Programação Concorrente

Criptografia e Segurança

João Guilherme

14/01/2025

1. Introdução

A criptografia tem um papel crucial na segurança da comunicação, especialmente em áreas como a militar, onde a confidencialidade e a integridade das informações são essenciais para o sucesso das operações. Este trabalho visa explorar a aplicação prática da criptografia na comunicação militar, combinando a cifra de Vigenère com técnicas de programação concorrente para simular um sistema de transmissão de mensagens seguro e eficiente. A cifra de Vigenère, um método de criptografia que utiliza uma palavra-chave para codificar mensagens, será implementada em um sistema concorrente que simula as diferentes etapas do processo de comunicação, desde a geração da mensagem até sua decodificação e execução. O objetivo é demonstrar como a criptografia e a programação concorrente podem ser combinadas para criar um sistema de comunicação robusto e seguro, que atenda às demandas de ambientes complexos e desafiadores, como o militar.

2. Formalização do Problema

O objetivo principal deste trabalho consiste na modelagem e elaboração de uma estrutura de processamento criptográfico concorrente de mensagens de um exército. Para esse fim, foi utilizado a cifra de Vigenère, um método de criptografia, aliado aos mecanismos de sincronização estudados em aula – Locks e Variáveis de condições – inspirados pelos problemas clássicos de sincronização como o problema dos produtores e consumidores e o problema dos canibais.

Este modelo simula uma cadeia de comando militar onde os generais criam ordens, os tenentes as codificam e o pombos as transportam para os cabos, que decodificam e executam a mensagem.

Já a cifra de Vigenère emprega uma palavra-chave para determinar diferentes deslocamentos. Cada letra da palavra-chave corresponde a um deslocamento específico no alfabeto, e a mensagem é cifrada letra por letra, aplicando o deslocamento indicado pela letra correspondente na palavra-chave. Esse processo se repete ao longo da mensagem, utilizando a palavra-chave de forma cíclica. A repetição da palavra-chave e a variação dos deslocamentos tornam a cifra de Vigenère mais segura que a cifra de César simples, dificultando a análise de frequência e a quebra do código por métodos convencionais.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W

Figura 1 - Tabula reta usada na codificação da cifra de Vigenère

Usualmente é utilizado uma matriz chamada tábula reta para codificação das mensagens. Entretanto, como é possível perceber na figura 1, os deslocamentos podem ser descritos na aritmética modular como:

$$M + C \equiv R \; MOD \; 26$$

Em que R representa caractere resultante, M o caractere mensagem e C o caractere chave. De maneira semelhante, é possível decriptar a mensagem calculando a função inversa em respeito a M:

$$M \equiv (R - C) \; MOD \; 26$$

Esta implementação **autoral** simplifica o problema, não sendo necessário o armazenamento de uma matriz razoavelmente grande, como a tabula reta.

3. Descrição do Algoritmo

Como dito anteriormente, o código simula um sistema de comunicação militar com quatro entidades: general, tenente, pombo e cabo, cada uma representada por uma função executada por uma *thread* separada. A comunicação entre elas se dá através de buffers compartilhados ("mesas" e "mochilas") protegidos por locks e sincronizados por variáveis de condição.

O programa inicia criando quatro *arrays* de *threads* representando as entidades General, Tenente, Pombo e Cabo, respectivamente, e sua quantidade de instâncias.

```
int main(){
    pthread_t gen[G], cab[C], ten[T], pru[P];
    int* ptr_gen;
    for (int i = 0; i < G; i++){
        ptr_gen = (int*) malloc(sizeof(int));
        *ptr_gen = i;
        pthread_create(&gen[i], NULL, general, ptr_gen);
    }

    for (int i = 0; i < T; i++){
        pthread_create(&ten[i], NULL, tenente, NULL);
    }

    for (int i = 0; i < P; i++){
        pthread_create(&pru[i], NULL, pombo, NULL);
    }

    for (int i = 0; i < C; i++){
        pthread_create(&cab[i], NULL, cabo, NULL);
    }

    pthread_join(gen[0], NULL);
    return 0;
}
```

A thread **general** é responsável por gerar ordens militares aleatórias. Para garantir que não haja conflito de acesso à mesa, um lock_mensagem é usado. A variável de condição cond_mensagem sinaliza quando a mesa está disponível para escrita ou leitura. Note que a estrutura do processo preserva a ordenação FIFO - *first in first out*. - das ordens, ou seja, as primeiras ordens são as primeiras a serem processadas.

```
void* general(void* meuid){
    int ptr_mod;
    while(1){
        char* ordem = pensa_ordem();
        sleep(3);
        pthread_mutex_lock(&lock_mensagem);
            while (ptr_general % QTO == ptr_tenente % QTO){
                pthread_cond_wait(&cond_mensagem, &lock_mensagem);
            }
            ptr_mod = ptr_general % QTO;
            mesa_general[ptr_mod] = ordem;
            printf("Ordem: \"%s\". Inserida na posicao %d da mesa general\n", mesa_general[ptr_mod],
ptr_mod);
            ptr_general++;
            if ((ptr_general - 2) % QTO == ptr_tenente % QTO){
                pthread_cond_signal(&cond_mensagem);
            }
        pthread_mutex_unlock(&lock_mensagem);
    }
    pthread_exit(0);
}
```

A função pensa_ordem é a mente criativa por trás das ordens militares geradas no código. Para cada general, ela formula uma ordem aleatória, combinando uma especificação com uma localização criando um conjunto dinâmico de instruções para os tenentes decifrarem e os cabos executarem

Já a thread **tenente** atua como um intermediário entre o general e o pombo, responsável pelo transporte. Sua principal função é **codificar as mensagens** utilizando locks e variáveis de condição para sincronizar suas ações. O lock_mensagem, em conjunto com a variável de condição cond_mensagem, controla o acesso à mesa do general, isso previne leituras de mensagens inexistentes e garante que as ordens sejam processadas na ordem correta. De maneira análoga, o lock_pombo e a variável de condição cond_tenente regulam o acesso à mochila de envio. O tenente só coloca uma mensagem codificada na mochila se houver espaço disponível.

```
void* tenente(void* meuid){
    char* mensagem;      char* cripta;
    while(1){
        sleep(3);
        pthread_mutex_lock(&lock_mensagem);
            while ((ptr_tenente + 1) % QTO == (ptr_general % QTO) ){
                pthread_cond_wait(&cond_mensagem, &lock_mensagem);
            }
            ptr_tenente++;
            int ptr_mod = ptr_tenente %  QTO;
            mensagem = mesa_general[ptr_mod];
            printf("Ordem: \"%s\" retirada da posicao %d da mesa general\n", mensagem, ptr_mod);
            if ((ptr_general + 1) % QTO == ptr_mod){
                pthread_cond_broadcast(&cond_mensagem);
            }
        pthread_mutex_unlock(&lock_mensagem);
        cripta = codifica(mensagem, chave);
```

```
        free(mensagem);
        pthread_mutex_lock(&lock_pombo);
            while(ptr_mochila >= CAPOMBO){
                pthread_cond_wait(&cond_tenente, &lock_pombo);
            }
            mochila_envio[ptr_mochila] = cripta;
            printf("Cripta \"%s\" colocada na posicao %d da mochila\n", mochila_envio[ptr_mochila],
ptr_mochila);
            ptr_mochila++;
            if (ptr_mochila == CAPOMBO){
                pthread_cond_signal(&cond_pombo);
            }
        pthread_mutex_unlock(&lock_pombo);
             }
    pthread_exit(0);
}
```

A thread **pombo** transfere as mensagens codificadas da mochila de envio para a mochila de destino quando atinge sua capacidade e, em seguida, as coloca na mesa do cabo. A sincronização é feita com lock_pombo e cond_pombo para acesso à mochila, e lock_cabo e cond_cabo para acesso à mesa do cabo. Note que é preciso copiar os valor entre as mochilas para se evitar **starvation** da mochila de envio na entrega das cartas. Há ainda a inversão da ordem de mensagens para manter o comportamento FIFO das mensagens.

```
void* pombo(void* meuid){
    while(1){
        pthread_mutex_lock(&lock_pombo);
            while (ptr_mochila != CAPOMBO){
                pthread_cond_wait(&cond_pombo, &lock_pombo);
            }
            for (int i = 0; i < CAPOMBO; i++){
                mochila_destino[i] = mochila_envio[CAPOMBO - i - 1];
            }
            ptr_mochila = 0;
            pthread_cond_broadcast(&cond_tenente);
        pthread_mutex_unlock(&lock_pombo);

        printf("Tranferindo Mensagens...\n");
        sleep(1);

        pthread_mutex_lock(&lock_cabo);
            while (ptr_cabo > QTB - CAPOMBO){
                pthread_cond_wait(&cond_cabo, &lock_cabo);
            }
            for (int i = 0; i < CAPOMBO; i++){
                mesa_cabo[ptr_cabo + i] = mochila_destino[i];
                printf("Cripta: \"%s\" transferida na posicao %d para mesa cabo\n", mesa_cabo[ptr_cabo +
i], ptr_cabo + i);
            }
            ptr_cabo += CAPOMBO;

            if (ptr_cabo ==  CAPOMBO){
                pthread_cond_signal(&cond_cabo);
            }
        pthread_mutex_unlock(&lock_cabo);
    }
}
```

A thread **Cabo** representa o receptor final da mensagem na cadeia de comunicação. Sua função principal é decodificar as mensagens recebidas e, em seguida, executar a ordem. Para garantir que a comunicação flua sem problemas, a thread cabo utiliza o lock_cabo e a variável de condição cond_cabo para sincronizar o acesso à mesa_cabo. A thread verifica se

há mensagens na mesa e, caso não haja, ela espera até ser sinalizada pelo pombo. Após retirar uma mensagem da mesa, o cabo a decodifica usando a mesma chave secreta que o tenente utilizou para codificá-la. Essa sincronização garante que o cabo não tente ler mensagens de uma mesa vazia e que as mensagens sejam processadas na ordem correta, completando o ciclo de comunicação.

```
void* cabo(void* meuid){
    char* cripta;
    char* mensagem;
    while(1){
        pthread_mutex_lock(&lock_cabo);
                while (ptr_cabo == 0){
                    pthread_cond_wait(&cond_cabo, &lock_cabo);
                }
                ptr_cabo--;
                cripta = mesa_cabo[ptr_cabo];
                printf("Cripta: \"%s\" retirada da posicao %d da mesa cabo\n", cripta, ptr_cabo);
                if (ptr_cabo == QTO - 1){
                    pthread_cond_signal(&cond_cabo);
                }
        pthread_mutex_unlock(&lock_cabo);
        mensagem = decodifica(cripta, chave);
        free(cripta);
        printf("A Mensagem eh: \"%s\"\n", mensagem);
        free(mensagem);
    }

}
```

Já a função auxiliar codifica recebe como parâmetros a mensagem a ser criptografada e a chave secreta. Ela itera sobre cada caractere da mensagem, aplicando um deslocamento específico com base no caractere correspondente na chave. O deslocamento é calculado utilizando aritmética modular, garantindo que o resultado permaneça dentro do intervalo de caracteres válidos.

```
char* codifica(char* mensagem, char* chave){
    int tmn_chave = strlen(chave);
    int tmn_mensagem = strlen(mensagem);
    char* cifra = (char*) malloc(tmn_mensagem + 1);
    int c_mensagem, c_chave, c_cifra;

    for (int i = 0; i < tmn_mensagem; i++){
        c_mensagem = mensagem[i];
        c_chave = chave[i % tmn_chave];

        c_cifra = modulo( (c_chave - RG_INIT ) + (c_mensagem - RG_INIT), RANGE) + RG_INIT;
        cifra[i] = c_cifra;
    }
    cifra[tmn_mensagem] = '\0';
    return cifra;
}
```

A função decodifica realiza o processo inverso da função codifica, recuperando a mensagem original a partir da cifra e da chave secreta. Ela também itera sobre cada caractere da cifra, aplicando o deslocamento inverso correspondente na chave.

```
char* decodifica(char* cifra, char* chave){
    int tmn_chave = strlen(chave);
    int tmn_cifra = strlen(cifra);
```

```
    char* mensagem = (char*) malloc(tmn_cifra + 1);
    int c_cifra, c_chave, c_mensagem;

    for (int i = 0; i < tmn_cifra; i++){
        c_cifra = cifra[i];
        c_chave = chave[i % tmn_chave];

        c_mensagem = modulo(c_cifra - c_chave, RANGE) + RG_INIT;
        mensagem[i] = c_mensagem;
    }
    mensagem[tmn_cifra] = '\0';
    return mensagem;
}
```

O funcionamento, assim como mais explicações sobre o código está disponível no seguinte vídeo. O código na integra foi disponibilizado na mesma plataforma que este documento.

4. Conclusão

O desenvolvimento deste trabalho permitiu explorar e integrar conceitos de criptografia e programação concorrente, culminando em uma simulação de um sistema de comunicação militar seguro. A utilização da cifra de Vigenère e a implementação de threads para simular as diferentes entidades do processo de comunicação demonstraram a viabilidade da aplicação prática da criptografia em cenários reais. As ferramentas de sincronização, como locks e variáveis de condição, foram essenciais para garantir a comunicação eficiente e ordenada entre as threads, assegurando a integridade das mensagens transmitidas.

Este trabalho, portanto, comprova a importância da criptografia e da programação concorrente na construção de sistemas de comunicação robustos e seguros, capazes de proteger informações sensíveis em ambientes complexos, como o militar. Adicionalmente, o projeto proporcionou um aprendizado aprofundado sobre a aplicação prática de conceitos teóricos, abrindo caminho para futuras explorações e desenvolvimento de soluções mais complexas e eficientes no âmbito da segurança da informação.

Os resultados obtidos na simulação, demonstrados no vídeo, evidenciaram o sucesso da implementação, com a comunicação entre as entidades ocorrendo de forma segura e eficiente, comprovando a robustez do sistema desenvolvido.