar a flexibilidade em cenários onde o cliente tem conhecimento de condições específicas que exigem uma mudança de protocolo.

6.3 TRABALHOS FUTUROS

Com a conclusão deste trabalho, surgem várias possibilidades de continuidade de pesquisa e aprimoramento do **pAdapt**.

Apesar de haver sido empregado esforço na melhoria do desempenho do middleware, e mesmo com o desempenho satisfatório apresentado nos experimentos, ainda é possível efetuar melhorias no desempenho. Um dos pontos é através da otimização em alguns protocolos, como o QUIC e o HTTP/2, com a utilização de multiplexação nas mensagens, o que pode melhorar o desempenho em cenários com múltiplas mensagens simultâneas.

Outra possibilidade de continuação do trabalho é a implementação de novos protocolos, como o AMQP, o MQTT, o CoAP e também protocolos bluetooth, o que pode ampliar ainda mais o alcance do **pAdapt** em diferentes domínios de aplicação.

Adicionar novas estratégias de adaptação que utilizem métricas diversas e configuráveis, como memória, CPU, consumo de energia, bateria e segurança. Neste item pode-se evoluir para a escolha de evolução somente para determinados protocolos, ou seja, o desenvolvedor pode optar por uma adaptação evolutiva segura, onde somente protocolos seguros seriam utilizados. Ou ainda, o desenvolvedor escolher uma evolução que varie de acordo com o estado atual de carga do dispositivo.

É possível configurar os componentes do gMidArch através do mADL, mas para novos usuários isso pode ser desafiador. Sendo assim uma interface gráfica para configuração dos componentes do gMidArch pode facilitar o uso e a adoção do middleware por desenvolvedores.

A interface gráfica também pode ser utilizada para criação automática de *proxies* para os objetos remotos. Como parte da arquitetura RPC, objetos remotos têm a necessidade de

utilizarem um *proxy* para comunicação. No entanto, a geração dos *proxies* atualmente é feita de forma manual, o que levanta um novo ponto de melhoria. Uma interface gráfica que receba o objeto e gere automaticamente o *proxy* para o objeto remoto pode facilitar a adoção do gMidArch por novos usuários.

Outro ponto de melhoria é para que as aplicações clientes possam também enviar uma solicitação de adaptação ao servidor, permitindo que o cliente escolha o protocolo de transporte a ser utilizado. Isso pode ser útil em cenários onde o cliente tem conhecimento sobre condições diferentes da do servidor, e.g., a necessidade de controle do uso de energia em dispositivos IoT. E sendo assim o cliente pode sugerir um protocolo mais adequado para a comunicação baseado em suas necessidades.

Uma outra possibilidade de continuação do trabalho é a implementação de um sistema de aprendizado de máquina para prever o melhor protocolo a ser utilizado em cada situação.

REFERÊNCIAS

ABGAZ, Y.; MCCARREN, A.; ELGER, P.; SOLAN, D.; LAPUZ, N.; BIVOL, M.; JACKSON, G.; YILMAZ, M.; BUCKLEY, J.; CLARKE, P. Decomposition of monolith applications into microservices architectures: A systematic review. *IEEE Transactions on Software Engineering*, v. 49, n. 8, p. 4213–4242, Aug 2023. ISSN 1939-3520.

AL-JAROODI, J.; MOHAMED, N.; JAWHAR, I. A service-oriented middleware framework for manufacturing industry 4.0. *SIGBED Rev.*, Association for Computing Machinery, New York, NY, USA, v. 15, n. 5, p. 29–36, nov. 2018. Disponível em: <https://doi.org/10.1145/3292384.3292389>.

BELSHE, M.; PEON, R.; THOMSON, M. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC Editor, 2015. (RFC, 7540). Disponível em: <https://www.rfc-editor.org/info/rfc7540>.

BISHOP, M. *HTTP/3*. RFC Editor, 2022. RFC 9114. (Request for Comments, 9114). Disponível em: <https://www.rfc-editor.org/info/rfc9114>.

BRINKSCHULTE, M. Self-organizing middleware for cyber-physical networks. In: *Proceedings of the 20th International Middleware Conference Doctoral Symposium*. New York, NY, USA: Association for Computing Machinery, 2019. (Middleware '19), p. 14–16. ISBN 9781450370394. Disponível em: <https://doi.org/10.1145/3366624.3368158>.

CABRAL, B.; VENâNCIO, R.; COSTA, P.; FONSECA, T.; FERREIRA, L. L.; SEVERINO, R.; BARROS, A. Multiprotocol middleware translator for iot. In: *2024 27th Euromicro Conference on Digital System Design (DSD)*. [S.l.: s.n.], 2024. p. 327–334.

CAVALCANTI, D.; ROSA, N. Customizable and adaptable middleware of things. *International Journal of Communication Systems*, v. 37, n. 15, p. e5887, 2024. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/dac.5887>.

CHEN, W.; YE, K.; WANG, Y.; XU, G.; XU, C.-Z. How does the workload look like in production cloud? analysis and clustering of workloads on alibaba cluster trace. In: *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. [S.l.: s.n.], 2018. p. 102–109. ISSN 1521-9097.

DA, K.; DALMAU, M.; ROOSE, P. A Survey of adaptation systems. *International Journal on Internet and Distributed Computing Systems*, International journal on Internet and distributed computing systems, v. 2, n. 1, p. 1–18, nov. 2011. Disponível em: <https://hal.science/hal-00689773>.

ELHABBASH, A.; ELKHATIB, Y.; BOULOUKAKIS, G.; SALAMA, M. A middleware for automatic composition and mediation in iot systems. In: *Proceedings of the 12th International Conference on the Internet of Things*. New York, NY, USA: Association for Computing Machinery, 2023. (IoT '22), p. 127–134. ISBN 9781450396653. Disponível em: <https://doi.org/10.1145/3567445.3567451>.

FIELDING, R.; NOTTINGHAM, M.; RESCHKE, J. *HTTP Semantics*. [S.l.], 2022. Disponível em: <https://www.rfc-editor.org/rfc/rfc9110.txt>.

FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Tese (Doutorado) — University of California, Irvine, Irvine, CA, USA, sep 2000.

FIELDING, R. T.; NOTTINGHAM, M.; RESCHKE, J. *HTTP/1.1*. RFC Editor, 2022. RFC 9112. (Request for Comments, 9112). Disponível em: <https://www.rfc-editor.org/info/rfc9112>.

FOWLER, L. *Microservices: A definition of this new architectural term*. 2014. <https://martinfowler.com/articles/microservices.html>. Acesso em: 21 mai. 2025.

GO gRPC. *gRPC-Go package documentation*. 2025. <https://github.com/grpc/grpc-go>. Acesso em: 21 mar. 2025.

GO-RABBITMQ. *Go RabbitMQ package documentation*. 2025. <https://www.rabbitmq.com>. Acesso em: 21 mar. 2025.

GODFREY, R.; INGHAM, D.; SCHLOMING, R. *OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0*. [S.l.], 2012. Disponível em: <http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-complete-v1.0-os.pdf>.

GORILLA. *Gorilla web toolkit documentation*. 2025. <https://gorilla.github.io/>. Acesso em: 21 mar. 2025.

Grace Hopper. Grace hopper, cdr., u.s.n. *Computerworld*, IDG Enterprise, v. 10, n. 4, p. 9, Jan 1976.

GRPC. *gRPC documentation*. 2025. <https://grpc.io/docs/>. Acesso em: 20 jun. 2025.

IBM. *An Architectural Blueprint for Autonomic Computing*. [S.l.], 2005.

ISO/IEC. *Information technology – Open Systems Interconnection – Basic Reference Model: The basic model*. 1994.

IYENGAR, J.; THOMSON, M. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC Editor, 2021. RFC 9000. (Request for Comments, 9000). Disponível em: <https://rfc-editor.org/rfc/rfc9000.txt>.

KEPHART, J.; CHESS, D. The vision of autonomic computing. *Computer*, v. 36, p. 41 – 50, 02 2003.

KHEZEMI, N.; MINANI, J.; SABIR, F.; MOHA, N.; GUéHéNEUC, Y.-G.; EL-BOUSSAIDI, G. A systematic literature review of iot system architectural styles and their quality requirements. *IEEE Internet of Things Journal*, v. 11, p. 37599–37616, 12 2024.

Lalanda, P.; Morand, D.; Chollet, S. Autonomic mediation middleware for smart manufacturing. *IEEE Internet Computing*, v. 21, n. 1, p. 32–39, 2017.

LANGLEY, A.; RIDDOCH, A.; WILK, A.; VICENTE, A.; KRASIC, C.; ZHANG, D.; YANG, F.; KOURANOV, F.; SWETT, I.; IYENGAR, J.; BAILEY, J.; DORFMAN, J.; ROSKIND, J.; KULIK, J.; WESTIN, P.; TENNETI, R.; SHADE, R.; HAMILTON, R.; VASILIEV, V.; CHANG, W.-T.; SHI, Z. The quic transport protocol: Design and internet-scale deployment. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. New York, NY, USA: Association for Computing Machinery, 2017. (SIGCOMM '17), p. 183–196. ISBN 9781450346535. Disponível em: <https://doi.org/10.1145/3098822.3098842>.

MEDVIDOVIC, N.; TAYLOR, R. N. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 26, n. 1, p. 70–93, jan. 2000. ISSN 0098-5589.

MITRE. *Vulnerability Details: CVE-2013-3587 - Man-in-the-middle attack on SSL 1.2*. 2013. <https://www.cve.org/CVERecord?id=CVE-2013-3587>. Acessado em 19 de julho de 2025.

NAZIRIDIS, N. *Uma introdução ao HTTP/2*. 2018. <https://www.ssl.com/pt/article/an-introduction-to-http2/>. Acessado em 19 de julho de 2025.

PEREIRA, F.; VALENTE, M.; BIGONHA, R.; BIGONHA, M. Arcademis: A framework for object-oriented communication middleware development. *Software: Practice and Experience*, v. 36, p. 495 – 512, 04 2006.

POSTEL, J. *User Datagram Protocol*. RFC Editor, 1980. RFC 768. (Request for Comments, 768). Disponível em: <https://www.rfc-editor.org/info/rfc768>.

PRADEEP, P.; KRISHNAMOORTHY, S.; VASILAKOS, A. V. A holistic approach to a context-aware iot ecosystem with adaptive ubiquitous middleware. *Pervasive and Mobile Computing*, v. 72, p. 101342, 2021. ISSN 1574-1192. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1574119221000134>.

quic-go Contributors. *quic-go: A QUIC implementation in pure Go*. 2024. <https://github.com/quic-go/quic-go/releases/tag/v0.42.0>. Acessado em 17 de julho de 2025.

RabbitMQ. *RabbitMQ 4.1 Documentation - Compatibility and Conformance*. 2025. <https://www.rabbitmq.com/docs/specification>. Acesso em: 24 jun. 2025.

RESCORLA, E. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC Editor, 2018. RFC 8446. (Request for Comments, 8446). Disponível em: <https://rfc-editor.org/rfc/rfc8446.txt>.

ROSA, N.; CAVALCANTI, D.; CAMPOS, G.; SILVA, A. Adaptive middleware in go - a software architecture-based approach. *Journal of Internet Services and Applications*, 2020.

ROSA, N. S.; CAMPOS, G. M.; CAVALCANTI, D. J. Lightweight formalisation of adaptive middleware. *Journal of Systems Architecture*, v. 97, p. 54–64, 2019. ISSN 1383-7621. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1383762118300936>.

RPC-GO. *Go RPC package documentation*. 2025. <https://pkg.go.dev/net/rpc>. Acesso em: 21 mar. 2025.

SALEHIE, M.; TAHVILDARI, L. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 4, n. 2, maio 2009. ISSN 1556-4665. Disponível em: <https://doi.org/10.1145/1516533.1516538>.

SANGEETA, K.; S, A. D. V.; JAIN, A.; YADAV, D. K.; TYAGI, L. K.; MOHSEN, Z. S. Adaptive middleware solutions for seamless integration of legacy and modern ict systems in smart grids. In: *2023 International Conference on Power Energy, Environment and Intelligent Control (PEEIC)*. [S.l.: s.n.], 2023. p. 443–448.

SHI, W.; CAO, J.; ZHANG, Q.; LI, Y.; XU, L. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, v. 3, n. 5, p. 637–646, Oct 2016. ISSN 2327-4662.

Singhai, A.; Sane, A.; Campbell, R. H. Quarterware for middleware. In: *Proceedings. 18th International Conference on Distributed Computing Systems (Cat. No.98CB36183)*. [S.l.: s.n.], 1998. p. 192–201.

SöYLEMEZ, M.; TEKINERDOGAN, B.; KOLUKıSA, A. Challenges and solution directions of microservice architectures: A systematic literature review. *Applied Sciences*, v. 12, p. 5507, 05 2022.

THOMSON, M.; BENFIELD, C. *RFC 9113: HTTP/2*. USA: RFC Editor, 2022.

THURLOW, R. *RPC: Remote Procedure Call Protocol Specification Version 2*. RFC Editor, 2009. RFC 5531. (Request for Comments, 5531). Disponível em: <https://www.rfc-editor.org/info/rfc5531>.

VERGNAUD, T.; HUGUES, J.; PAUTET, L.; KORDON, F. Polyorb: A schizophrenic middleware to build versatile reliable distributed applications. In: LLAMOSÍ, A.; STROHMEIER, A. (Ed.). *Reliable Software Technologies - Ada-Europe 2004*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 106–119. ISBN 978-3-540-24841-5.

VOLTER, M.; KIRCHER, M.; ZDUN, U. *Remoting Patterns: Foundations of Enterprise, Internet and Real Time Distributed Object Middleware*. [S.l.]: John Wiley & Sons Ltd, 2005.

WEYNS, D. *An Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective*. [S.l.: s.n.], 2021. ISBN 9781119574910.

Xie, W.; Li, Z.; Zhao, Y. Coservices: A web service based middleware framework for interactive cooperative applications. In: *2013 Third International Conference on Intelligent System Design and Engineering Applications*. [S.l.: s.n.], 2013. p. 507–513.

APÊNDICE A – MODELO DOCKER COMPOSE PARA EXPERIMENTOS

Foi criado um arquivo *docker-compose* para permitir a execução dos experimentos realizados neste trabalho de forma autônoma, garantindo uma maior confiabilidade nos resultados. Para isso o *docker-compose* foi criado como um *model* do **pAdapt**, ou seja, um arquivo modelo que permite a substituição de variáveis para a criação de diferentes cenários de teste. O Código Fonte 15 apresenta o modelo utilizado nos experimentos, onde as variáveis são definidas no início de cada experimento executado, e substituídas no modelo do *docker-compose* para a criação do ambiente de teste.

Código Fonte 15 – Modelo de configuração do Docker Compose utilizado nos experimentos

```
version: "3.3"
services:
  server:
    image: <image.server>
    deploy:
      replicas: 1
      restart_policy:
        condition: none
      resources:
        limits:
          cpus: "0.4"
          memory: 256M
        reservations:
          cpus: "0.4"
          memory: 256M
    environment:
      CA_PATH: "/usr/src/app/examples/certs/myCA.pem"
      CRT_PATH: "/usr/src/app/examples/certs/server.pem"
      KEY_PATH: "/usr/src/app/examples/certs/server.key"
      TIME_TO_START_SERVER: 1
      INJECTION_INTERVAL: <adaptation.interval>

  client:
    image: <image.client>
    deploy:
      replicas: 1
      restart_policy:
        condition: none
      resources:
        limits:
          cpus: "0.4"
          memory: 256M
```

```
        reservations:
          cpus: "0.4"
          memory: 256M
    environment:
      CA_PATH: "/usr/src/app/examples/certs/myCA.pem"
      <specific.env.client>
      SAMPLE_SIZE: <sample.size>
      AVERAGE_WAITING_TIME: <average.waiting.time>
      TIME_TO_START_CLIENT: 5
```

Fonte: Elaborado pelo autor (2025)