PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS

ESCOLA POLITÉCNICA E DE ARTES

GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

DESENVOLVIMENTO DE UM CHAT SEGURO COM AUTENTICAÇÃO DE USUÁRIOS E CRIPTOGRAFIA DE DADOS USANDO RUST

BRUNO LINO DO NASCIMENTO SANTANA

GOIÂNIA

2025

BRUNO LINO DO NASCIMENTO SANTANA

DESENVOLVIMENTO DE UM CHAT SEGURO COM AUTENTICAÇÃO DE USUÁRIOS E CRIPTOGRAFIA DE DADOS USANDO RUST

Trabalho de Conclusão de Curso apresentado à Escola de Ciências Exatas e da Computação, da Pontifícia Universidade Católica de Goiás, como parte dos requisitos para a obtenção do título de Bacharel em Engenharia de Computação.

Orientador:

Prof. Dr. José Luiz de Freitas Júnior

Banca examinadora:

Profa. Ma. Ludmilla Reis Pinheiro dos Santos

Prof. Me. Olegário Correa da Silva Neto

GOIÂNIA

2025

AGRADECIMENTOS

Agradeço primeiramente a Deus, por me dar forças e saúde para enfrentar os desafios desta jornada.

Aos meus pais e familiares, pelo apoio incondicional e incentivo em todos os momentos. Aos colegas e amigos que, de alguma forma, contribuíram com palavras de apoio, trocas de conhecimento ou simplesmente por estarem presentes nos momentos difíceis.

Por fim, ao professor José Luiz de Freitas Júnior, pela dedicação, paciência e orientação ao longo de todo o desenvolvimento deste trabalho.

“O homem não teria alcançado o possível se, repetidas vezes, não tivesse tentado o impossível.”

Max Weber

RESUMO

Este Trabalho de Conclusão de Curso apresenta o desenvolvimento de um sistema de chat seguro, utilizando autenticação de usuários e criptografia de dados, implementado na linguagem de programação *Rust*. O objetivo principal foi estudar e aplicar técnicas de segurança da informação, abordando conceitos como funções *hash*, criptografia simétrica e assimétrica, armazenamento seguro e comunicação protegida.

Durante o desenvolvimento, foram utilizados algoritmos de criptografia como AES (*Advanced Encryption Standard*) para garantir a confidencialidade das mensagens, e *bcrypt* para assegurar a proteção das senhas dos usuários através de *hash* seguro. Além disso, empregou-se o algoritmo RSA (*Rivest–Shamir–Adleman*) na troca segura da chave AES, garantindo que mesmo em redes inseguras, a chave simétrica fosse compartilhada de maneira protegida contra interceptações.

O servidor foi projetado para gerenciar múltiplas conexões simultâneas, assegurando que toda comunicação seja protegida contra acessos não autorizados. O trabalho demonstra que, com o uso das ferramentas corretas, é possível construir sistemas seguros, destacando o *Rust* como uma linguagem moderna e eficiente, especialmente indicada para aplicações onde a segurança e o desempenho são fundamentais.

Palavras-chave: *Rust*, Criptografia, Autenticação, Funções *Hash*, *Chat* Seguro, AES, RSA, *Bcrypt*.

ABSTRACT

This Final Project presents the development of a secure chat system, using user authentication and data encryption, implemented in the Rust programming language. The main objective was to study and apply information security techniques, addressing concepts such as hash functions, symmetric and asymmetric cryptography, secure storage and secure communication.

During development, encryption algorithms such as AES (Advanced Encryption Standard) were used to guarantee the confidentiality of messages, and bcrypt to ensure the protection of user passwords through secure hashing. In addition, the RSA (Rivest–Shamir–Adleman) algorithm was used to securely exchange the AES key, ensuring that even in insecure networks, the symmetric key was shared in a manner protected against interception.

The server was designed to manage multiple simultaneous connections, ensuring that all communication is protected against unauthorized access. The work demonstrates that, with the use of the right tools, it is possible to build secure systems, highlighting Rust as a modern and efficient language, especially suitable for applications where security and performance are essential.

Keywords: Rust, Cryptography, Authentication, Hash Functions, Secure Chat, AES, RSA, Bcrypt.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS E ABREVIAÇÕES

AES – *Advanced Encryption Standard*

RSA – *Rivest-Shamir-Adleman*

IDE – *Integrated Development Environment*

NIST – *National Institute of Standards and Technology*

XOR – *Exclusive Or*

SHA – *Secure Hash Algorithm*

TLS – *Transport Layer Security*

IV – *Initialization Vector*

CBC – *Cipher Block Chaini*

TCP – *Transmission Control Protocol*

IP – *Internet Protocol*

MITM – *Man-in-the-middle*

EksBlowfish – *Expensive Key Schedule Blowfish*

DES – *Data Encryption Standard*

ACID – Atomicidade, Consistência, Isolamento e Durabilidade

SUMÁRIO

CAPÍTULO I - INTRODUÇÃO

A segurança na comunicação digital tem se tornado um aspecto fundamental em um cenário cada vez mais conectado, onde a troca de informações sensíveis e confidenciais acontece de forma constante por meio de plataformas *online*. O crescimento exponencial da digitalização, aliado ao aumento de ataques cibernéticos, vazamentos de dados, fraudes e interceptações de informações, evidencia a necessidade urgente de implementar mecanismos de proteção. Garantir a confidencialidade, integridade e autenticidade das informações não é mais uma opção, mas sim uma exigência essencial para empresas, organizações e usuários em geral.

Dentro desse contexto, a criptografia surge como uma das principais ferramentas para assegurar a proteção de dados contra acessos não autorizados. Especificamente, o algoritmo AES (*Advanced Encryption Standard*) se destaca pela sua eficiência e segurança na proteção de dados em trânsito, sendo amplamente utilizado em aplicações críticas que exigem alto desempenho e resistência a ataques. Complementando essa abordagem, o algoritmo RSA (*Rivest–Shamir–Adleman*) é adotado para garantir a segurança na troca de chaves simétricas, permitindo que usuários compartilhem segredos de forma segura mesmo em ambientes potencialmente comprometidos. A combinação de criptografia simétrica e assimétrica aumenta a segurança nos dados em sistemas de comunicação.

Aliado a isso, a linguagem de programação *Rust* desempenha um papel estratégico. *Rust* é reconhecida por oferecer segurança de memória, concorrência segura e alta performance, características que a tornam extremamente adequada para o desenvolvimento de sistemas críticos, como servidores de comunicação segura. Diferente de outras linguagens tradicionais, *Rust* previne classes inteiras de vulnerabilidades, como estouros de *buffer* e condições de corrida (um tipo de falha que ocorre quando duas ou mais partes de um programa acessam e manipulam um recurso compartilhado ao mesmo tempo, resultando em comportamentos imprevisíveis ou incorretos), o que contribui significativamente para aumentar o nível de segurança da aplicação.

1.1 OBJETIVO GERAL

Desenvolver uma aplicação em *Rust* com suporte à autenticação segura de usuários e troca de mensagens criptografadas, utilizando algoritmos de *hash* e criptografia, com armazenamento persistente em banco de dados.

1.2 OBJETIVOS ESPECÍFICOS

Este trabalho tem como objetivos:

- Desenvolver uma aplicação segura utilizando a linguagem *Rust*, explorando suas capacidades para controle de concorrência, segurança de memória e robustez na manipulação de erros;
- Implementar uma rotina de cadastro e login de usuários com senhas protegidas por *hashing*, utilizando o algoritmo *bcrypt*;
- Utilizar criptografia simétrica e assimétrica para proteger as mensagens enviadas entre os usuários;
- Armazenar mensagens criptografadas de forma segura em um banco de dados PostgreSQL, garantindo persistência e integridade dos dados;
- Demonstrar o uso prático da linguagem *Rust* em aplicações reais que demandam alto grau de segurança, performance e confiabilidade.

1.3 JUSTIFICATIVA

A escolha do tema justifica-se pela necessidade de construir soluções seguras de comunicação, explorando tecnologias modernas e eficazes. A linguagem *Rust* surge como uma forte candidata para esse propósito por oferecer um modelo de segurança de memória sem coletor de lixo, evitando falhas comuns como ponteiros nulos, condições de corrida e estouros de buffer. Esses recursos tornam *Rust* ideal para aplicações onde desempenho e segurança são críticos, como é o caso de sistemas de mensagens.

Neste trabalho, desenvolveu um sistema de *chat* seguro, que emprega autenticação de usuários utilizando funções *hash (Bcrypt),* e criptografia para proteger o conteúdo das mensagens trocadas. Além disso, o uso de um banco de dados para persistência e histórico das mensagens garante integridade e confiabilidade mesmo em casos de desconexão dos usuários.

Dessa forma, o projeto não apenas demonstra a viabilidade técnica da linguagem *Rust* para o desenvolvimento de sistemas seguros, mas também contribui com uma solução prática e aplicável para comunicação protegida, validando conceitos de segurança da informação, criptografia e desenvolvimento seguro.

1.4 METODOLOGIA

No decorrer do trabalho, utilizou-se como materiais computadores que utilizem o sistema operacional Windows 11. Para a implementação dos algoritmos, foi usada a linguagem de programação *Rust*, em sua versão mais atualizada (1.85.0), o ambiente de desenvolvimento integrado (IDE) utilizado foi o *vscode*, juntamente com o *pgAdmin4*. Os estudos foram realizados através de artigos, livros, slides e sites.

1.5 ORGANIZAÇÃO DO TRABALHO

Este trabalho está dividido em cinco capítulos. Neste primeiro capítulo foram apresentadas as motivações para o desenvolvimento do sistema, descrevendo os objetivos gerais e específicos.

No segundo capítulo explora os conceitos fundamentais da criptografia e sua aplicação em um *chat* seguro desenvolvido na linguagem *Rust*.

No terceiro capítulo explora o conceito de função *hash*, explica a sua importância na parte de autenticação de usuários e como foi feita sua implementação no projeto.

No quarto capítulo são descritas as ferramentas, linguagem e bibliotecas adotadas, utilizadas no desenvolvimento do projeto e são apresentados os principais resultados obtidos com o desenvolvimento do projeto.

No quinto capítulo são apresentadas a considerações finais, dificuldades encontradas e sugestões para trabalhos futuros.

CAPÍTULO II - CRIPTOGRAFIA E SEGURANÇA EM COMUNICAÇÃO DIGITAL

Este capítulo explora os conceitos fundamentais da criptografia, tipos e os algoritmos AES e RSA.

2.1 INTRODUÇÃO

A criptografia é um dos pilares da segurança da informação, sendo amplamente utilizada para garantir a confidencialidade, integridade e autenticidade das comunicações digitais. Em um ambiente onde a troca de informações ocorre em tempo real, como em sistemas de mensagens instantâneas, a proteção contra acessos não autorizados se torna essencial.

Com o avanço da tecnologia e o aumento das ameaças cibernéticas, é fundamental o uso de técnicas de criptografia robustas para proteger os dados dos usuários e evitar interceptações indevidas. De acordo com Schneier (2015), a criptografia é uma das ferramentas mais poderosas para proteger a integridade, a confidencialidade e a autenticidade dos dados no ambiente digital.

2.2 CONCEITOS BÁSICOS DE CRIPTOGRAFIA

A criptografia consiste na transformação de dados legíveis (texto claro) em um formato ilegível (texto cifrado), garantindo que apenas os destinatários autorizados possam acessar a informação original. Esse processo é essencial para proteger conversas privadas em aplicações de mensagens, impedindo que terceiros tenham acesso ao conteúdo transmitido.

De acordo com Stallings (2015), a criptografia é um dos principais mecanismos de segurança para prevenir a exposição não autorizada de dados, impedindo que informações sigilosas sejam interceptadas ou modificadas.

No *chat* seguro desenvolvido em *Rust*, a criptografia permite que as mensagens enviadas entre os usuários sejam protegidas durante toda a transmissão, mesmo em redes potencialmente inseguras.

2.3 PRINCÍPIOS BÁSICOS DA CRIPTOGRAFIA

Os princípios da criptografia são baseados em quatro elementos fundamentais:

- Confidencialidade: O objetivo principal da criptografia é proteger o conteúdo das mensagens, transformando-as em um formato ilegível para quem não tem a chave correta;

- Integridade: Garantir que a mensagem não seja modificada durante a transmissão. Funções *hash* são amplamente usadas para verificar a integridade dos dados. Se uma transação é alterada, o *hash* correspondente também será modificado, alertando para uma possível fraude;
- Autenticidade: Envolve a verificação da identidade das partes envolvidas em uma transação;
- Não-repúdio: Refere-se à capacidade de garantir que uma parte envolvida não possa negar sua participação em uma transação.

2.3 TIPOS DE CRIPTOGRAFIA

A classificação dos algoritmos de criptografia é baseada nos dois tipos principais de chaves utilizadas: simétrica e assimétrica.

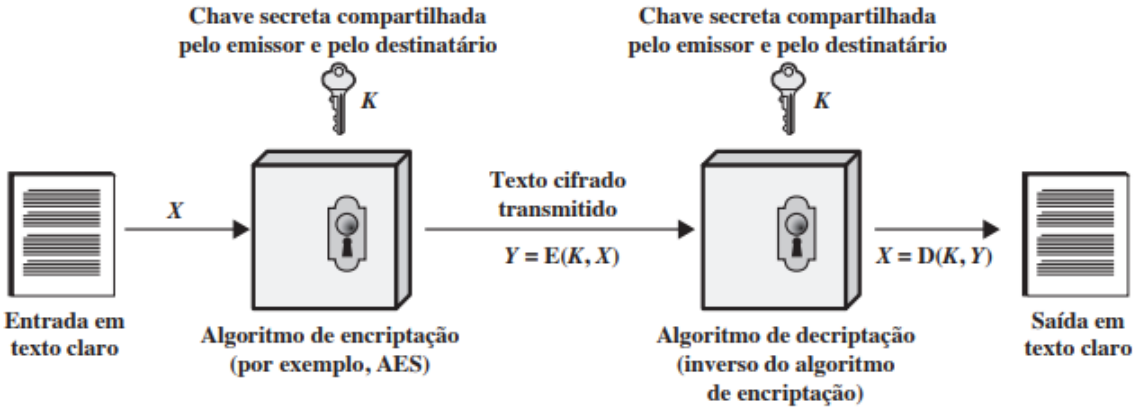
A criptografia simétrica utiliza uma única chave para criptografar e descriptografar dados. Essa chave deve ser mantida em segredo entre as partes que compartilham a informação, sendo ideal para aplicações de grande volume de dados devido à sua velocidade.

- Vantagens: Alta velocidade de processamento, especialmente com algoritmos como o AES.
- Desvantagens: A necessidade de compartilhar a mesma chave entre as partes pode comprometer a segurança em grandes redes.

A criptografia simétrica é eficiente em termos de processamento, mas sua segurança depende do compartilhamento seguro da chave entre as partes (SCHNEIER, 2015).

A Figura 1 mostra o processo de criptografia simétrica, onde tanto o emissor quanto o destinatário compartilham uma mesma chave secreta *K*. Primeiro, o texto claro *X* é transformado em um texto cifrado *Y* pelo algoritmo de encriptação, como o AES. Em seguida, o texto cifrado *Y* é transmitido ao destinatário, que o decifra utilizando o mesmo algoritmo de decriptação e a chave secreta *K*, obtendo o texto claro original *X* de volta. Esse método é eficiente, mas depende da segurança e do sigilo da chave compartilhada.

Figura 1 - Modelo simplificado da encriptação simétrica.



Fonte: STALLINGS, 2015.

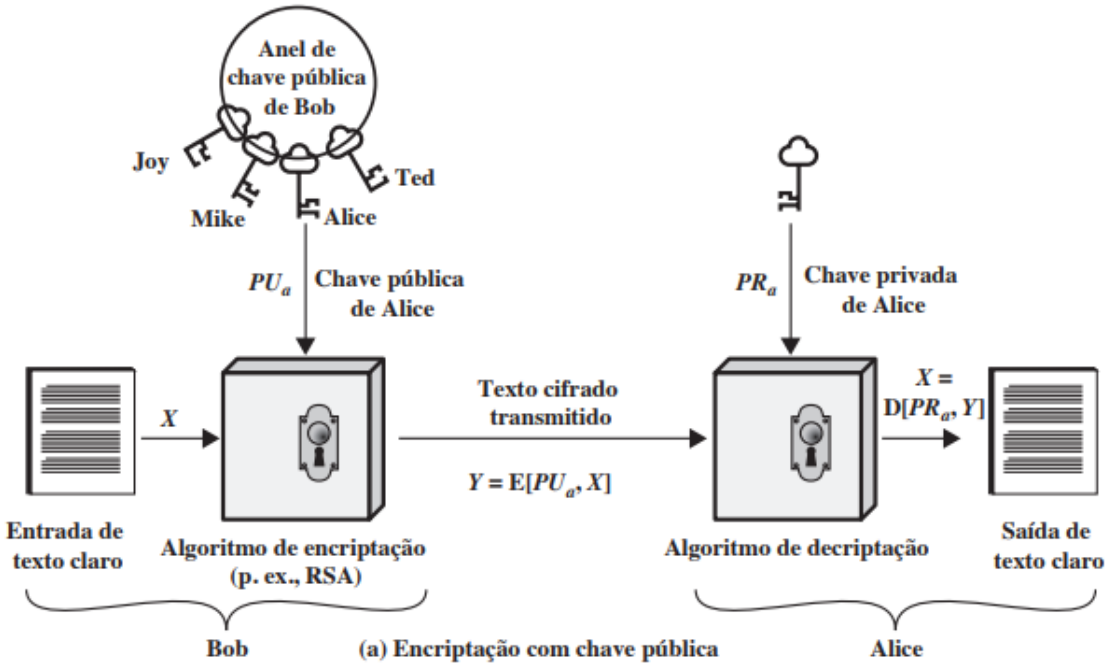
A criptografia assimétrica utiliza um par de chaves: uma pública e uma privada. A chave pública pode ser compartilhada abertamente para criptografar mensagens, enquanto a chave privada, que deve ser mantida em segredo, é usada para descriptografá-las. Esse modelo é a base de algoritmos como o RSA.

- Vantagens: Elimina a necessidade de compartilhar uma única chave, facilitando a segurança em redes abertas.
- Desvantagens: É geralmente mais lenta do que a criptografia simétrica, devido à complexidade dos cálculos matemáticos envolvidos.

A criptografia assimétrica permite uma comunicação segura sem a necessidade de pré-compartilhamento de chaves, o que a torna essencial para ambientes distribuídos e inseguros (STALLINGS, 2014).

A Figura 2 mostra o processo de criptografia assimétrica, onde são utilizadas chaves públicas e privadas. Nesse exemplo, Bob deseja enviar uma mensagem para Alice de forma segura. Ele usa a chave pública de Alice, que está disponível para qualquer pessoa no anel de chaves públicas, para encriptar o texto claro *X*, gerando o texto cifrado *Y*. Esse texto cifrado é então transmitido para Alice. Ao recebê-lo, Alice utiliza sua chave privada, que é exclusivamente dela, para decriptar *Y* e obter o texto claro original *X*. Esse método permite a comunicação segura sem a necessidade de compartilhamento direto de uma chave secreta entre as partes.

Figura 2 - Modelo simplificado da encriptação assimétrica.



Fonte: STALLINGS, 2015.

2.4 ALGORITMO AES

O AES é um dos algoritmos de criptografia mais utilizados para a proteção de dados em várias indústrias. Desenvolvido para substituir o antigo algoritmo DES (*Data Encryption Standard*), o AES se destaca por sua eficiência e segurança. Desde que foi adotado como padrão pelo *National Institute of Standards and Technology* (NIST) em 2001, ele se tornou a escolha predominante para a criptografia de dados sensíveis na web.

O AES foi projetado para ser eficiente tanto em software quanto em hardware, oferecendo alta velocidade e segurança para uma ampla gama de aplicações (SCHNEIER, 2015).

O AES é um algoritmo de criptografia simétrica por blocos, o que significa que utiliza a mesma chave tanto para a criptografia quanto para a descriptografia dos dados, segundo NIST (2023), um bloco é uma sequência de bits de um dado comprimento fixo. Uma cifra de bloco é uma família de permutações de blocos que é parametrizada por uma sequência de bits chamada de chave.

Ele funciona dividindo os dados em blocos de bits e aplicando uma série de operações matemáticas, chamadas de "rodadas", que variam conforme o comprimento da chave utilizada:

- AES-128: 10 rodadas
- AES-192: 12 rodadas
- AES-256: 14 rodadas

Cada rodada consiste em quatro etapas fundamentais:

- *SubBytes* (Substituição de *Bytes*): Nessa etapa, cada *byte* presentado na matriz *State* é substituído por outro valor, utilizando uma tabela de substituição conhecida como *S-Box* mostrada na Figura 3. Essa tabela foi cuidadosamente projetada para resistir a ataques como criptoanálise linear e diferencial. O processo de substituição, ilustrado nas Figuras 4 e 5, consiste em uma operação não linear fundamental no algoritmo AES. Cada *byte* do bloco de dados é interpretado como um valor hexadecimal que representa um índice na tabela de substituição conhecida como *S-Box*, o valor correspondente encontrado nessa posição na *S-Box* substitui diretamente o *byte* original.

Figura 3 - Tabela S-Box.

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

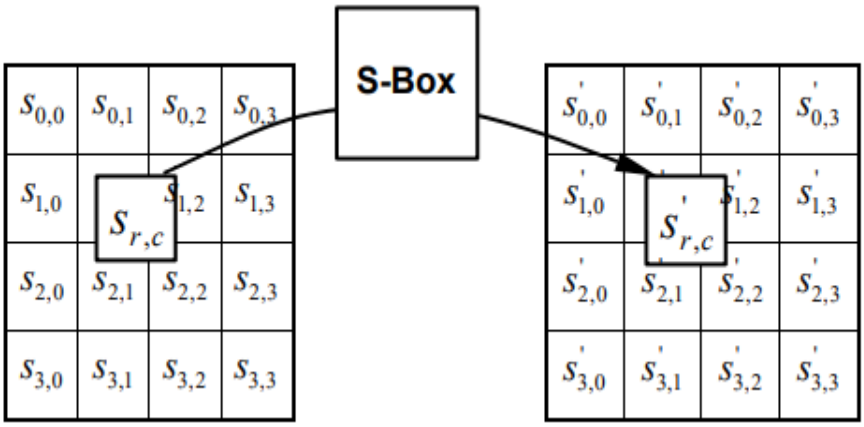
Fonte: NIST, 2023.

Figura 4 - Transformação SubBytes.

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

Fonte: NIST, 2023.

Figura 5 - Aplica o S-box a cada byte do estado.

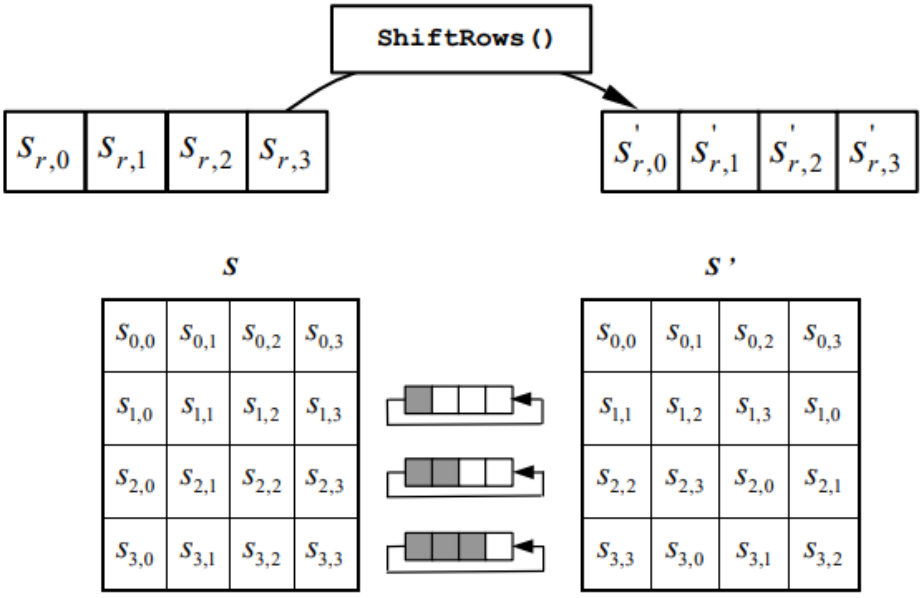


Fonte: NIST, 2023.

- *ShiftRows* (Deslocamento de Linhas): Essa etapa consiste no deslocamento circular dos *bytes* nas linhas da matriz *State*. A primeira linha permanece inalterada, enquanto a segunda linha é deslocada uma posição para a esquerda, a terceira linha sofre um deslocamento de duas posições, e a quarta linha, três posições como mostrado na Figura 6. Esse deslocamento faz com que *bytes* que antes estavam agrupados na mesma coluna se distribuam entre diferentes colunas, aumentando significativamente o

entrelaçamento dos dados e tornando mais complexo qualquer tipo de análise estatística por parte de um atacante.

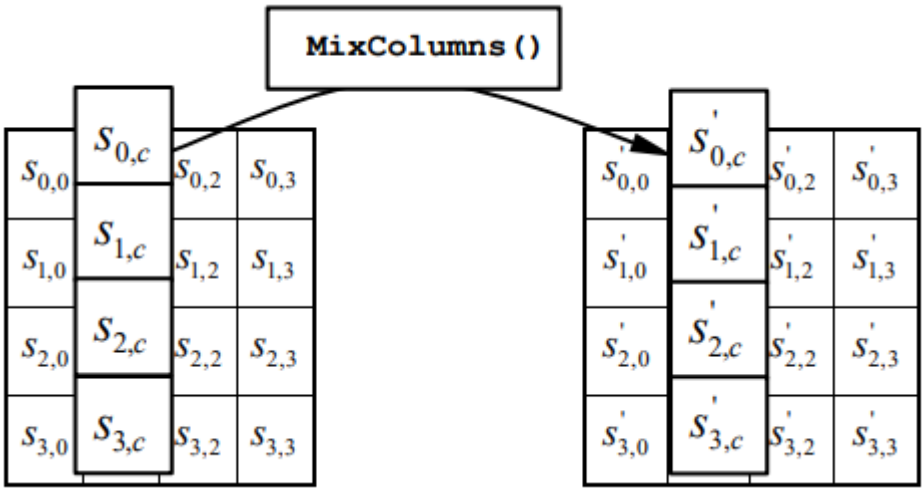
Figura 6 - ShiftRows.



Fonte: NIST,2023.

- *MixColumns* (Mistura de Colunas): Nessa operação mostrada na Figura 7, cada coluna da matriz *State* é tratada como um vetor de quatro *bytes* e multiplicada por uma matriz fixa, utilizando operações no campo finito $GF(2^8)$.Isso significa que tanto as somas quanto as multiplicações seguem regras matemáticas específicas desse campo, baseadas em operações de XOR e multiplicações modulares. O efeito dessa transformação é que cada byte de uma coluna passa a ser uma combinação dos quatro *bytes* originais daquela coluna, fazendo com que qualquer modificação em um único *byte* se propague, afetando toda a coluna. É importante destacar que essa etapa é aplicada em todas as rodadas do algoritmo, exceto na última, na qual ela é propositalmente omitida para simplificar o processo de decifragem.

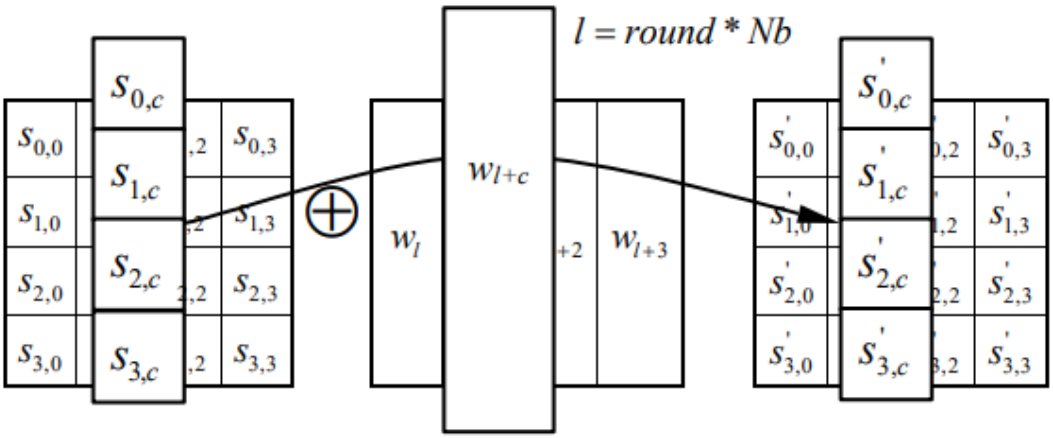
Figura 7 - Operação MixColumns.



Fonte: NIST, 2023.

- *AddRoundKey* (Adição de Chave): Nessa etapa, Figura 8, cada *byte* da matriz *State* é combinado, por meio da operação XOR, com um *byte* correspondente da chave de rodada, que é previamente derivada da chave principal através de um processo conhecido como expansão de chave. A utilização do XOR se justifica pela sua simplicidade e eficiência, além de ser uma operação reversível e essencial para que o processo de decifragem funcione corretamente.

Figura 8 - AddRoundKey.



Fonte: NIST, 2023.

As propriedades de segurança do AES são resultado de uma estrutura de substituição-permutação, que combina operações em várias rodadas para resistir a ataques conhecidos (NIST, 2023).

Essas etapas tornam o AES um algoritmo altamente resistente a ataques, uma vez que a combinação dessas operações impede que qualquer transformação inversa possa recuperar o dado original sem a chave correta.

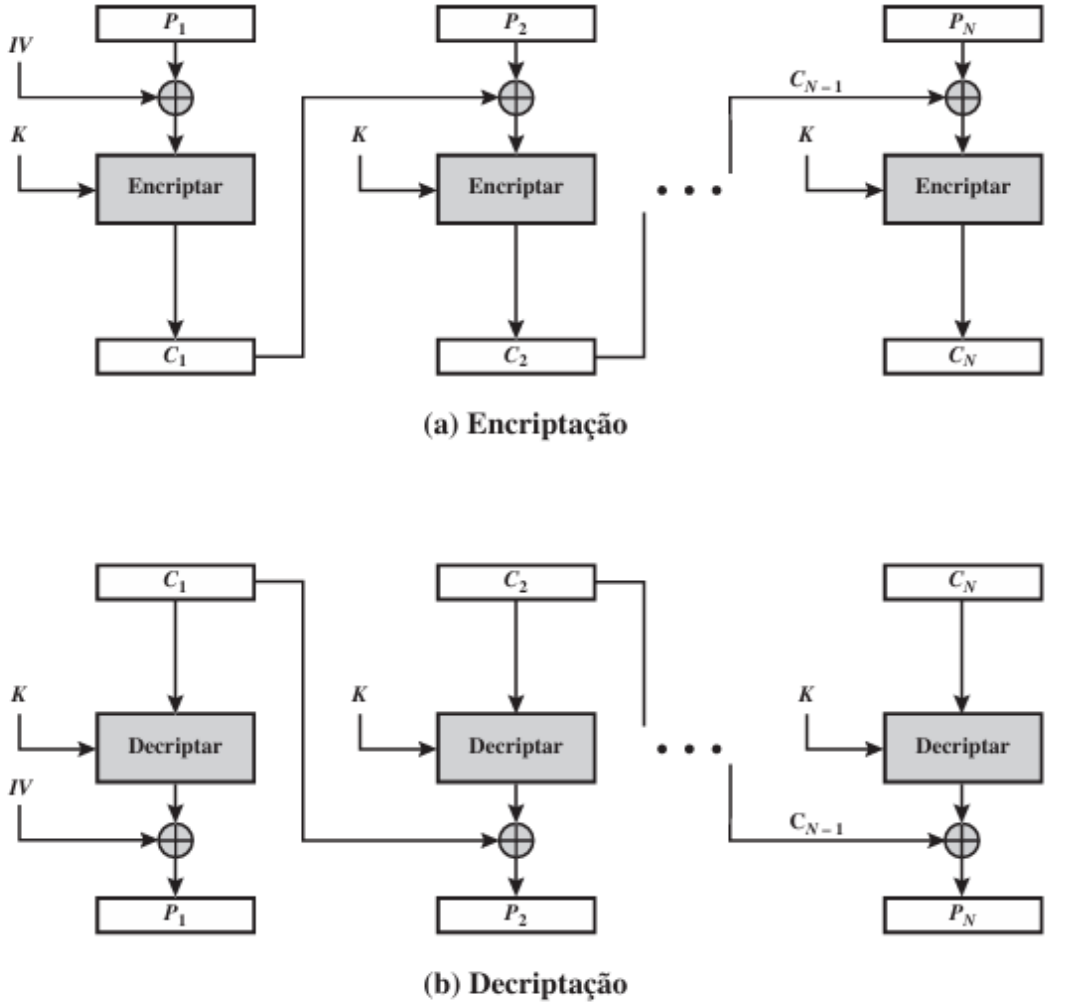
2.4.1 MODO DE OPERAÇÃO CBC COM PREENCHIMENTO PKCS7

Na criptografia simétrica, os algoritmos de cifra de bloco, operam sobre dados de tamanho fixo, geralmente blocos de 128 bits. No entanto, mensagens reais raramente possuem um tamanho exato múltiplo desse tamanho de bloco. Além disso, para evitar padrões e tornar a criptografia mais segura, são utilizados modos de operação que definem como os blocos de dados são processados em sequência. Um dos modos de operação mais conhecidos e amplamente utilizados é o CBC (*Cipher Block Chaining*).

O modo CBC introduz uma dependência entre os blocos de texto claro, de forma que a cifragem de um bloco depende não apenas do próprio conteúdo, mas também do resultado da cifragem do bloco anterior. Nesse esquema, a entrada do algoritmo de encriptação é o XOR do bloco de texto claro atual e do bloco de texto cifrado anterior; a mesma chave é usada para cada bloco. Com efeito, encadeamos o processamento da sequência de blocos de texto claro.

A entrada da função de encriptação para cada bloco de texto claro não possui qualquer relacionamento fixo com o de texto claro. Portanto, padrões repetitivos de b *bits* não são expostos. Para a decriptação, cada bloco de cifra é passado pelo algoritmo de decriptação. O resultado segue por um XOR com o bloco de texto cifrado anterior, a fim de produzir o bloco de texto claro (STALLINGS, 2015), esse processo é mostrado na Figura 9. No início do processo, como não há um bloco anterior ao primeiro bloco da mensagem, utiliza-se um valor inicial chamado de IV (vetor de inicialização). Esse IV deve ser aleatório e único para cada operação de cifragem, embora não precise ser secreto, desde que seja transmitido ou armazenado de forma íntegra junto com o texto cifrado.

Figura 9 - Modo cipher block chaining (CBC).



Fonte: STALLINGS, 2015.

Porém, como mencionado, os algoritmos de cifra de bloco requerem que o tamanho da mensagem seja exatamente um múltiplo do tamanho do bloco. Para resolver essa limitação, aplica-se um esquema de preenchimento, sendo o PKCS7 um dos mais utilizados, que padroniza a sintaxe de mensagens criptográficas. Esse método consiste em adicionar *bytes* no final da mensagem, de forma que o comprimento total atinja o múltiplo necessário. O valor de cada *byte* de preenchimento corresponde à quantidade total de *bytes* adicionados. Por exemplo, se faltarem quatro *bytes* para completar o bloco, serão adicionados quatro *bytes* com o valor 0x04. Caso a mensagem já possua um tamanho exato, é adicionado um bloco inteiro com bytes de valor igual ao tamanho do bloco (no caso do AES, 0x10, que representa 16 em decimal).

Esse esquema de preenchimento é amplamente adotado por garantir que, no momento da decifragem, seja possível identificar com clareza onde termina a mensagem original e onde começa o preenchimento, bastando analisar o valor do último *byte* do último bloco decifrado. Assim, o modo CBC em conjunto com o preenchimento PKCS7 proporciona uma solução robusta e segura para a cifragem de dados de tamanhos arbitrários, sendo amplamente empregado em padrões de segurança, protocolos de comunicação e armazenamento seguro de informações sensíveis.

2.5 ALGORITMO RSA

O RSA é um dos algoritmos mais utilizados para criptografia de chave pública. Desenvolvido em 1977, ele revolucionou a área da criptografia por permitir o uso de um par de chaves, uma pública e uma privada para realizar operações de criptografia e descriptografia.

O RSA é uma cifra de bloco na qual o texto às claras e o texto cifrado são inteiros entre 0 e n - 1 para algum n (STALLINGS, 2014).

O RSA é baseado em conceitos de teoria dos números e operações matemáticas complexas, como a fatoração de grandes números primos. O processo de funcionamento do RSA pode ser dividido nas seguintes etapas:

- Geração das Chaves: Para iniciar o processo, dois números primos grandes, *p* e *q*, são escolhidos. Esses números são multiplicados para formar o número *n* (*n* = *pq*), que serve como um dos componentes da chave pública. Em seguida, é calculada a função totiente de *n*, $\phi(n) = (p-1)(q-1)$, a partir da qual será escolhida uma chave de criptografia pública, *e*, e uma chave de descriptografia privada, *d*, de modo que $(e \times d) mod\ \phi(n) = 1$.
- Criptografia: Para criptografar uma mensagem *M*, ela é convertida em um número menor que *n*. Em seguida, aplica-se a fórmula de criptografia:

$$C = M^e\ mod\ n$$

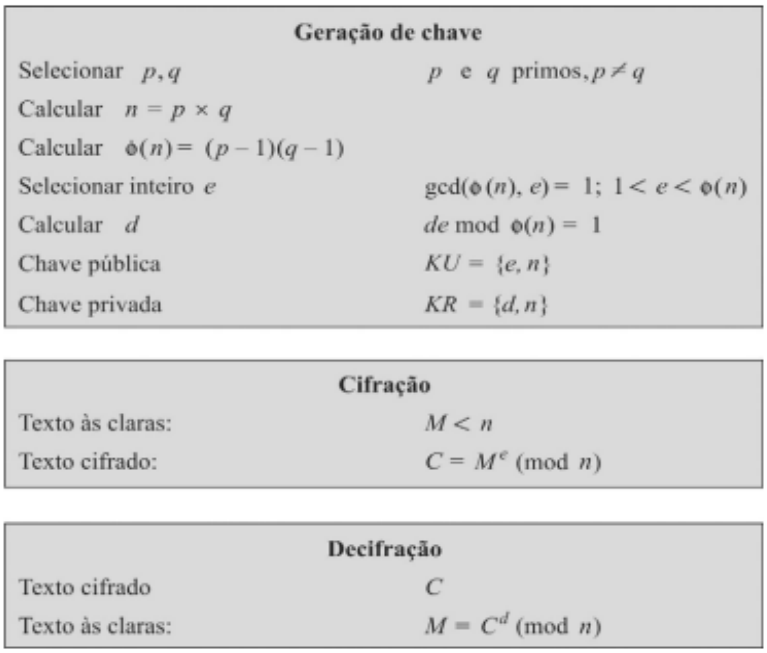
onde *C* é o texto cifrado enviado para o destinatário.

- Descriptografia: Ao receber o texto cifrado *C*, o destinatário utiliza sua chave privada *d* e calcula o texto original:

$$M = C^d\ mod\ n$$

A figura 10 mostra o processo de criptografia e decriptação no algoritmo RSA, composto por três etapas principais: geração de chave, cifração e decifração.

Figura 10 - O algoritmo RSA.



Geração de chave	
Selecionar p, q	p e q primos, $p \neq q$
Calcular $n = p \times q$	
Calcular $\phi(n) = (p-1)(q-1)$	
Selecionar inteiro e	$\gcd(\phi(n), e) = 1;\ 1 < e < \phi(n)$
Calcular d	$de \bmod \phi(n) = 1$
Chave pública	$KU = \{e, n\}$
Chave privada	$KR = \{d, n\}$

Cifração	
Texto às claras:	$M < n$
Texto cifrado:	$C = M^e \pmod n$

Decifração	
Texto cifrado	C
Texto às claras:	$M = C^d \pmod n$

Fonte: STALLINGS, 2015.

Exemplo didático de utilização do RSA: Adotando p = 17 e q = 11. Primeiramente, calcula-se n, $\phi(n)$:

$$n = p * q = 17 * 11 = 187$$

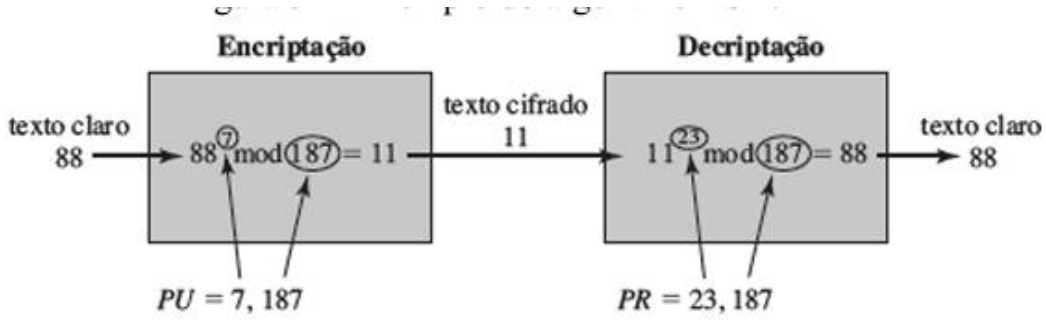
$$\phi\,(n) = (p-1) * (q-1) = (17-1) * (11-1) = 160$$

Em seguida, é escolhido e tal que $mdc(160, e) = 1$. Para este exemplo, adota-se e = 7. Depois, determina-se d a partir de e:

$$d = e^{-1}(mod\ \phi(n)) = 7-1\ (mod\ 160) = 23$$

Assim, tem-se PU = [7,187] e PR = [23,187]. A Figura 11 mostra um exemplo da encriptação e decriptação de uma mensagem com estes dados.

Figura 11 - Exemplo do algoritmo rsa.



Fonte: STALLINGS, 2015.

A segurança do RSA é baseada na dificuldade de fatorar grandes números primos, o que o torna seguro contra a maioria dos ataques de criptoanálise praticáveis (SCHNEIER, 2015).

A complexidade do RSA e a dificuldade de fatorar *n* garantem a segurança dos dados, especialmente em contextos de troca de informações sensíveis em redes abertas como a internet.

Embora o RSA seja amplamente seguro, ele apresenta algumas limitações, como a baixa velocidade em comparação com algoritmos simétricos, o que pode ser um problema em transações que requerem alta performance. Além disso, o aumento da capacidade de processamento dos computadores e o advento da computação quântica representam potenciais ameaças à segurança do RSA.

Como resposta, muitos sistemas combinam o RSA com criptografia simétrica para garantir tanto segurança quanto eficiência, prática comum em protocolos como o TLS (*Transport Layer Security*).

CAPÍTULO III - FUNÇÕES HASH E AUTENTICAÇÃO DE USUÁRIOS

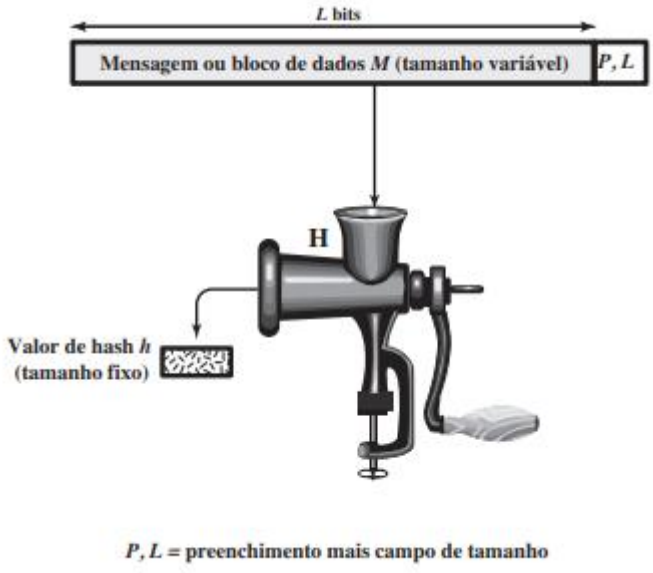
Este capítulo explora o conceito de função *hash*, explica a sua importância na parte de autenticação de usuários e como foi feita sua implementação no projeto.

3.1 INTRODUÇÃO

A segurança na autenticação de usuários é um dos aspectos mais críticos no desenvolvimento de sistemas de comunicação digital. Com o aumento das ameaças cibernéticas, garantir que apenas usuários autenticados tenham acesso a um sistema é essencial para proteger informações sensíveis. Nesse contexto, as funções *hash* desempenham um papel fundamental na autenticação de usuários, permitindo armazenar senhas de forma segura e impedir ataques de engenharia reversa.

Stallings (2015) afirma que uma “boa” função de *hash* tem a propriedade de que os resultados da aplicação da função a um grande conjunto de entradas produzirão saídas que são distribuídas por igual e aparentemente de modo aleatório. De acordo com o autor, no funcionamento de uma função *hash*, conforme mostrado na Figura 12, a entrada é ajustada para que seu tamanho seja um múltiplo inteiro de um valor fixo. Esse ajuste inclui um preenchimento que contém o tamanho da mensagem original, expresso em *bits*. Segundo Stallings (2015) o campo de tamanho é uma medida de segurança a fim de aumentar a dificuldade para um invasor produzir uma mensagem alternativa com o mesmo valor de *hash*.

Figura 12 - Função de hash criptográfica h = H(M).



Fonte: STALLINGS, 2015.

Devido a essas propriedades, as funções *hash* são amplamente utilizadas na autenticação de usuários, pois permitem armazenar senhas de forma segura sem a necessidade de guardar o texto original.

3.2 UTILIZAÇÃO NA AUTENTICAÇÃO DE USUÁRIOS

No *chat* seguro desenvolvido em *Rust*, a autenticação dos usuários utiliza funções *hash* para garantir a segurança das senhas armazenadas. Para isso, foi utilizado o *bcrypt*, que se destaca por ser especificamente projetado para proteger senhas contra-ataques de força bruta e tentativas de quebra por dicionário. O *bcrypt* é baseado no algoritmo de cifra de bloco *Blowfish*, mas utiliza uma variante chamada *EksBlowfish* (*Expensive Key Schedule Blowfish*).

O algoritmo *EksBlowfish* é uma modificação da função de expansão de chave do algoritmo de cifra de bloco *Blowfish*, desenvolvida especificamente para o *bcrypt* com o objetivo de aumentar o custo computacional da derivação de senhas. Enquanto o *Blowfish* tradicional possui uma rotina de expansão de chave projetada para ser rápida, o *EksBlowfish* torna essa etapa intencionalmente lenta e custosa, dificultando ataques de força bruta e ataques paralelos utilizando hardware especializado, como GPUs e ASICs.

O funcionamento do *EksBlowfish* envolve três entradas principais: o *salt*, a senha e o fator de custo(*cost*). Inicialmente, ele realiza uma etapa de expansão da chave, que consiste em inserir a senha e o *salt* nos *arrays* de *subchaves* (*P-array*) e nas caixas de substituição (*S-boxes*) internas do *Blowfish*. Em seguida, o algoritmo executa 2^{cost} iterações de uma rotina que aplica repetidamente o próprio algoritmo de cifragem para embaralhar ainda mais esses dados internos. Esse processo faz com que qualquer pequena alteração na senha ou no salt resulte em um *hash* completamente diferente. O tempo de execução do algoritmo cresce exponencialmente com o fator de custo, permitindo que o algoritmo se mantenha seguro à medida que a capacidade de processamento dos atacantes evolui. Assim, o *EksBlowfish* é o principal componente que garante a resiliência do *bcrypt* contra-ataques de escalabilidade e força bruta ao longo do tempo.

No sistema implementado, as senhas informadas pelos usuários no momento do cadastro são imediatamente processadas pelo *bcrypt*, que aplica o algoritmo de *hash* junto com um *salt* aleatório. O resultado é armazenado no banco de dados, substituindo a senha original. Isso significa que, mesmo que o banco seja comprometido, não é possível recuperar as senhas em texto claro.

Esse método proporciona uma autenticação segura, sem a necessidade de armazenar senhas reais, garantindo conformidade com boas práticas de segurança da informação.

A utilização de funções *hash* na autenticação de usuários é essencial para garantir a segurança das credenciais no *chat* seguro desenvolvido em *Rust.* Com a implementação adequada dessas técnicas, é possível proteger os usuários contra acessos indevidos, ataques de força bruta e vazamento de senhas, garantindo um ambiente de comunicação mais confiável e seguro.

3.3 FUNÇÃO DE HASH SHA-256

A função SHA-256 (Secure Hash Algorithm 256 bits) pertence à família de algoritmos SHA-2, desenvolvida pela Agência de Segurança Nacional dos Estados Unidos (NSA) e padronizada pelo Instituto Nacional de Padrões e Tecnologia (NIST), conforme especificado na publicação NIST (2015). Trata-se de uma função de hash amplamente utilizada para garantir integridade e autenticidade de dados em diversos contextos, como assinaturas digitais, autenticação de senhas e verificação de

arquivos. O funcionamento do SHA-256 é baseado em um processo de compressão iterativo que transforma blocos de dados de 512 *bits* em uma saída final de 256 *bits*. Inicialmente, a mensagem de entrada é preenchida para que seu tamanho seja múltiplo de 512 *bits*, incluindo um sufixo com o comprimento original da mensagem. Em seguida, a entrada é dividida em blocos, e cada bloco é processado por meio de operações lógicas e aritméticas, como rotações, deslocamentos e funções booleanas, utilizando constantes fixas e valores iniciais definidos pelo padrão. Em cada iteração, o bloco influencia o estado interno de oito registradores de 32 *bits*, que são atualizados com base em funções de mistura. Após o processamento de todos os blocos, o resultado concatenado dos registradores forma o *digest* final de 256 *bits*. Esse processo garante que qualquer modificação mínima na entrada produza uma saída completamente diferente, caracterizando o efeito avalanche.

Segundo Stallings (2015), uma função de *hash* criptográfica deve atender a três propriedades fundamentais: (1) resistência à pré-imagem, que torna inviável descobrir a mensagem original a partir de seu *hash*; (2) resistência à segunda pré-imagem, que impede encontrar uma segunda mensagem com o mesmo *hash* de uma mensagem dada; e (3) resistência à colisão, isto é, dificuldade de encontrar duas entradas diferentes que produzam o mesmo *hash*. O SHA-256 foi projetado para atender a essas propriedades com alto nível de segurança.

CAPÍTULO IV – IMPLEMENTAÇÃO E RESULTADO ALCANÇADOS

Durante o desenvolvimento do projeto, foram utilizadas tecnologias que permitiram a construção de uma aplicação segura. A seguir, são descritas as principais ferramentas, linguagens e bibliotecas adotadas.

4.1 VISUAL STUDIO CODE (VSCODE)

O *Visual Studio Code* foi o ambiente de desenvolvimento integrado (IDE) escolhido para a implementação deste projeto. Trata-se de uma ferramenta gratuita, de código aberto e multiplataforma, amplamente utilizada por desenvolvedores em diversas linguagens.

Sua interface leve e personalizável, aliada a uma vasta gama de extensões, foi fundamental para proporcionar uma experiência de desenvolvimento produtiva com a linguagem *Rust*. Foram utilizadas extensões específicas como o *rust-analyzer*, que oferecem funcionalidades como autocompletar, navegação entre arquivos, verificação de erros em tempo real e sugestões de refatoração, tudo isso diretamente no editor.

4.2 LINGUAGEM *RUST*

Rust é uma linguagem de programação moderna, focada em segurança, desempenho e concorrência. Criada pela *Mozilla Research*, ela se destaca por oferecer garantia de segurança de memória sem a necessidade de um coletor de lixo (*garbage collector*), tornando-se uma excelente escolha para aplicações que exigem alta eficiência e confiabilidade, como sistemas embarcados, desenvolvimento de software de baixo nível e aplicações seguras.

Segundo Klabnik e Nichols (2025) *rust* permite que desenvolvedores trabalhem com código de baixo nível sem assumir o risco usual de falhas ou brechas de segurança. O compilador de *Rust* atua como uma barreira protetora, impedindo que erros comuns, como condições de corrida e falhas de memória, cheguem à fase de execução do programa.

4.2.1 CARACTERÍSTICAS PRINCIPAIS DA LINGUAGEM RUST

Rust apresenta diversas características inovadoras que a tornam uma escolha atrativa para o desenvolvimento de software seguro e eficiente:

- Segurança de Memória: *Rust* evita erros comuns como *segmentation faults* e vazamento de memória ao utilizar o conceito de *ownership* (posse),

garantindo que apenas uma parte do código possa modificar um determinado recurso por vez;

- Concorrência Segura: Diferente de outras linguagens, *Rust* impede acessos concorrentes inseguros a dados compartilhados, evitando problemas como *data races* sem comprometer o desempenho;
- Alto Desempenho: Com um tempo de execução próximo ao de C e C++, *Rust* é ideal para aplicações que exigem eficiência e controle sobre os recursos de hardware;
- Sistema de Tipos Avançado: A tipagem estática previne erros comuns em tempo de compilação, aumentando a robustez do código e reduzindo a quantidade de falhas em tempo de execução;
- Ecossistema e Ferramentas Modernas: *Rust* conta com um gerenciador de pacotes (Cargo), um sistema de compilação eficiente e uma comunidade ativa que contribui para a evolução da linguagem.

4.2.2 GERENCIAMENTO DE MEMÓRIA EM RUST

Uma das maiores inovações do *Rust* é seu modelo de gerenciamento de memória baseado em posse e empréstimos (*ownership* e *borrowing*). Diferente de linguagens como Java e Python, que utilizam coleta de lixo, *Rust* garante segurança e eficiência através de:

- *Ownership* (Posse): Cada valor tem um único dono e é desalocado automaticamente quando sai do escopo.
- *Borrowing* (Empréstimo): Permite que várias partes do código acessem dados sem precisar copiá-los, desde que sigam regras estritas para evitar concorrência insegura.
- *Lifetimes* (Tempos de Vida): Garantem que as referências permaneçam válidas durante sua utilização.

Esses mecanismos ajudam a evitar problemas como vazamento de memória e ponteiros nulos, comuns em C e C++.

Rust é uma linguagem de programação que equilibra segurança e desempenho, tornando-se uma opção ideal para aplicações que exigem confiabilidade. Seu sistema de gerenciamento de memória inovador e seus mecanismos de segurança a tornam uma escolha poderosa para o desenvolvimento

de sistemas seguros, como o *chat* criptografado apresentado neste trabalho. A adoção de *Rust* para essa finalidade permite garantir um ambiente robusto e confiável para a troca de mensagens seguras entre usuários.

4.3 POSTGRESQL

O PostgreSQL foi escolhido como sistema gerenciador de banco de dados para a aplicação. Trata-se de um banco de dados relacional de código aberto, altamente confiável, estável e com mais de 30 anos de desenvolvimento ativo.

Dentre os principais motivos para sua escolha, destacam-se o suporte avançado a tipos de dados complexos, o rigor na conformidade com o padrão SQL, o suporte a transações ACID (Atomicidade, Consistência, Isolamento e Durabilidade), e a facilidade de integração com aplicações escritas em *Rust* através de bibliotecas como *sqlx* e *tokio_postgres*.

Na aplicação de *chat*, o PostgreSQL foi responsável por armazenar de forma persistente as informações dos usuários e o histórico de mensagens trocadas entre eles.

A criação das tabelas foi realizada programaticamente durante a inicialização da aplicação, o que garante que o banco esteja preparado para uso mesmo em ambientes novos, facilitando a implantação e escalabilidade futura do sistema.

4.4 ALGORITMOS UTILIZADOS

No contexto da segurança da informação, a escolha dos algoritmos criptográficos é fundamental para garantir a confidencialidade, integridade e autenticidade dos dados. Neste trabalho, foram utilizados três algoritmos principais: AES, RSA, PBKDF2 com HMAC-SHA256 e *bcrypt*, cada um com propósitos específicos dentro da arquitetura do sistema de *chat* seguro.

O AES é o padrão oficial de criptografia simétrica definido pelo NIST, reconhecido mundialmente por sua segurança e confiabilidade. Trata-se de um algoritmo que opera sobre blocos de 128 *bits* e permite o uso de chaves de 128, 192 ou 256 *bits*, realizando múltiplas rodadas de substituição, permutação e difusão para transformar o texto claro em texto cifrado. No sistema proposto, o AES é utilizado para criptografar as mensagens trocadas entre os usuários, após a realização de uma troca segura de chaves, garantindo que o conteúdo das conversas permaneça sigiloso mesmo em caso de interceptação. Além disso, o AES foi empregado para criptografar

a chave privada RSA dos usuários antes de ser armazenada no banco de dados, utilizando uma chave derivada da senha do usuário, o que eleva significativamente a segurança do armazenamento.

O RSA é um algoritmo de criptografia assimétrica baseado na dificuldade de fatoração de grandes números primos. Ele opera com um par de chaves: uma pública, usada para criptografar dados, e outra privada, usada para descriptografá-los. No *chat* desenvolvido, o RSA foi utilizado para realizar a troca segura da chave de sessão AES entre os usuários. Ao enviar uma mensagem, o remetente criptografa a chave AES com a chave pública do destinatário, garantindo que apenas ele, com sua chave privada, possa descriptografar e acessar a mensagem. Essa abordagem elimina a necessidade de troca de chaves simétricas por meios inseguros, reforçando a privacidade das comunicações.

O PBKDF2 é uma técnica padronizada pela RFC 8018 que transforma uma senha em uma chave criptográfica segura, aplicando um número elevado de iterações de uma função *hash* (no caso, HMAC-SHA256) e um valor aleatório conhecido como *salt*. Essa combinação dificulta ataques de força bruta e ataques de dicionário, pois cada senha resulta em uma chave única, mesmo que duas pessoas escolham a mesma senha.

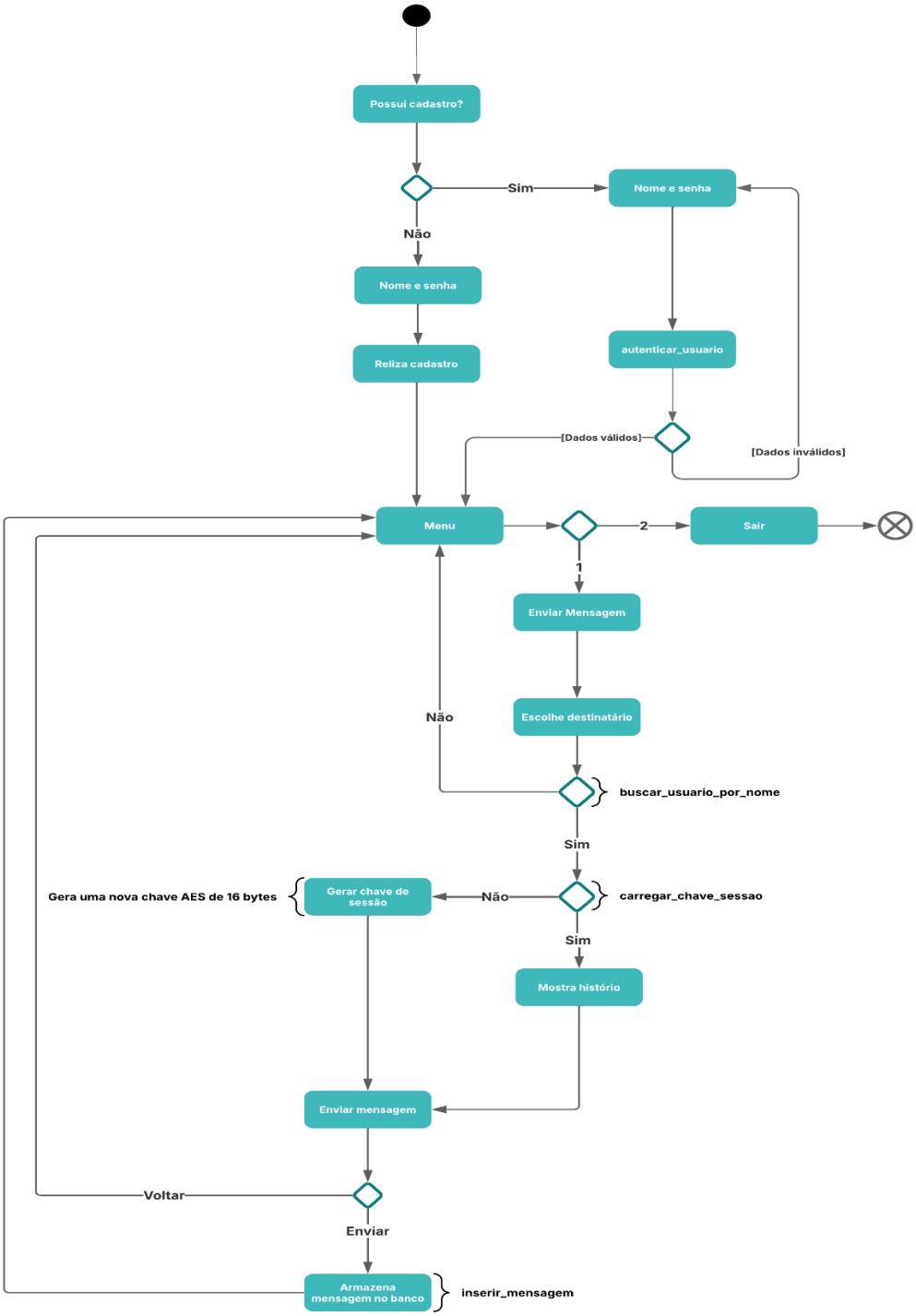
Por fim, o algoritmo *bcrypt* foi utilizado para proteger as senhas dos usuários. O *bcrypt* é uma função de *hash* baseada no algoritmo *Blowfish* e foi projetado especificamente para armazenar senhas de forma segura. Ele incorpora um *salt* único e aplica múltiplas iterações de *hashing*, o que dificulta significativamente ataques de força bruta ou dicionário. No sistema proposto, as senhas fornecidas pelos usuários são convertidas em *hash* usando *bcrypt* antes de serem armazenadas no banco de dados, impedindo a reversão mesmo em caso de vazamento de dados.

4.5 DIAGRAMAS DO SISTEMA

Para facilitar a compreensão da lógica de funcionamento da aplicação, foi elaborado um diagrama de atividade, mostrado na Figura 13, que representa a execução do sistema. Esse diagrama destaca as principais etapas percorridas pelo usuário, desde o momento do login ou cadastro até o envio de mensagens. Além disso, também foi desenvolvido um diagrama de sequência, mostrado nas Figuras 14 e 15, que detalha a interação entre os componentes do sistema (cliente, servidor e

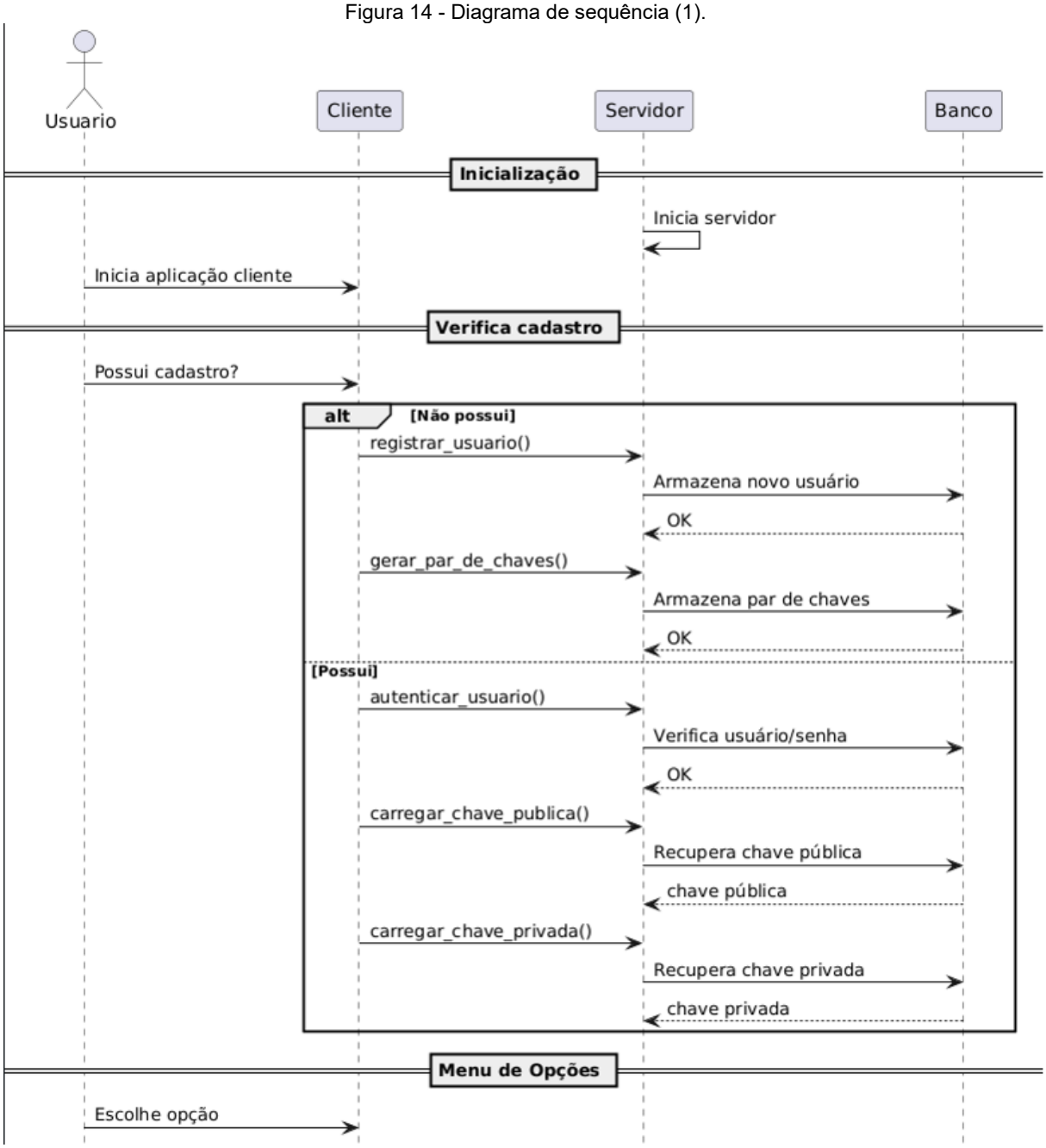
banco de dados). A combinação desses diagramas contribui para uma visualização clara e estruturada do fluxo da aplicação e das responsabilidades de cada elemento envolvido.

Figura 13 - Diagrama de atividades.



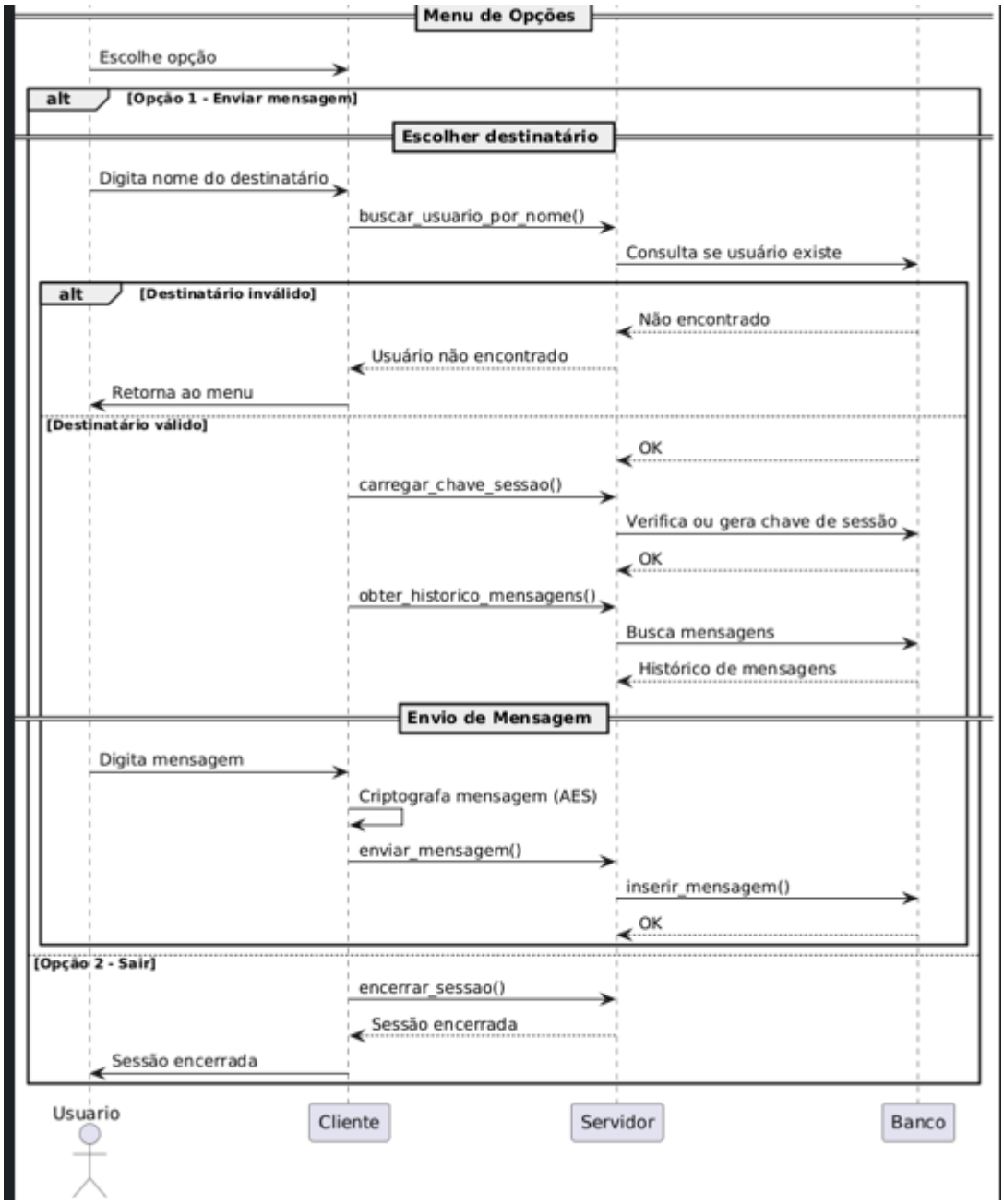
Fonte: Autoria própria.

Figura 14 - Diagrama de sequência (1).



Fonte: Autoria própria.

Figura 15 - Diagrama de sequência (2).



Fonte: Autoria própria.

4.6 IMPLEMENTAÇÃO DO MÓDULO DE BANCO DE DADOS

A estrutura de persistência de dados do sistema foi implementada no módulo *database.rs,* responsável por gerenciar o armazenamento e a recuperação de informações essenciais, como usuários, chaves criptográficas e mensagens trocadas

no *chat*. Para viabilizar esse gerenciamento, foram utilizadas duas bibliotecas distintas de acesso ao banco de dados PostgreSQL: t*okio-postgres* e *sqlx*.

A aplicação mantém duas conexões independentes com o banco, cada uma destinada a um tipo específico de operação:

- *tokio-postgres*: é usada para execuções diretas de comandos SQL, como criação de tabelas e inserções.;
- *sqlx*: é utilizada para operações que exigem mapeamento automático entre registros do banco e estruturas *Rust*, como nas consultas de mensagens. Seu sistema de pool de conexões assíncronas melhora a escalabilidade e facilita o uso em aplicações concorrentes.

Além disso, o módulo cuida da criptografia da chave privada RSA dos usuários antes de armazená-la no banco, e implementa funções seguras para recuperação, autenticação e gerenciamento de chaves de sessão AES.

4.6.1 FUNÇÃO NEW()

A função *new()* tem como objetivo inicializar toda a infraestrutura de banco de dados utilizada pela aplicação. O processo inicia com a criação de uma conexão ao PostgreSQL por meio da biblioteca *tokio-postgres*, utilizada para a execução direta de comandos SQL como a criação das tabelas *usuarios*, mensagens e *chaves_sessoes*. Em paralelo, uma segunda conexão é aberta com a biblioteca *sqlx*, responsável por manter um pool de conexões assíncronas, permitindo que múltiplas requisições sejam processadas de forma eficiente e escalável.

Durante a inicialização, a função *new()* garante que as seguintes tabelas estejam devidamente criadas:

- *usuarios*: armazena os nomes de usuário, o *hash* da senha (gerado com o algoritmo *bcrypt*), a chave pública RSA e a chave privada RSA criptografada com AES, aumentando significativamente a segurança dos dados sensíveis;
- mensagens: guarda o histórico completo das mensagens trocadas entre usuários, com os campos de remetente, destinatário, conteúdo da mensagem criptografado e data/hora;
- *chaves_sessoes*: gerencia as chaves AES utilizadas em sessões seguras entre pares de usuários. A chave é armazenada duas vezes, cada uma

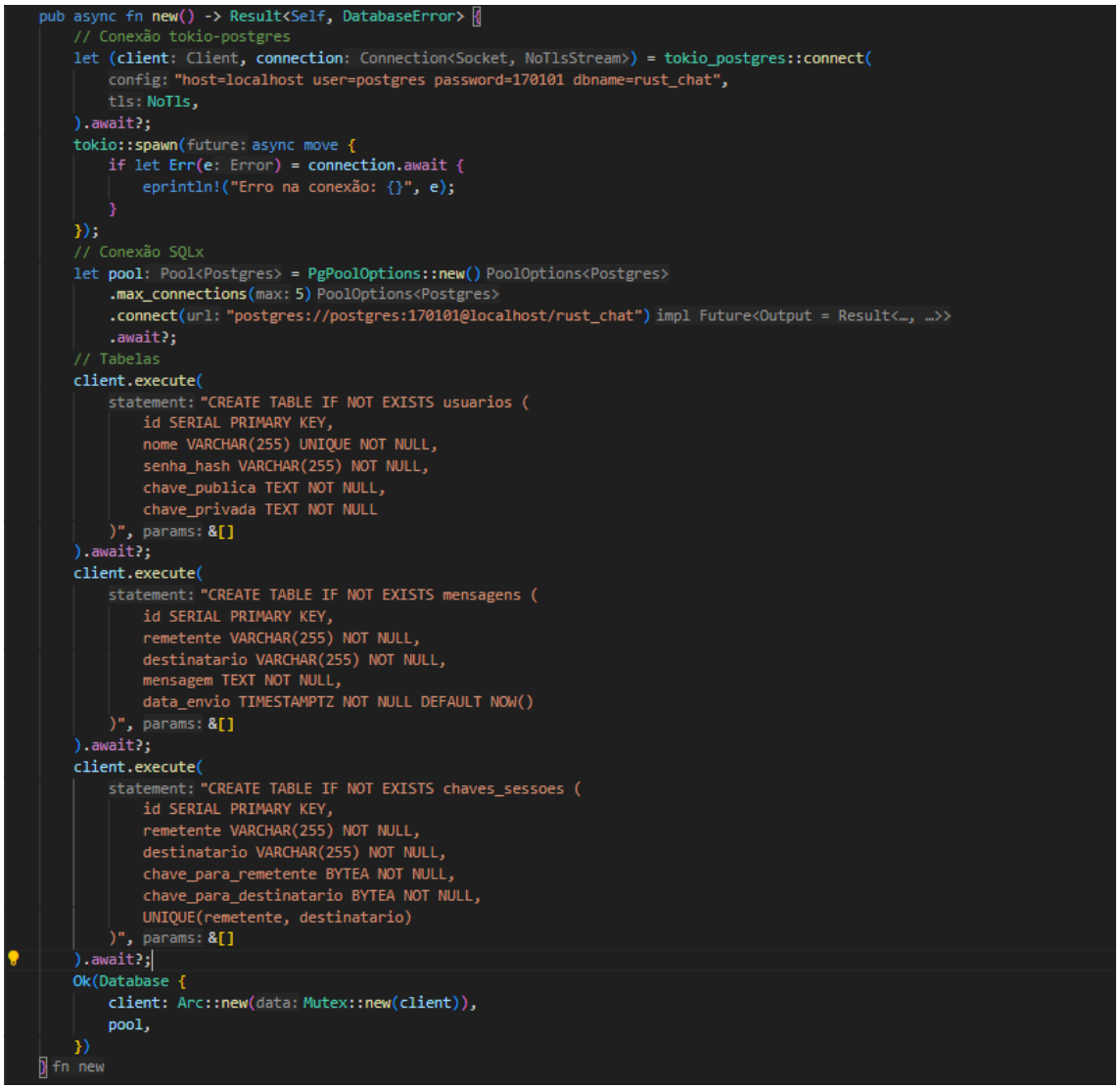
criptografada com a chave pública do respectivo destinatário, permitindo o compartilhamento seguro.

Ao final da execução, a função retorna uma instância da estrutura *Database*, que encapsula:

- um *Arc<Mutex<Client>>*, representando a conexão do *tokio-postgres* protegida por mutex para garantir segurança no ambiente assíncrono;
- e um *PgPool*, gerenciado pela *sqlx*, permitindo o acesso eficiente a partir de diversas partes do sistema.

Esse modelo de arquitetura proporciona flexibilidade, eficiência e segurança, sendo adequado para aplicações em tempo real como o *chat* seguro proposto, conforme mostrado na Figura 16.

Figura 16 - Função responsável por retornar a estrutura do banco de dados.



```rust
pub async fn new() -> Result<Self, DatabaseError> {
    // Conexão tokio-postgres
    let (client: Client, connection: Connection<Socket, NoTlsStream>) = tokio_postgres::connect(
        config: "host=localhost user=postgres password=170101 dbname=rust_chat",
        tls: NoTls,
    ).await?;
    tokio::spawn(future: async move {
        if let Err(e: Error) = connection.await {
            eprintln!("Erro na conexão: {}", e);
        }
    });
    // Conexão SQLx
    let pool: Pool<Postgres> = PgPoolOptions::new() PoolOptions<Postgres>
        .max_connections(max: 5) PoolOptions<Postgres>
        .connect(url: "postgres://postgres:170101@localhost/rust_chat") impl Future<Output = Result<…, …>>
        .await?;
    // Tabelas
    client.execute(
        statement: "CREATE TABLE IF NOT EXISTS usuarios (
            id SERIAL PRIMARY KEY,
            nome VARCHAR(255) UNIQUE NOT NULL,
            senha_hash VARCHAR(255) NOT NULL,
            chave_publica TEXT NOT NULL,
            chave_privada TEXT NOT NULL
        )", params: &[]
    ).await?;
    client.execute(
        statement: "CREATE TABLE IF NOT EXISTS mensagens (
            id SERIAL PRIMARY KEY,
            remetente VARCHAR(255) NOT NULL,
            destinatario VARCHAR(255) NOT NULL,
            mensagem TEXT NOT NULL,
            data_envio TIMESTAMPTZ NOT NULL DEFAULT NOW()
        )", params: &[]
    ).await?;
    client.execute(
        statement: "CREATE TABLE IF NOT EXISTS chaves_sessoes (
            id SERIAL PRIMARY KEY,
            remetente VARCHAR(255) NOT NULL,
            destinatario VARCHAR(255) NOT NULL,
            chave_para_remetente BYTEA NOT NULL,
            chave_para_destinatario BYTEA NOT NULL,
            UNIQUE(remetente, destinatario)
        )", params: &[]
    ).await?;
    Ok(Database {
        client: Arc::new(data: Mutex::new(client)),
        pool,
    })
} fn new
```

Fonte: Autoria própria.

4.6.2 REGISTRO E AUTENTICAÇÃO DE USUÁRIOS

As funções responsáveis pelo registro e autenticação de usuários no sistema são fundamentais para garantir a segurança e o controle de acesso à aplicação de mensagens. A função r*egistrar_usuario* é acionada sempre que um novo usuário deseja se cadastrar na plataforma. Nessa etapa, a senha informada pelo usuário não é armazenada em texto claro: ela é imediatamente convertida em um *hash* seguro por meio do algoritmo *bcrypt,* utilizando o custo padrão, o que torna o processo de geração computacionalmente caro, desestimulando ataques por força bruta. Esse *hash* é então armazenado na tabela *usuarios*, juntamente com o nome de usuário, a chave pública

RSA e a chave privada RSA previamente criptografada com uma chave derivada da senha do usuário, agregando uma camada extra de proteção.

Durante o processo de login, a função *autenticar_usuario* entra em ação. Ela recupera o *hash* correspondente ao nome de usuário informado e utiliza novamente o *bcrypt*, agora para comparar o valor digitado com o *hash* armazenado. A autenticação só é bem-sucedida caso o resultado da verificação seja positivo. Dessa forma, mesmo em um cenário de comprometimento do banco de dados, as senhas reais dos usuários continuam protegidas, uma vez que o processo de reversão do *hash bcrypt* é impraticável.

Esse conjunto de funções estabelece um mecanismo de autenticação, essencial para garantir a integridade e privacidade das comunicações entre usuários no sistema. A Figura 17 mostra um trecho do código correspondente à implementação das rotinas de registro e autenticação.

Figura 17 - Funções responsáveis pelo registro e autenticação de usuários.

```rust
// Registro com chaves
pub async fn registrar_usuario(
    &self, nome: &str, senha: &str, chave_privada: &str, chave_publica: &str
) -> Result<(), DatabaseError> {
    let senha_hash: String = hash(password: senha, DEFAULT_COST)?;
    let client: MutexGuard<'_, Client> = self.client.lock().await;
    client.execute(
        statement: "INSERT INTO usuarios (nome, senha_hash, chave_publica, chave_privada) VALUES ($1, $2, $3, $4)",
        params: &[&nome, &senha_hash, &chave_publica, &chave_privada],
    ).await?;
    Ok(())
}

pub async fn autenticar_usuario(&self, nome: &str, senha: &str) -> Result<bool, DatabaseError> {
    let client: MutexGuard<'_, Client> = self.client.lock().await;
    let row: Option<Row> = client.query_opt(
        statement: "SELECT senha_hash FROM usuarios WHERE nome = $1",
        params: &[&nome],
    ).await?;

    if let Some(row: Row) = row {
        let senha_hash: String = row.get(idx: "senha_hash");
        Ok(verify(password: senha, &senha_hash)?)
    } else {
        Ok(false)
    }
}
```

Fonte: Autoria própria.

4.6.3 ARMAZENAMENTO DE MENSAGENS

O armazenamento de mensagens no sistema é realizado de forma organizada e segura por meio da função *inserir_mensagem*. Essa função é responsável por gravar no banco de dados todas as mensagens trocadas entre os usuários da aplicação. Sempre que um usuário envia uma mensagem, o sistema realiza a

criptografia do conteúdo com uma chave AES previamente acordada, e o resultado é codificado em *Base64* (transforma dados binários em uma *string*). Esse conteúdo criptografado, juntamente com o nome do remetente, o nome do destinatário e a data e hora do envio, é então armazenado na tabela mensagens do banco de dados *PostgreSQL*. A marcação temporal é gerada automaticamente pelo próprio banco, garantindo a consistência cronológica das conversas sem depender do relógio da aplicação cliente.

Esse modelo de armazenamento permite a rastreabilidade completa das conversas, possibilitando consultas futuras por meio da função de recuperação de histórico, também disponível no sistema. A estrutura da tabela e o modo como os dados são organizados favorecem tanto a segurança quanto a clareza da persistência das comunicações.

Como mostrado na Figura 18, a função *inserir_mensagem* mostra como é realizada de forma direta e segura, promovendo a integridade do histórico de mensagens da aplicação.

Figura 18 - Função de inserir mensagens no banco.

```rust
// Mensagens (criptografadas)
pub async fn inserir_mensagem(
    &self,
    remetente: &str,
    destinatario: &str,
    mensagem_criptografada_base64: &str
) -> Result<(), DatabaseError> {
    let client: MutexGuard<'_, Client> = self.client.lock().await;

    client.execute(
        statement: "INSERT INTO mensagens (remetente, destinatario, mensagem) VALUES ($1, $2, $3)",
        params: &[&remetente, &destinatario, &mensagem_criptografada_base64],
    ).await?;

    Ok(())
}
```

Fonte: Autoria própria.

4.6.4 HISTÓRICO DE MENSAGENS

A função *obter_historico_mensagens* é responsável por recuperar o histórico completo de mensagens trocadas entre dois usuários. Ela realiza uma consulta na tabela mensagens do banco de dados, buscando todas as mensagens onde o remetente é igual ao primeiro usuário e o destinatário é o segundo, ou vice-versa,

garantindo assim que todas as conversas entre ambos sejam consideradas, independentemente de quem enviou a mensagem.

Essa função utiliza o método *query_as* da biblioteca *sqlx* para mapear diretamente os resultados da consulta SQL para a estrutura Mensagem definida anteriormente no código.

O retorno da função é um vetor com todas as mensagens ordenadas de forma crescente com base na data de envio, o que facilita a exibição cronológica das conversas. Esse histórico é fundamental para que os usuários possam visualizar conversas anteriores e retomar interações com base em mensagens passadas. A Figura 19 mostra o código responsável por implementar essa funcionalidade no sistema.

Figura 19 - Função para obter histórico de mensagens.

```rust
pub async fn obter_historico_mensagens(
    &self,
    usuario1: &str,
    usuario2: &str,
) -> Result<Vec<Mensagem>, DatabaseError> {
    let mensagens: Vec<Mensagem> = sqlx::query_as::<_, Mensagem>(
        sql: r#"
        SELECT remetente, destinatario, mensagem,
               data_envio AT TIME ZONE 'America/Sao_Paulo' AS data_envio
        FROM mensagens
        WHERE (remetente = $1 AND destinatario = $2)
           OR (remetente = $2 AND destinatario = $1)
        ORDER BY data_envio ASC
        "#
    )QueryAs<'_, Postgres, Mensagem, …>
    .bind(usuario1) QueryAs<'_, Postgres, Mensagem, …>
    .bind(usuario2) QueryAs<'_, Postgres, Mensagem, …>
    .fetch_all(executor: &self.pool) impl Future<Output = Result<…, …>>
    .await?;

    Ok(mensagens)
}
```

Fonte: Autoria própria.

4.6.5 GERENCIAMENTO DE CHAVES PÚBLICAS E PRIVADAS (RSA)

O gerenciamento de chaves é realizado pelo RSA, permitindo a criptografia assimétrica segura entre os usuários. A função *carregar_chave_publica* é responsável por recuperar do banco de dados a chave pública de um determinado usuário. Essa

chave é armazenada no formato PEM e, ao ser lida, é convertida para o tipo *RsaPublicKey* da biblioteca *rsa*, permitindo seu uso direto na criptografia de mensagens ou de chaves de sessão.

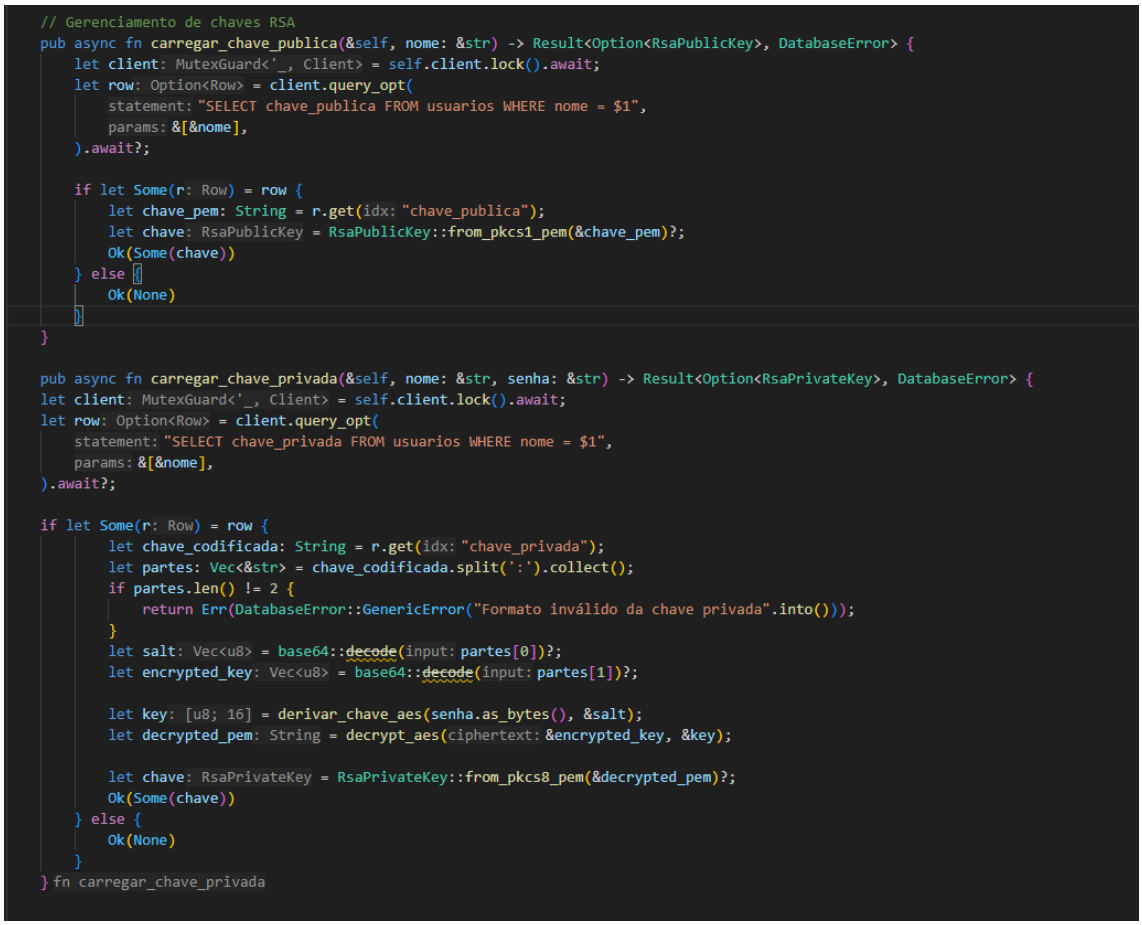
Já a função *carregar_chave_privada* implementa um nível adicional de segurança ao tratar a chave privada do usuário. Durante o registro, essa chave é criptografada utilizando o AES com uma chave derivada da senha do próprio usuário. No momento do *login*, para carregar a chave privada, o sistema:

- Recupera o valor armazenado no banco;
- Separa o *salt* e os dados criptografados;
- Deriva a chave AES com base na senha digitada;
- Descriptografa o conteúdo da chave privada;
- Reconstrói a chave *RsaPrivateKey* a partir do conteúdo PEM.

Esse processo impede que a chave privada fique exposta mesmo em caso de vazamento do banco de dados, já que o conteúdo está protegido por criptografia simétrica baseada em senha.

A Figura 20 mostra o funcionamento dessas duas funções e como o sistema trata a segurança das chaves RSA com criptografia adicional no armazenamento.

Figura 20 - Gerenciamento de chaves RSA.



```rust
// Gerenciamento de chaves RSA
pub async fn carregar_chave_publica(&self, nome: &str) -> Result<Option<RsaPublicKey>, DatabaseError> {
    let client: MutexGuard<'_, Client> = self.client.lock().await;
    let row: Option<Row> = client.query_opt(
        statement: "SELECT chave_publica FROM usuarios WHERE nome = $1",
        params: &[&nome],
    ).await?;

    if let Some(r: Row) = row {
        let chave_pem: String = r.get(idx: "chave_publica");
        let chave: RsaPublicKey = RsaPublicKey::from_pkcs1_pem(&chave_pem)?;
        Ok(Some(chave))
    } else {
        Ok(None)
    }
}

pub async fn carregar_chave_privada(&self, nome: &str, senha: &str) -> Result<Option<RsaPrivateKey>, DatabaseError> {
let client: MutexGuard<'_, Client> = self.client.lock().await;
let row: Option<Row> = client.query_opt(
    statement: "SELECT chave_privada FROM usuarios WHERE nome = $1",
    params: &[&nome],
).await?;

if let Some(r: Row) = row {
        let chave_codificada: String = r.get(idx: "chave_privada");
        let partes: Vec<&str> = chave_codificada.split(':').collect();
        if partes.len() != 2 {
            return Err(DatabaseError::GenericError("Formato inválido da chave privada".into()));
        }
        let salt: Vec<u8> = base64::decode(input: partes[0])?;
        let encrypted_key: Vec<u8> = base64::decode(input: partes[1])?;

        let key: [u8; 16] = derivar_chave_aes(senha.as_bytes(), &salt);
        let decrypted_pem: String = decrypt_aes(ciphertext: &encrypted_key, &key);

        let chave: RsaPrivateKey = RsaPrivateKey::from_pkcs8_pem(&decrypted_pem)?;
        Ok(Some(chave))
    } else {
        Ok(None)
    }
} fn carregar_chave_privada
```

Fonte: Autoria própria.

4.6.6 GERENCIAMENTO DE CHAVES DE SESSÃO (AES)

Para viabilizar a comunicação segura entre dois usuários, o sistema implementa o conceito de chaves de sessão AES, que são compartilhadas apenas entre remetente e destinatário. Essas chaves são geradas dinamicamente e protegidas usando criptografia RSA.

A função *salvar_chave_sessao* é chamada quando uma nova chave de sessão for criada para comunicação entre dois usuários. Essa chave simétrica é criptografada duas vezes: uma com a chave pública do remetente e outra com a chave pública do destinatário. Ambas as versões criptografadas são armazenadas na tabela *chaves_sessoes*, permitindo que cada usuário recupere sua respectiva versão posteriormente.

A função *carregar_chave_sessao* permite que o usuário atual recupere a chave de sessão correspondente a uma conversa com outro usuário. O sistema determina

se o usuário é o remetente ou destinatário, e retorna a versão apropriada da chave, que pode ser posteriormente descriptografada com sua chave privada RSA.

Essa abordagem garante que apenas os dois participantes da comunicação possam acessar a chave de sessão usada na cifragem das mensagens, mantendo a confidencialidade e autenticidade das conversas. A Figura 21 mostra o trecho de código responsável por essas funcionalidades.

Figura 21 - Gerenciamento de chaves de sessão AES.

```rust
// Gerenciamento de chaves de sessão AES
pub async fn salvar_chave_sessao(
    &self,
    remetente: &str,
    destinatario: &str,
    chave_para_remetente: &[u8],
    chave_para_destinatario: &[u8],
) -> Result<(), DatabaseError> {
    let client: MutexGuard<'_, Client> = self.client.lock().await;
    client.execute(
        statement: "INSERT INTO chaves_sessoes (remetente, destinatario, chave_para_remetente, chave_para_destinatario)
         VALUES ($1, $2, $3, $4)
         ON CONFLICT (remetente, destinatario)
         DO UPDATE SET chave_para_remetente = $3, chave_para_destinatario = $4",
        params: &[&remetente, &destinatario, &chave_para_remetente, &chave_para_destinatario],
    ).await?;

    Ok(())
}
pub async fn carregar_chave_sessao(
    &self,
    usuario_atual: &str,
    outro_usuario: &str,
) -> Result<Option<Vec<u8>>, DatabaseError> {
    let client: MutexGuard<'_, Client> = self.client.lock().await;
    let row: Option<Row> = client.query_opt(
        statement: "SELECT chave_para_remetente, chave_para_destinatario, remetente, destinatario
         FROM chaves_sessoes
         WHERE (remetente = $1 AND destinatario = $2)
            OR (remetente = $2 AND destinatario = $1)",
        params: &[&usuario_atual, &outro_usuario],
    ).await?;
    if let Some(r: Row) = row {
        let remetente: String = r.get(idx: "remetente");
        let destinatario: String = r.get(idx: "destinatario");

        if usuario_atual == remetente {
            let chave: Vec<u8> = r.get(idx: "chave_para_remetente");
            Ok(Some(chave))
        } else if usuario_atual == destinatario {
            let chave: Vec<u8> = r.get(idx: "chave_para_destinatario");
            Ok(Some(chave))
        } else {
            Ok(None)
        }
    } else {
        Ok(None)
    }
} fn carregar_chave_sessao
```

Fonte: Autoria própria.

4.7 IMPLEMENTAÇÃO DA CRIPTOGRAFIA

O módulo *criptografia.rs* é responsável por fornecer as funções essenciais de segurança que sustentam a confidencialidade e autenticidade das comunicações no *chat*. Ele implementa operações com criptografia simétrica (AES), criptografia assimétrica (RSA), e ainda técnicas de derivação de chave seguras (PBKDF2).

A criptografia simétrica é realizada por meio do algoritmo AES no modo CBC com preenchimento PKCS7. A função *encrypt_aes* recebe os dados em texto claro e uma chave de 128 *bits*, gera um vetor de inicialização (IV) aleatório e realiza a criptografia. O IV é então concatenado ao início da mensagem cifrada para que a função *decrypt_aes*, que executa o processo inverso, consiga realizar a descriptografia corretamente. Esse método garante que mensagens iguais produzam saídas cifradas diferentes a cada execução, aumentando a segurança. A Figura 22 mostra o código correspondente à essas funções.

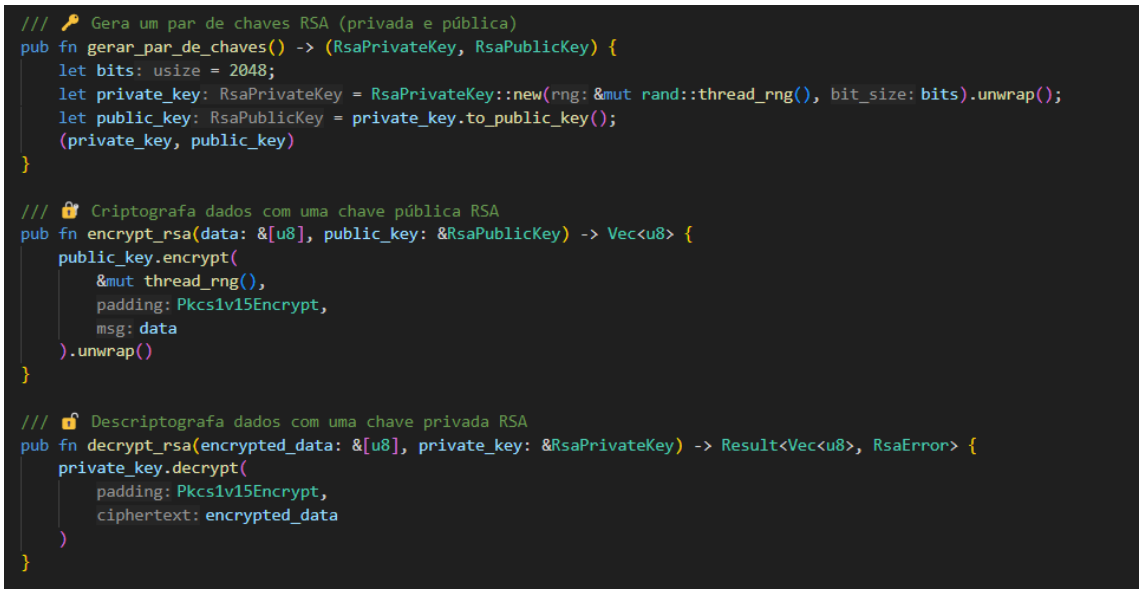
Figura 22 - Implementação de criptografia e descriptografia AES com IV aleatório embutido.

```rust
/// 🔒 Criptografa com AES usando IV aleatório embutido na mensagem
pub fn encrypt_aes(data: &str, key: &[u8; 16]) -> Vec<u8> {
    let iv: [u8; 16] = rand::thread_rng().gen();
    let cipher: Cbc<Aes128, Pkcs7> = Aes128Cbc::new_from_slices(key, &iv).unwrap();
    let mut ciphertext: Vec<u8> = cipher.encrypt_vec(plaintext: data.as_bytes());

    // Prefixa o IV no início da mensagem criptografada
    let mut result: Vec<u8> = iv.to_vec();
    result.append(&mut ciphertext);
    result
}

/// 🔓 Descriptografa com AES, extraindo o IV do início da mensagem
pub fn decrypt_aes(ciphertext: &[u8], key: &[u8; 16]) -> String {
    let (iv: &[u8], data: &[u8]) = ciphertext.split_at(mid: 16);
    let cipher: Cbc<Aes128, Pkcs7> = Aes128Cbc::new_from_slices(key, iv).unwrap();
    let decrypted: Vec<u8> = cipher.decrypt_vec(ciphertext: data).unwrap();
    String::from_utf8(vec: decrypted).unwrap()
}
```

Fonte: Autoria própria.

Para a criptografia assimétrica, foi utilizado o algoritmo RSA com chaves de 2048 *bits*. A função *gerar_par_de_chaves* é responsável por gerar o par de chaves RSA (pública e privada), utilizadas para proteger a troca de chaves simétricas entre os usuários. A função *encrypt_rsa* cifra os dados com a chave pública, enquanto *decrypt_rsa* realiza a descriptografia com a chave privada correspondente. Esse mecanismo mostrado na Figura 23 garante que somente o destinatário previsto possa decifrar a chave de sessão.

Figura 23 - Implementação de geração e uso de chaves RSA para criptografia assimétrica.

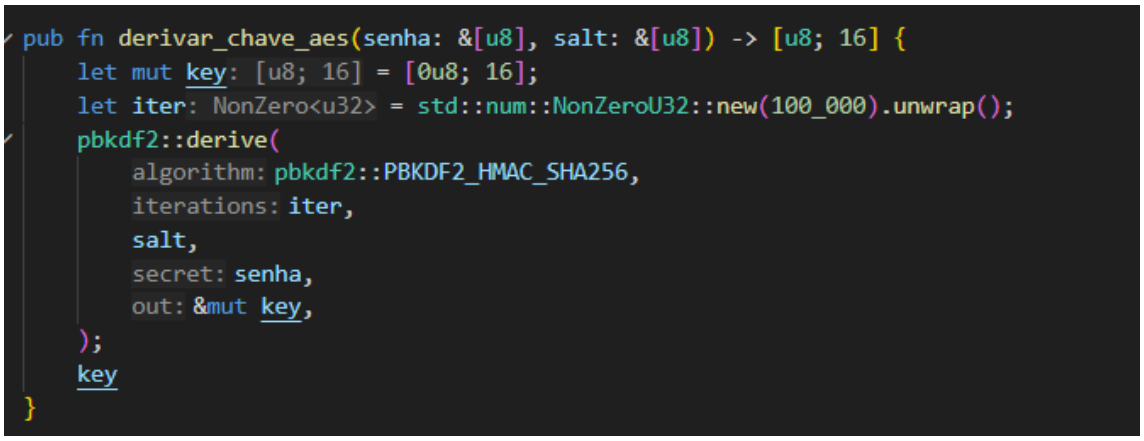


```rust
/// 🔑 Gera um par de chaves RSA (privada e pública)
pub fn gerar_par_de_chaves() -> (RsaPrivateKey, RsaPublicKey) {
    let bits: usize = 2048;
    let private_key: RsaPrivateKey = RsaPrivateKey::new(rng: &mut rand::thread_rng(), bit_size: bits).unwrap();
    let public_key: RsaPublicKey = private_key.to_public_key();
    (private_key, public_key)
}

/// 🔐 Criptografa dados com uma chave pública RSA
pub fn encrypt_rsa(data: &[u8], public_key: &RsaPublicKey) -> Vec<u8> {
    public_key.encrypt(
        &mut thread_rng(),
        padding: Pkcs1v15Encrypt,
        msg: data
    ).unwrap()
}

/// 🔓 Descriptografa dados com uma chave privada RSA
pub fn decrypt_rsa(encrypted_data: &[u8], private_key: &RsaPrivateKey) -> Result<Vec<u8>, RsaError> {
    private_key.decrypt(
        padding: Pkcs1v15Encrypt,
        ciphertext: encrypted_data
    )
}
```

Fonte: Autoria própria.

Outro ponto importante na segurança do sistema foi a adoção da função *derivar_chave_aes*, que utiliza o algoritmo PBKDF2 com HMAC-SHA256 para derivar uma chave AES segura a partir da senha do usuário e um valor de *salt* aleatório. Essa chave é usada especificamente para criptografar a chave privada RSA antes que ela seja armazenada no banco de dados, protegendo-a mesmo em caso de acesso indevido ao sistema de armazenamento. A Figura 24 mostra o código referente à essa função.

Figura 24 - Derivação de chave AES a partir de senha com PBKDF2.



```rust
pub fn derivar_chave_aes(senha: &[u8], salt: &[u8]) -> [u8; 16] {
    let mut key: [u8; 16] = [0u8; 16];
    let iter: NonZero<u32> = std::num::NonZeroU32::new(100_000).unwrap();
    pbkdf2::derive(
        algorithm: pbkdf2::PBKDF2_HMAC_SHA256,
        iterations: iter,
        salt,
        secret: senha,
        out: &mut key,
    );
    key
}
```

Fonte: Autoria própria.

4.8 IMPLEMENTAÇÃO RESPONSÁVEL PELO CLIENTE

O arquivo *cliente.rs* é responsável por toda a lógica da aplicação no lado do cliente, incluindo o cadastro e autenticação de usuários, a troca de mensagens seguras e a comunicação com o servidor. Ele atua como a interface principal entre o usuário e o sistema, possibilitando interações seguras por meio de uma série de etapas cuidadosamente implementadas.

Ao iniciar a aplicação, o sistema pergunta se o usuário já possui uma conta. Caso não possua, é iniciado o processo de cadastro, no qual o usuário informa um nome de usuário e uma senha. Um par de chaves RSA (pública e privada) é gerado localmente, sendo que a chave privada é criptografada utilizando uma chave derivada da senha do usuário, garantindo sua confidencialidade. Ambas as chaves, bem como a senha (convertida em *hash* com *bcrypt*), são armazenadas no banco de dados de forma segura. Esse processo é mostrado na Figura 25.

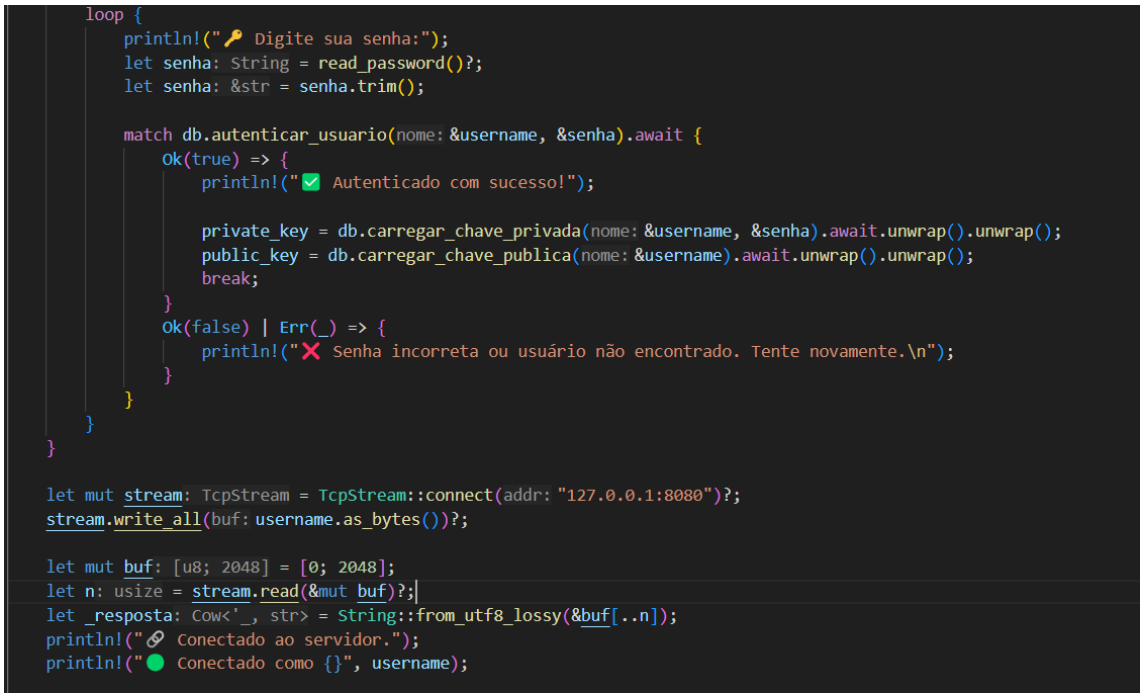
Figura 25 - Cadastro de usuários.

```rust
if !tem_conta {
    println!("🔑 Digite sua senha para cadastro:");
    let mut senha: String = String::new();
    io::stdin().read_line(buf: &mut senha)?;
    let senha: &str = senha.trim();

    let (priv_key: RsaPrivateKey, pub_key: RsaPublicKey) = gerar_par_de_chaves();

    let chave_privada_pem: Zeroizing<String> = priv_key.to_pkcs8_pem(rsa::pkcs8::LineEnding::LF).unwrap();
    let salt: [u8; 16] = rand::thread_rng().gen();
    let key: [u8; 16] = derivar_chave_aes(senha.as_bytes(), &salt);
    let chave_privada_encriptada: Vec<u8> = encrypt_aes(data: &chave_privada_pem, &key);
    let chave_codificada: String = format!("{}:{}", base64::encode(salt), base64::encode(chave_privada_encriptada));

    let chave_publica_pem: String = pub_key.to_pkcs1_pem(rsa::pkcs1::LineEnding::LF).unwrap();

    db.registrar_usuario(nome: &username, &senha, chave_privada: &chave_codificada, &chave_publica_pem) impl Future<Output = Res…
        .await Result<(), DatabaseError>
        .expect(msg: "❌ Falha ao registrar usuário");

    println!("✅ Usuário registrado com sucesso!");

    private_key = priv_key;
    public_key = pub_key;
```

Fonte: Autoria própria.

Se o usuário optar por realizar *login*, a senha fornecida é validada por meio da função *autenticar_usuario*, e a chave privada é carregada e descriptografada a partir do banco de dados. Após o *login* bem-sucedido, o cliente estabelece uma conexão com o servidor TCP (*Transmission Control Protocol*) e envia seu nome de usuário. Esse processo é mostrado na Figura 26.

Figura 26 - Autenticação e conexão com o servidor.



```
    loop {
        println!("🔑 Digite sua senha:");
        let senha: String = read_password()?;
        let senha: &str = senha.trim();

        match db.autenticar_usuario(nome: &username, &senha).await {
            Ok(true) => {
                println!("✅ Autenticado com sucesso!");

                private_key = db.carregar_chave_privada(nome: &username, &senha).await.unwrap().unwrap();
                public_key = db.carregar_chave_publica(nome: &username).await.unwrap().unwrap();
                break;
            }
            Ok(false) | Err(_) => {
                println!("❌ Senha incorreta ou usuário não encontrado. Tente novamente.\n");
            }
        }
    }
}

let mut stream: TcpStream = TcpStream::connect(addr: "127.0.0.1:8080")?;
stream.write_all(buf: username.as_bytes())?;

let mut buf: [u8; 2048] = [0; 2048];
let n: usize = stream.read(&mut buf)?;
let _resposta: Cow<'_, str> = String::from_utf8_lossy(&buf[..n]);
println!("🔗 Conectado ao servidor.");
println!("🟢 Conectado como {}", username);
```

Fonte: Autoria própria.

O menu apresentado, mostrado na Figura 27, permite que o usuário envie novas mensagens ou encerre a sessão. Antes de enviar uma nova mensagem, o sistema busca e exibe o histórico de conversas com o destinatário escolhido, utilizando a função *obter_historico_mensagens*. O texto digitado é criptografado utilizando AES-128 antes de ser transmitido pela rede, assegurando a proteção dos dados em trânsito. Além disso, a mensagem também é armazenada localmente no banco de dados para possibilitar consultas futuras.

Figura 27 - Menu de opções para o usuário.

Fonte: Autoria própria.

Por fim, caso o usuário opte por encerrar sua sessão, o cliente envia um comando de *logout* ao servidor e finaliza sua conexão TCP de forma adequada, garantindo que todos os recursos sejam liberados corretamente e evitando inconsistências no estado da aplicação.

4.9 IMPLEMENTAÇÃO RESPONSÁVEL PELO SERVIDOR

A função *start_server*, implementada no arquivo *servidor.rs*, é responsável por iniciar o servidor TCP da aplicação e gerenciar a comunicação entre os clientes conectados. O servidor opera de forma concorrente, utilizando múltiplas threads para garantir que diversas conexões possam ser tratadas simultaneamente, sem bloqueios ou interferências.

Ao ser iniciado, o servidor realiza o *bind* no endereço IP (*Internet Protocol*) 127.0.0.1 e porta 8080, utilizando o tipo *TcpListener*. Em seguida, entra em um loop contínuo onde aguarda conexões de entrada por meio do método *incoming*. Cada nova conexão recebida representa um novo cliente que deseja interagir com o sistema.

Para tratar cada cliente individualmente, o servidor cria uma nova tarefa (*thread)* com *thread::spawn*, isolando o tratamento das mensagens e das desconexões de cada usuário. O nome de usuário é recebido assim que o cliente se conecta e, após ser armazenado, o servidor responde com uma confirmação ("OK\n"), sinalizando que a conexão foi aceita com sucesso.

As conexões ativas são gerenciadas por um *HashMap<String, TcpStream>*, protegido por um *Arc<Mutex<...>>*, permitindo acesso seguro e compartilhado entre *threads*. Essa estrutura armazena os *streams* TCP dos usuários, permitindo que mensagens sejam encaminhadas diretamente de um cliente para outro.

Durante a execução, o servidor permanece escutando mensagens enviadas pelos clientes. Se a mensagem recebida começar com o prefixo MSG:, o servidor a interpreta como uma tentativa de envio de mensagem para outro usuário. A *string* é então dividida em partes (remetente, destinatário e conteúdo) e o conteúdo é encaminhado diretamente ao destinatário. Em caso de desconexão ou *logout*, o servidor remove o usuário do *HashMap* e finaliza sua *thread* associada.

A Figura 28 mostra o trecho de código correspondente à implementação da função *start_server*, destacando o processo de aceitação de conexões, gerenciamento de usuários e roteamento de mensagens entre clientes conectados.

Figura 28 - Função responsável por iniciar o servidor.

```rust
pub async fn start_server() {
    let listener: TcpListener = TcpListener::bind(addr: "127.0.0.1:8080").expect(msg: "Erro ao iniciar servidor");
    println!("🚀 Servidor rodando em 127.0.0.1:8080");
    let clients: Arc<Mutex<HashMap<String, TcpStream>>> = Arc::new(data: Mutex::new(HashMap::new()));
    for stream: Result<TcpStream, Error> in listener.incoming() {
        match stream {
            Ok(mut stream: TcpStream) => {
                let clients: Arc<Mutex<HashMap<String, …>>> = Arc::clone(self: &clients);
                thread::spawn(move || {
                    let mut buffer: [u8; 1024] = [0; 1024];
                    let _ = stream.read(buf: &mut buffer).unwrap();
                    let username: String = String::from_utf8_lossy(&buffer).trim().to_string();
                    println!("🟢 {} conectou", username);
                    let _ = stream.write_all(buf: b"OK\n");
                    clients.lock().unwrap().insert(k: username.clone(), v: stream.try_clone().unwrap());
                    loop {
                        let mut buf: [u8; 1024] = [0; 1024];
                        match stream.read(&mut buf) {
                            Ok(0) => {
                                println!("🔴 {} desconectou", username);
                                clients.lock().unwrap().remove(&username);
                                break;
                            }
                            Ok(n: usize) => {
                                let msg: String = String::from_utf8_lossy(&buf[..n]).to_string();
                                if msg.starts_with("LOGOUT") {
                                    println!("👋 {} saiu", username);
                                    clients.lock().unwrap().remove(&username);
                                    break;
                                }
                                if msg.starts_with("MSG:") {
                                    let partes: Vec<&str> = msg.trim().splitn(n: 3, pat: ':').collect();
                                    if partes.len() == 3 {
                                        let destinatario: &str = partes[1];
                                        let conteudo: &str = partes[2];
                                        println!("📩 {} para {}: {}", username, destinatario, conteudo);
                                        if let Some(dest: &mut TcpStream) = clients.lock().unwrap().get_mut(destinatario) {
                                            let _ = dest.write_all(buf: format!("MSG:{}:{}\n", username, conteudo).as_bytes());
                                        }
                                    }
                                }
                            }
                            Err(e: Error) => {
                                println!("❌ Erro no {}: {}", username, e);
                                clients.lock().unwrap().remove(&username);
                                break;
                            }
                        }
                    }
                });
            }
            Err(e: Error) => {
                println!("Erro de conexão: {}", e);
            }
        }
```

Fonte: Autoria própria.

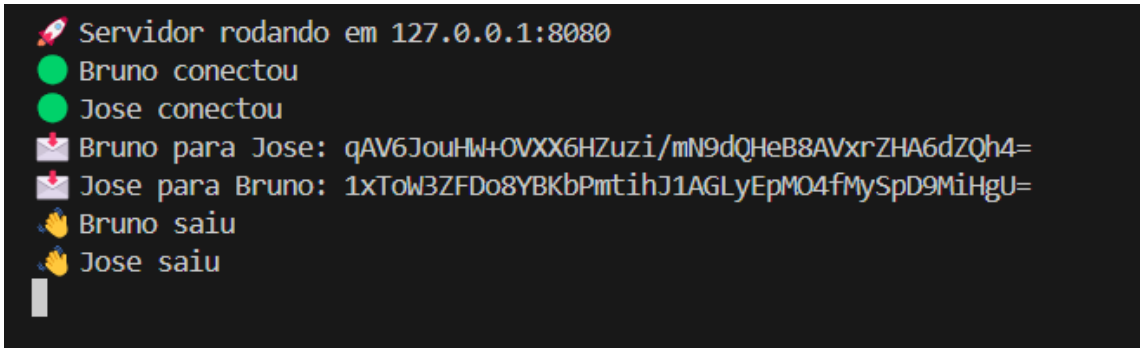
A Figura 29 mostra a execução prática do servidor durante uma sessão de comunicação entre dois usuários. Inicialmente, observa-se a mensagem "Servidor rodando em 127.0.0.1:8080", indicando que o servidor foi iniciado com sucesso e está escutando conexões na porta 8080. Em seguida, os usuários “Bruno” e “Jose” se conectam à aplicação, eventos que são registrados com os indicadores "Bruno conectou" e "Jose conectou", confirmando a recepção das conexões.

Após a conexão dos usuários, o servidor começa a encaminhar as mensagens trocadas entre eles. Cada mensagem é exibida no terminal com o remetente, o destinatário e o conteúdo cifrado em *base64*. Por exemplo, "Bruno para Jose" e "Jose para Bruno" indicam que ambos os usuários enviaram mensagens um ao outro com

sucesso, e que o servidor agiu corretamente como intermediário, apenas repassando os dados criptografados sem ter acesso ao conteúdo original.

Por fim, quando os usuários encerram suas sessões, o servidor registra suas desconexões com as mensagens "Bruno saiu" e "Jose saiu", encerrando corretamente suas conexões.

Figura 29 - Execução do servidor.



Fonte: Autoria própria.

4.10 TRATAMENTO DE ERROS

O arquivo *erro.rs* foi desenvolvido com o objetivo de centralizar e padronizar o tratamento de erros em todo o sistema, promovendo maior consistência, legibilidade e facilidade de manutenção. Utilizando a biblioteca *thiserror*, foi definido um *enum* chamado *DatabaseError*, que engloba diferentes tipos de falhas que podem ocorrer durante a execução da aplicação, incluindo erros de conexão com o banco de dados PostgreSQL (via *tokio_postgres* e *sqlx*), falhas em operações de criptografia (como *bcrypt* e rsa), problemas com decodificação de dados em Base64, além de erros genéricos que podem ser representados por mensagens de texto.

Cada variante do enum foi devidamente anotada com #[from], permitindo a conversão automática de erros provenientes de bibliotecas externas para o tipo *DatabaseError*. Isso simplifica significativamente o uso do *operador ?*, possibilitando a propagação de erros sem a necessidade de conversões manuais, resultando em um código mais limpo e expressivo. Além disso, foram atribuídas mensagens de erro descritivas a cada variante por meio da macro #[error("...")], o que facilita a rastreabilidade dos problemas e melhora a clareza dos logs do sistema.

Essa abordagem não apenas torna o tratamento de exceções mais consistente, mas também facilita a manutenção e evolução futura do projeto, o tratamento de erros é mostrado na Figura 30.

Figura 30 - Tratamento de erros.

```rust
10 implementations
pub enum DatabaseError {
    #[error("Erro no banco de dados postgres: {0}")]
    PostgresError(#[from] tokio_postgres::Error),

    #[error("Erro no SQLx: {0}")]
    SqlxError(#[from] sqlx::Error),

    #[error("Erro no bcrypt: {0}")]
    BcryptError(#[from] bcrypt::BcryptError),

    #[error("Erro no RSA: {0}")]
    RsaPkcs1Error(#[from] pkcs1::Error),

    #[error("Erro no RSA (geral): {0}")]
    RsaError(#[from] RsaError),

    #[error("Erro no RSA PKCS#8: {0}")]
    RsaPkcs8Error(#[from] rsa::pkcs8::Error),

    #[error("Erro na decodificação Base64: {0}")]
    Base64Error(#[from] base64::DecodeError),

    #[error("Erro desconhecido: {0}")]
    GenericError(String),
}
```

Fonte: Autoria própria.

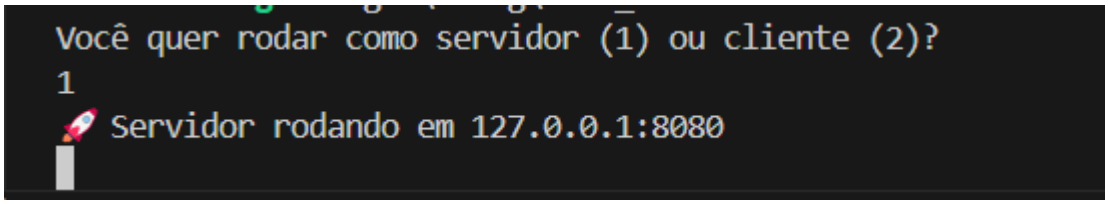
4.11 RESULTADOS ALCANÇADOS

A seguir, são apresentados os principais resultados obtidos com o desenvolvimento do projeto desse *chat* seguro utilizando *rust* com armazenamento de histórico criptografado.

4.11.1 INÍCIO DO SERVIDOR

O servidor é iniciado escutando conexões na porta 8080 da máquina local. Neste momento, ele aguarda que clientes se conectem e, ao receber conexões, registra os usuários e inicia o tratamento assíncrono para cada cliente, como mostrado na Figura 31.

Figura 31 - Inicialização do servidor aguardando conexões.

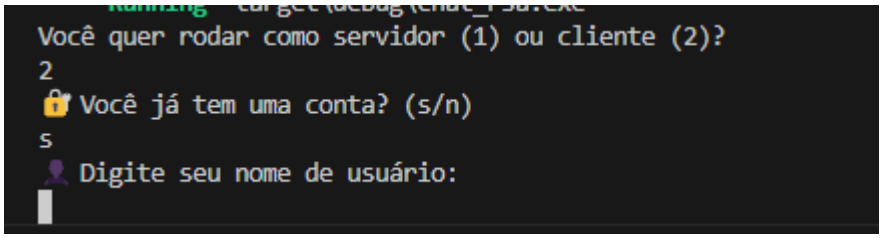


Fonte: Autoria própria.

4.11.2 CONEXÃO DO CLIENTE

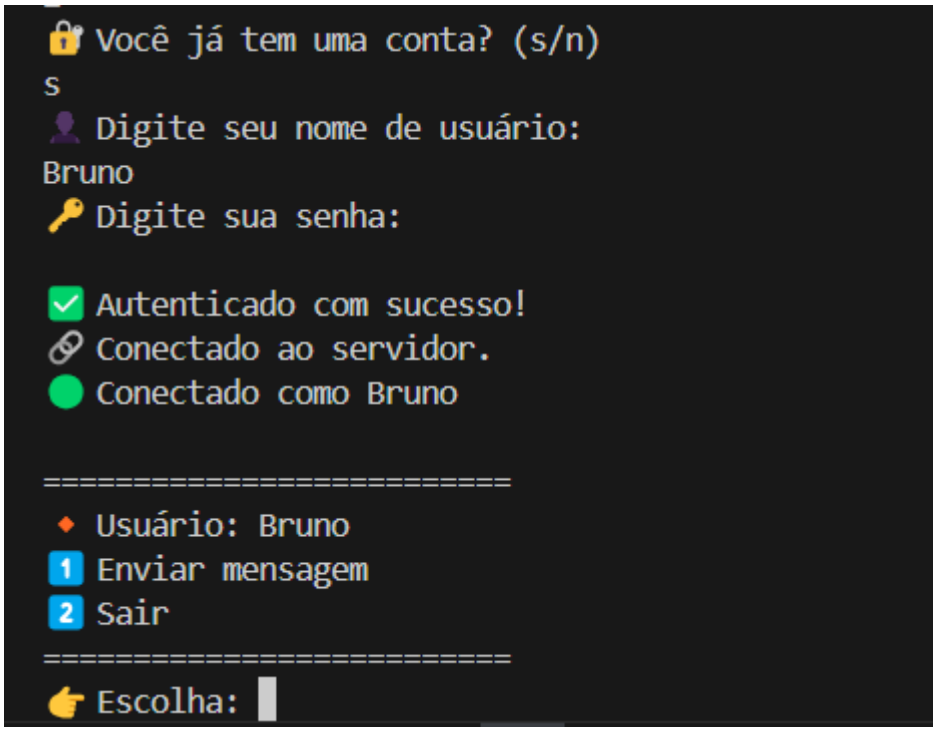
Ao executar o cliente, o usuário pode se autenticar ou se registrar garantindo controle de acesso via banco de dados e senha criptografada como mostrado na Figura 32. Após o *login*, o cliente se conecta ao servidor e entra no ambiente interativo, como mostrado na Figura 33.

Figura 32 - Autenticação do cliente.



Fonte: Autoria própria.

Figura 33 - Cliente autenticado.

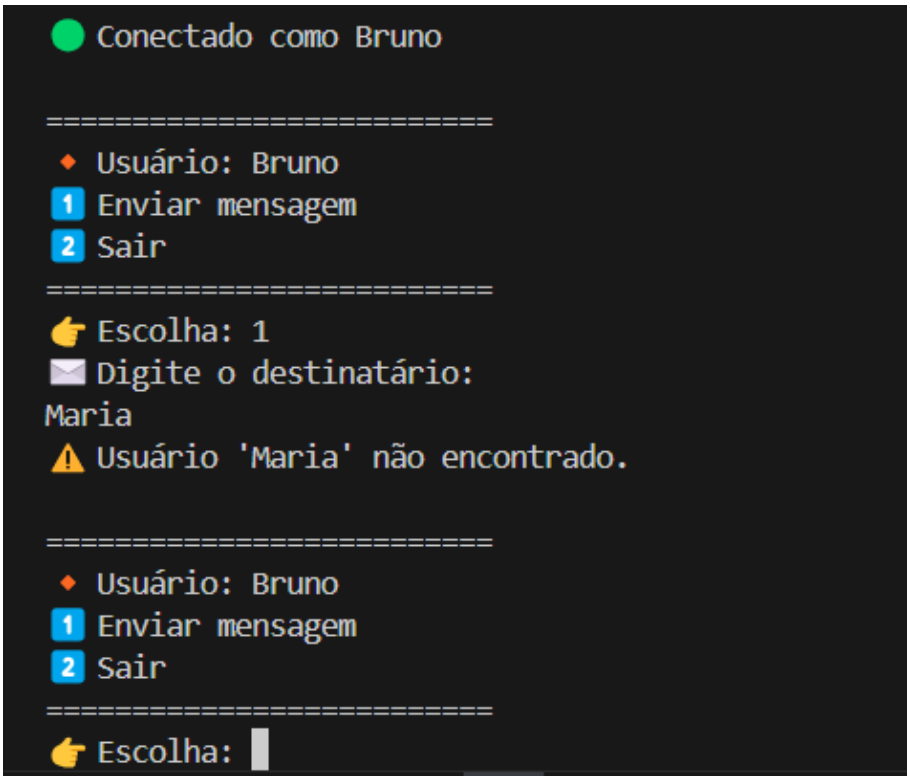


Fonte: Autoria própria.

4.11.3 ESCOLHA DO DESTINATÁRIO

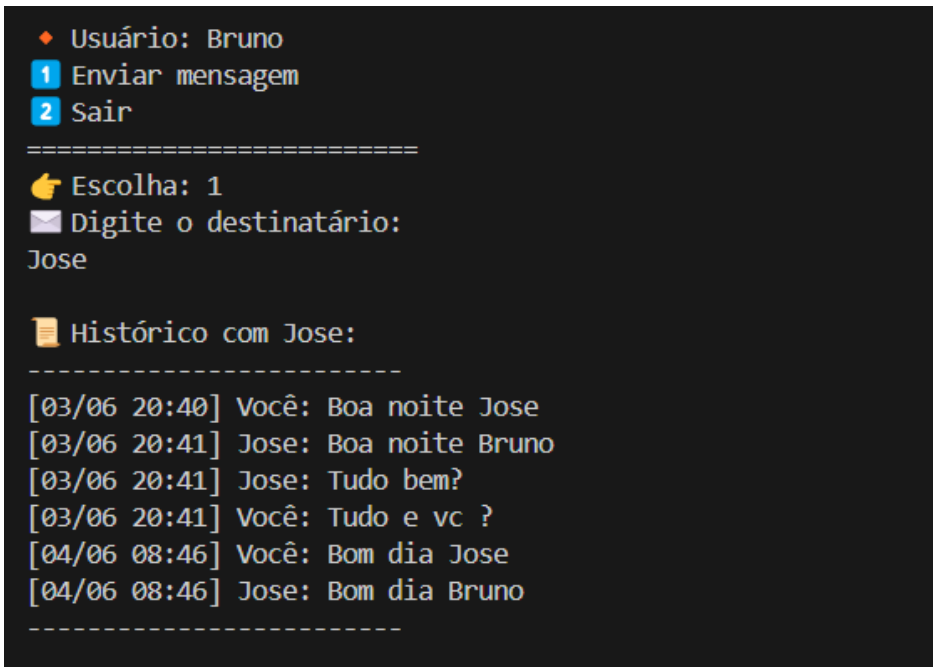
Ao escolher um destinatário, o sistema consulta o banco de dados e verifica se é um usuário cadastrado no sistema, caso não seja, o sistema retorna uma mensagem de erro como mostrado na Figura 34. No entanto se for um usuário cadastrado o sistema mostrará o histórico de conversas entre os dois usuários, essa ação é mostrada na Figura 35. As mensagens são descriptografadas e exibidas em ordem cronológica.

Figura 34 - Destinatário não encontrado.



Fonte: Autoria própria.

Figura 35 - Histórico de mensagens.

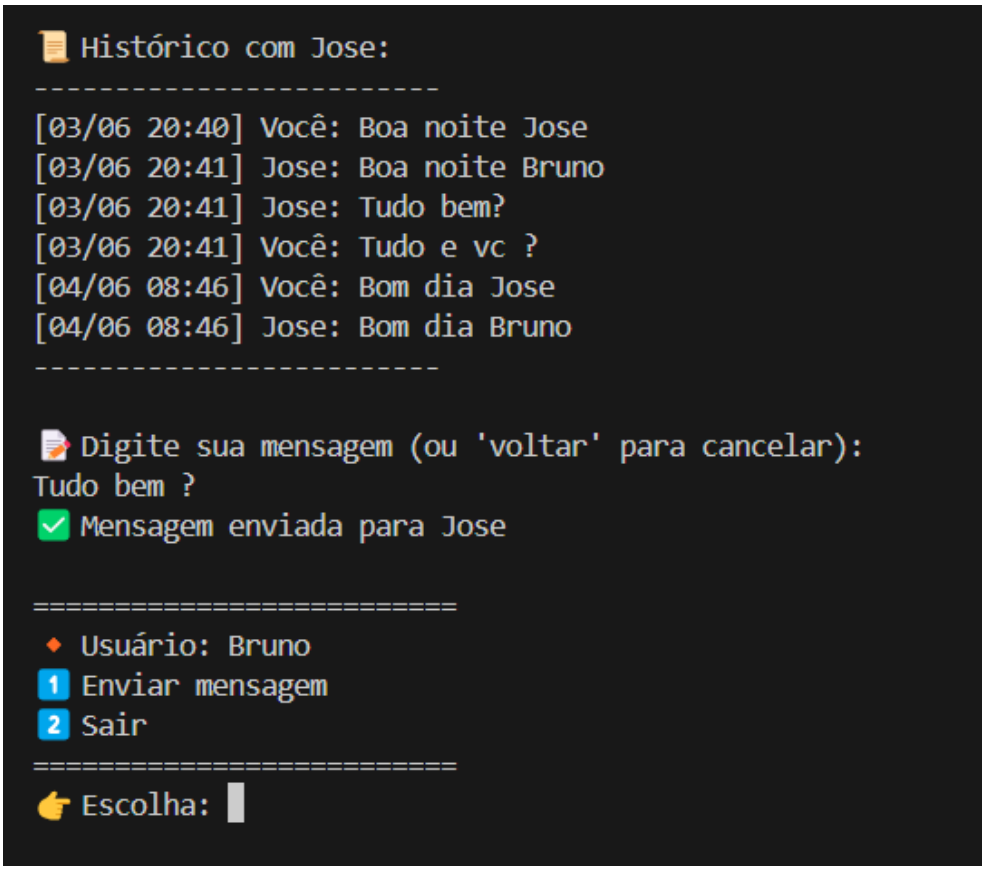


Fonte: Autoria própria.

4.11.4 ENVIO DE MENSAGENS

Abaixo do histórico de mensagens, o usuário digita a nova mensagem, que é então criptografada e enviada ao servidor. O servidor a encaminha ao destinatário. Esse processo é mostrado na Figura 36.

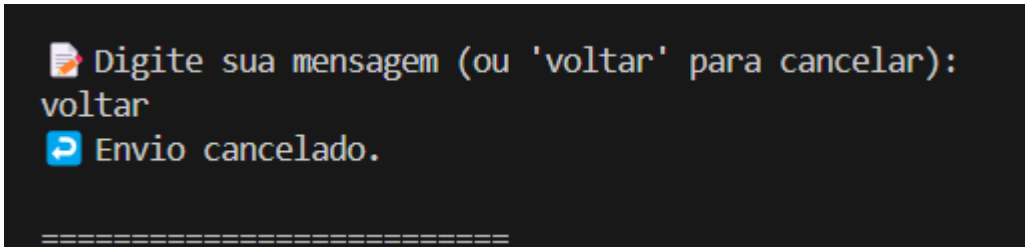
Figura 36 - Mensagem enviada com sucesso.



Fonte: Autoria própria.

O usuário tem a opção de digitar a mensagem “voltar” caso queira desistir de mandar a mensagem para aquele destinatário, como mostrado na Figura 37.

Figura 37 - Envio cancelado.

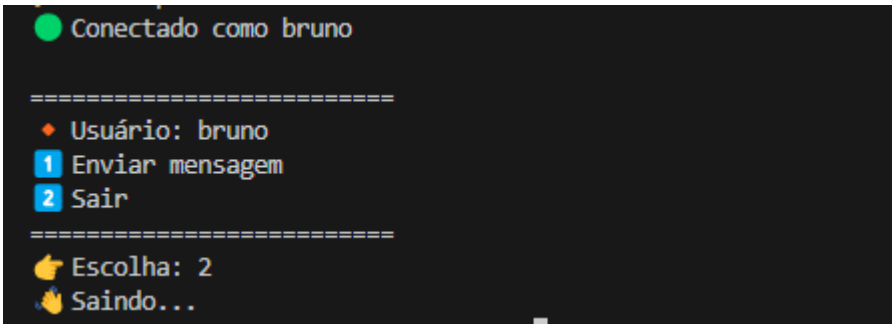


Fonte: Autoria própria.

4.11.5 ENCERRAMENTO DA SESSÃO

O usuário pode sair da aplicação a qualquer momento. A Figura 38 mostra que ao escolher a opção 2 no menu de opções o cliente envia um aviso de *logout* ao servidor, que encerra a conexão.

Figura 38 - Cliente encerrando a sessão.



Fonte: Autoria própria.

4.11.6 PERSISTÊNCIA DAS MENSAGENS NO BANCO DE DADOS

Para garantir a durabilidade, a confidencialidade e a integridade das comunicações entre os usuários, o sistema realiza o armazenamento persistente das mensagens e das chaves criptográficas em um banco de dados PostgreSQL. Esse processo envolve tanto o registro das mensagens trocadas quanto o gerenciamento seguro das chaves assimétricas (RSA) e simétricas (AES).

A tabela mensagens é o ponto central da persistência das comunicações. Cada mensagem, antes de ser armazenada, é criptografada com o algoritmo AES-128 em modo CBC utilizando uma chave de sessão exclusiva para cada par de usuários. Em seguida, o resultado é codificado em base64, impedindo que o conteúdo seja compreendido mesmo em caso de vazamento do banco.

A estrutura da tabela mensagens inclui:

- *id*: identificador único da mensagem (chave primária);
- *remetente*: nome do usuário que enviou a mensagem;
- *destinatario*: nome do usuário que recebeu a mensagem;
- mensagem: conteúdo da mensagem criptografado e codificado em base64;
- *data_envio*: data e hora do envio da mensagem.

Essa estrutura é mostrada na Figura 39, que apresenta um exemplo real de armazenamento das mensagens com os campos devidamente preenchidos.

Figura 39 - Tabela mensagens.

	id [PK] integer	remetente character varying (255)	destinatario character varying (255)	mensagem text	data_envio timestamp with time zone
1	14	Bruno	Jose	oBPneqinmXTAee4/zh9N3vful7wyz/NXd04Y/qzdggQ=	2025-06-03 20:40:53.609081-03
2	15	Jose	Bruno	yzbkK81p3wzf2JFMALpPiQVZ9KzcrgvChE7BLIXQWp...	2025-06-03 20:41:06.80975-03
3	16	Jose	Bruno	PlqyG75ZqrZGhU29UVb5m6crf8LBHfvAH/Ct1AasBJ...	2025-06-03 20:41:17.025788-03
4	17	Bruno	Jose	mIE/X0lueuyqzsREqavhXsAmnQl6GBrQZBlRtRJVDlk=	2025-06-03 20:41:27.490896-03
5	18	Bruno	Jose	qAV6JouHW+OVXX6HZuzi/mN9dQHeB8AVxrZHA6d...	2025-06-04 08:46:47.549365-03
6	19	Jose	Bruno	1xToW3ZFDo8YBKbPmtihJ1AGLyEpMO4fMySpD9Mi...	2025-06-04 08:46:59.806596-03
7	20	Bruno	Jose	OFx7+/ISWtCCKPu2cbMHi45sffBaeUXSTlUnQBSSCtk=	2025-06-04 10:02:04.549362-03

Fonte: Autoria própria.

Durante o registro, cada usuário tem um par de chaves RSA gerado automaticamente. A chave pública é armazenada em texto claro no banco, pois é utilizada por outros usuários para criptografar a chave de sessão AES. Já a chave privada é criptografada com AES-128 antes de ser armazenada, utilizando uma chave derivada da senha do próprio usuário. Essa estrutura é mostrada na Figura 40. Isso assegura que, mesmo se o banco de dados for acessado de forma indevida, as chaves privadas não possam ser utilizadas sem o conhecimento da senha original do usuário.

Figura 40 - Tabela usuários.

	id [PK] inte	nome character	senha_hash character varying (255)	chave_publica text	chave_privada text
1	17	Bruno	$2b$12$s9TB90idGKUucKWWxb6Nw.zfdv1cu19MjhcXMg51v/.Aacmrle...	-----BEGIN RSA PUBLIC KEY-----	x4w/JAvU2di5gWQd2ATJSg==:dLakvBANdpsncBkgKzmgc63tY/tCNwc6YJalxBmbHWmaSnqbc
2	18	Jose	$2b$12$chlufo5U7iFqZ/qRG9w0ReW/dJkeEaW7ZOAlPyK1YZxqYCB.ifBk.	-----BEGIN RSA PUBLIC KEY-----	atlJVw4p4uOoJroy6gK/FQ==:RPFkh+PxF7C/Y9LITA1OhluLBmsBT8EJp9HoNGwJCB43JeUyUq
3	19	Teste	$2b$12$G/5hSdDQ.56mh70cAjw5m.qKDgQHSB9i/9uuxu5igOTxZp9p/I.h6	-----BEGIN RSA PUBLIC KEY-----	HcxYBnrY5blcxlqker0jfw==:qszcK7him/n0/xC5lHBfdBlale9pSp56bw/z/GFHEXZvLQNkLQ4wubl

Fonte: Autoria própria.

Para suportar a troca segura de chaves simétricas entre pares de usuários, foi criada a tabela *chaves_sessoes* mostrada na Figura 41. Ela armazena a chave de sessão AES cifrada separadamente para o remetente e o destinatário, utilizando suas respectivas chaves públicas RSA.

A tabela *chaves_sessoes* contém os seguintes campos:

- *id*: identificador da sessão (chave primária);
- *remetente:* nome do usuário que iniciou a sessão;
- *destinatario:* nome do outro usuário participante da sessão;
- *chave_para_remetente*: chave AES cifrada com a chave pública do remetente;

- *chave_para_destinatario*: chave AES cifrada com a chave pública do destinatário.

Figura 41 - Tabela chaves_sessoes.

	id [PK] integer	remetente character varying (255)	destinatario character varying (255)	chave_para_remetente bytea	chave_para_destinatario bytea
1	13	Jose	Bruno	[binary data]	[binary data]
2	14	Bruno	Teste	[binary data]	[binary data]

Fonte: Autoria própria.

Essa estrutura garante que apenas os participantes da conversa possam descriptografar e utilizar a chave AES compartilhada, mantendo a confidencialidade das mensagens mesmo em canais abertos.

CAPÍTULO V - CONSIDERAÇÕES FINAIS

Neste capítulo são apresentadas as considerações finais deste Trabalho de Conclusão de Curso, bem como as dificuldades encontradas no mesmo e sugestões para trabalhos futuros.

5.1 CONSIDERAÇÕES FINAIS

Este Trabalho teve como objetivo estudar conceitos e práticas de segurança na comunicação digital, com foco na implementação de um sistema de *chat* seguro, utilizando técnicas de autenticação e criptografia, desenvolvido na linguagem de programação *Rust*.

Inicialmente, foram abordados os fundamentos da criptografia, funções *hash* e autenticação de usuários, essenciais para garantir a segurança na troca de informações. Discorreu-se sobre conceitos como criptografia simétrica e assimétrica, resumo de senha com funções *hash* (*bcrypt*) e os mecanismos de autenticação, além da estrutura de comunicação entre cliente e servidor.

Foram descritos de forma detalhada os algoritmos utilizados, como AES para criptografia das mensagens, RSA para a troca segura da chave simétrica e *bcrypt* para a geração do *hash* das senhas, garantindo a confidencialidade e a segurança no armazenamento e transmissão dos dados.

Foram desenvolvidos os principais componentes de um *chat* seguro: o servidor responsável pelo gerenciamento das conexões e autenticação dos usuários; o armazenamento persistente das mensagens no banco de dados; além da criptografia aplicada tanto na comunicação quanto nas informações sensíveis dos usuários.

5.2 DIFICULDADES ENCONTRADAS

A principal dificuldade enfrentada durante o desenvolvimento foi o uso da linguagem *Rust* que é uma linguagem mais recente, sua sintaxe, modelo de propriedade e gerenciamento de memória exigiram tempo e dedicação para serem compreendidos e aplicados corretamente. Além disso, a escassez de materiais didáticos e exemplos práticos voltados para aplicações completas também representou um desafio adicional.

5.3 TRABALHOS FUTUROS

Como continuidade deste trabalho, algumas melhorias e extensões podem ser implementadas:

- Interface gráfica: desenvolver uma interface visual amigável para facilitar a usabilidade do sistema;
- Envio de arquivos: adicionar suporte para troca de arquivos entre os usuários, mantendo a criptografia;
- Autenticação em múltiplas etapas (MFA): elevar a segurança na autenticação utilizando verificação via e-mail ou aplicativo autenticador.

REFERÊNCIAS

BARBOSA, Luis Alberto de Moraes et al. *RSA: Criptografia Assimétrica e Assinatura Digital*. Campinas: Universidade Estadual de Campinas, 2003.

KLABNIK, Steve; **NICHOLS**, Carol. *The Rust Programming Language*, 2025. Disponível em: https://doc.rust-lang.org/book/. Acesso em: 3 jul. 2025.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST). *Secure Hash Standard (SHS)*. FIPS PUB 180-4, 2015. Disponível em: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

NIST; **DWORKIN**, M.; **SONMEZ TURAN**, M.; **MOUHA**, N. *Advanced Encryption Standard (AES)*. Federal Inf. Process. Stds. (NIST FIPS), Instituto Nacional de Padrões e Tecnologia, Gaithersburg, MD, 2023. Disponível em: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=936594. Acesso em: 19 out. 2024.

SCHNEIER, Bruce. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 2015.

STALLINGS, William. *Criptografia e Segurança de Redes: Princípios e Práticas*. 6ª ed. São Paulo: Pearson, 2015.

STALLINGS, William. *Segurança de Computadores: Princípios e Práticas*. 2ª ed. São Paulo: Pearson, 2014.