O Algoritmo do rsync – Fundamentos, História e Aplicações

Introdução

Na área de Ciência da Computação, a transferência eficiente e sincronização de arquivos entre máquinas tem sido uma preocupação fundamental, especialmente com o crescimento da internet e dos sistemas distribuídos. O algoritmo **rsync**, criado por Andrew Tridgell em 1996, revolucionou esse processo ao permitir que somente as diferenças entre arquivos fossem transmitidas, reduzindo drasticamente a quantidade de dados transferidos.

---

1. História e Motivação

Antes do surgimento do rsync, os métodos mais comuns para sincronizar arquivos entre computadores envolviam a transferência completa de arquivos, mesmo que apenas uma pequena parte tivesse sido modificada. Protocolos como FTP (File Transfer Protocol) e SCP (Secure Copy Protocol) copiavam arquivos inteiros, sem considerar a redundância ou a possibilidade de transferir apenas as partes alteradas. Isso causava desperdício de largura de banda e aumento do tempo de sincronização.

Andrew Tridgell, ao trabalhar com redes e sistemas Unix, percebeu essa ineficiência e criou o **rsync** como parte de sua tese de doutorado na Australian National University (Tridgell, 1999). O algoritmo implementado em rsync usa uma técnica chamada **delta encoding** para identificar e transferir apenas as diferenças entre os arquivos, tornando a sincronização muito mais eficiente.

---

2. Funcionamento Básico do Algoritmo

O algoritmo rsync baseia-se em duas ideias principais:

- **Checksum de Blocos**: O arquivo destino é dividido em blocos fixos, e para cada bloco é calculado um checksum rápido (rolling checksum, baseado em Adler-32) e um checksum forte (MD4 ou MD5).
- **Busca de Blocos Correspondentes**: O arquivo fonte (local) é então analisado para encontrar quais blocos já existem no destino, usando os checksums para identificar correspondências.

O algoritmo envia apenas os blocos novos ou modificados, ao invés de enviar o arquivo inteiro.

Etapas principais (Tridgell, 1999):

1. O servidor destino divide seu arquivo em blocos e calcula checksums.
2. O cliente (fonte) calcula checksums similares sobre seu arquivo e identifica blocos coincidentes.
3. O cliente envia somente os dados dos blocos novos ou alterados.
4. O servidor reconstrói o arquivo usando os blocos existentes e os blocos novos.

Esse método reduz a transferência de dados significativamente, especialmente em arquivos grandes com pequenas modificações.

3. Por que surgiu o rsync?

A motivação de Tridgell para desenvolver rsync foi a necessidade de sincronizar backups e arquivos em redes lentas ou de alta latência, onde transferir arquivos inteiros era inviável.

Antes do rsync, soluções como **RCP (Remote Copy Protocol)** ou mesmo o uso do **tar** com pipes para compressão eram comuns, mas insuficientes em redes com restrições severas.

---

4. Vantagens do rsync

- **Eficiência na Transferência:** Transferência apenas das diferenças entre os arquivos, economizando largura de banda.
- **Versatilidade:** Pode ser usado localmente ou remotamente via SSH, suporta várias opções, como compressão, exclusão seletiva e preservação de permissões.
- **Robustez:** Capaz de lidar com grandes conjuntos de dados, inclusive sistemas de arquivos inteiros.
- **Incrementalidade:** Perfeito para backups incrementais e sincronização contínua.

Segundo Andrew Tridgell e Paul Mackerras (1997), o algoritmo do rsync é "uma solução prática e elegante para um problema computacionalmente desafiador, permitindo sincronização rápida e eficiente em ambientes reais".

---

5. Desvantagens e Limitações

- **Consumo de CPU:** A geração e comparação de checksums podem ser computacionalmente custosas em arquivos muito grandes.
- **Sincronização em Arquivos Binários Difíceis:** Em arquivos fortemente modificados ou corrompidos, pode haver maior custo de processamento.
- **Dependência da Conectividade:** Para sincronizações remotas, requer uma conexão de rede estável e segura.
- **Não Resolve Conflitos de Edição:** Rsync não gerencia edição concorrente de arquivos, apenas sincroniza cópias.

---

6. Conclusão

O algoritmo rsync representa um marco importante na ciência da computação aplicada à sincronização de dados. Sua criação atendeu a uma necessidade real, utilizando técnicas de checksum rolling e delta encoding para minimizar a transferência de dados. A eficiência e praticidade do rsync o tornaram um padrão de fato em sistemas Unix e Linux, amplamente usado para backups, deploys e sincronização em geral.

Conforme destacado por Tridgell (1999), rsync não apenas economiza tempo e recursos, mas também habilita a manutenção e a distribuição de dados em escala global de maneira eficiente.

Claro! Vamos aprofundar a explicação sobre **onde e como o algoritmo rsync é usado**, quais áreas de aplicação ele impacta, e exemplos práticos de seu uso.

Aplicações e Usos do Algoritmo rsync: Áreas e Casos Práticos

---

1. Contexto Geral do Uso do rsync

O algoritmo rsync, com sua capacidade de transferir apenas as diferenças entre arquivos, é amplamente utilizado em sistemas que demandam sincronização eficiente e transferência econômica de dados. A redução da largura de banda e do tempo de cópia faz dele uma ferramenta essencial para várias áreas que lidam com armazenamento e comunicação de dados.

---

2. Áreas que Utilizam o Algoritmo rsync

2.1 Administração de Sistemas e Backup

- **Sincronização de arquivos entre servidores:** Administradores usam rsync para manter cópias atualizadas de dados entre servidores locais e remotos, garantindo que as alterações sejam rapidamente propagadas.
- **Backups incrementais:** Em vez de fazer backup completo a cada vez, o rsync copia somente o que mudou, reduzindo o tempo e o espaço consumidos.
- **Restaurar sistemas após falhas:** Permite rápida recuperação ao sincronizar arquivos críticos a partir de backups remotos.

Exemplo prático:
Um servidor de arquivos corporativo faz backup noturno dos dados para uma máquina remota. O rsync analisa quais arquivos ou blocos foram alterados e transfere apenas essas mudanças, economizando horas de transmissão e reduzindo o impacto na rede.

2.2 Desenvolvimento de Software e DevOps

- **Deploy de aplicações:** Ferramentas de automação de deploy usam rsync para enviar apenas os arquivos alterados para servidores de produção, acelerando o processo e diminuindo downtime.
- **Sincronização de ambientes de desenvolvimento:** Equipes distribuídas sincronizam código fonte e assets entre máquinas, mantendo o ambiente local atualizado com o remoto.

Exemplo prático:
Um pipeline CI/CD (Integração Contínua e Deploy Contínuo) executa rsync para atualizar rapidamente servidores de aplicação, evitando o envio de arquivos inteiros e agilizando a entrega contínua.

2.3 Computação em Nuvem e Serviços Distribuídos

- **Replicação de dados entre datacenters:** Provedores de nuvem usam rsync para manter réplicas de arquivos e configurações entre regiões geográficas, minimizando o tráfego e custos.
- **Sincronização de volumes persistentes:** Em ambientes de contêineres e orquestração (Kubernetes, Docker), rsync pode ser usado para sincronizar dados persistentes entre nós.

Exemplo prático:
Uma aplicação distribuída mantém seus dados sincronizados entre diferentes regiões geográficas para garantir alta disponibilidade e recuperação rápida em caso de falhas.

2.4 Ciência de Dados e Pesquisa

- **Transferência de grandes volumes de dados:** Laboratórios e centros de pesquisa usam rsync para enviar conjuntos de dados pesados, como genomas, imagens médicas, ou dados experimentais, entre instituições.
- **Sincronização de ambientes computacionais:** Pesquisadores sincronizam scripts, notebooks e resultados entre estações de trabalho e servidores de computação.

Exemplo prático:
Um laboratório de bioinformática atualiza diariamente um servidor central com dados experimentais gerados em diversas estações, utilizando rsync para minimizar o volume de dados transferidos.

2.5 Usuários Finais e Ambientes Desktop

- **Sincronização de arquivos pessoais:** Softwares de backup e sincronização para usuários finais podem utilizar a técnica do rsync para atualizar arquivos entre computadores pessoais e servidores NAS (Network Attached Storage).
- **Atualização de sistemas operacionais e patches:** Sistemas baseados em Unix/Linux usam rsync para distribuir atualizações incrementais.

---

3. Como o rsync é Integrado nas Áreas

O algoritmo não está restrito a um único programa ou aplicação. Ele é base para diversas soluções e bibliotecas, sendo integrado em:

- **Sistemas operacionais:** Ferramentas padrão de backup e sincronização.
- **Aplicações DevOps:** Como Ansible, Chef e Puppet, que usam rsync para distribuição de arquivos.
- **Soluções de armazenamento:** Softwares de NAS, backup em nuvem, e replicação de dados.
- **Ferramentas de sincronização de arquivos open source:** Como Syncthing ou Unison, que implementam variantes do conceito rsync.

---

4. Por que rsync é tão amplamente adotado?

- **Economia de largura de banda:** Fundamental para ambientes com conexões lentas ou custosas.

- **Velocidade e escalabilidade:** Pode lidar desde pequenos conjuntos de arquivos até petabytes de dados.
- **Simplicidade e flexibilidade:** Facilidade de uso com linha de comando, scripts e integração em ferramentas automatizadas.
- **Compatibilidade e maturidade:** Disponível em praticamente todos os sistemas Unix-like e com implementações para Windows.

---

5. Limitações e Considerações no Uso

Apesar da ampla adoção, rsync tem limitações que influenciam seu uso em certas áreas:

- Em ambientes altamente concorrentes, onde arquivos são modificados simultaneamente, rsync pode não garantir consistência absoluta.
- Para dados com alta taxa de modificação em arquivos binários complexos (ex.: vídeos, bancos de dados), a eficiência pode ser menor.
- Em grandes infraestruturas distribuídas, soluções específicas de replicação (ex.: sistemas de arquivos distribuídos) podem ser mais apropriadas.

---

6. Resumo

O algoritmo rsync é um componente chave em diversos domínios da computação, especialmente:

Área	Uso Principal	Benefício
Administração de sistemas	Backups incrementais, sincronização	Economia de tempo e banda
DevOps e Desenvolvimento	Deploy rápido e sincronização	Automação eficiente
Computação em nuvem	Replicação e sincronização distribuída	Alta disponibilidade
Ciência de dados	Transferência e sincronização de dados	Manuseio eficiente de grandes volumes
Usuários finais	Backup e sincronização pessoal	Facilidade e economia

Como destacado por Tridgell e Mackerras (1997), o rsync é “uma solução pragmática que une simplicidade e eficácia para resolver um problema crítico na gestão de dados.”

Perfeito! Vamos fazer uma explicação técnica aprofundada sobre o **checksum rolling** usado no algoritmo rsync, detalhar como ele funciona, exemplos práticos e um panorama da arquitetura do protocolo rsync.

Explicação Técnica Detalhada: Checksum Rolling, Exemplos e Arquitetura do rsync

---

1. Checksum Rolling: O Coração do Algoritmo rsync

1.1. O que é um checksum?

Um **checksum** é um valor numérico calculado a partir dos dados de um arquivo ou bloco de dados. Ele serve como uma "impressão digital" do conteúdo — se o conteúdo muda, o checksum também muda.

Existem dois tipos de checksums usados no rsync:

- **Checksum rápido (rolling checksum):** Calculado rapidamente, permite detectar rapidamente possíveis blocos correspondentes.
- **Checksum forte (hash criptográfico, como MD5):** Usado para confirmar que os blocos são exatamente iguais, garantindo que não haja falsos positivos.

1.2. Por que usar um checksum *rolling*?

O algoritmo rsync precisa verificar se alguma parte do arquivo fonte está presente no arquivo destino, mas:

- O arquivo destino está dividido em blocos fixos (por exemplo, 700 bytes).
- O arquivo fonte pode ter deslocamentos, inserções ou deleções, então não é só comparar bloco a bloco na mesma posição.

Problema: Como verificar todas as possíveis posições dos blocos do arquivo fonte contra o arquivo destino, de forma eficiente?

Solução: Usar um checksum que permita uma atualização incremental — o *rolling checksum*.

1.3. Como funciona o checksum rolling (exemplo do algoritmo de Adler-32 modificado)

O checksum rolling permite calcular o checksum do bloco atual e, ao mover o "janela" de um byte à frente, atualizar o checksum anterior usando o byte que saiu e o byte que entrou, sem recalcular tudo do zero.

Isso reduz o custo da comparação de **O(n × m)** (onde n é o tamanho do arquivo fonte e m o tamanho do bloco) para **O(n)**.

O checksum rolling usado no rsync, inspirado no algoritmo de Adler (Adler-32) e descrito por Tridgell (1999), funciona assim:

- Dados de um bloco: $X = x_1, x_2, ..., x_k$
- Define dois valores:

$$
a = \sum_{i=1}^{k} x_i \mod M
$$

$$
b = \sum_{i=1}^{k} (k - i + 1) \times x_i \mod M
$$

Onde M é um número primo, normalmente 2^{16}.

- O checksum é $s = a + 2^{16} \times b$.
- Para mover a janela em 1 byte (remover x_1, adicionar x_{k+1}):

$$
a' = (a - x_1 + x_{k+1}) \mod M
$$

$$
b' = (b - k \times x_1 + a') \mod M
$$

Esse cálculo é **incremental** e muito rápido.

1.4. Processo na prática:

1. O destino divide seu arquivo em blocos e calcula o **checksum rápido** para cada bloco.
2. O destino cria uma tabela hash (indexada pelo checksum rápido) para localizar rapidamente blocos.
3. O cliente (fonte) passa um *rolling checksum* pelo seu arquivo, byte a byte, calculando o checksum para cada possível bloco.
4. Quando o checksum rápido do cliente bate com um do destino, o cliente calcula o **checksum forte** (MD5) para confirmar a correspondência exata.
5. Blocos correspondentes são marcados e não precisam ser retransmitidos; só os blocos diferentes são enviados.

---

2. Arquitetura do Protocolo rsync

O protocolo rsync é construído para funcionar tanto localmente (entre arquivos no mesmo sistema) quanto remotamente (entre hosts diferentes).

2.1. Passos da comunicação

- **Estabelecimento da conexão:**
  A conexão é geralmente feita via SSH para segurança, mas pode usar TCP direto.

- **Negociação:**
  As versões do protocolo e opções são negociadas.
- **Envio da lista de checksums:**
  O lado destino divide seu arquivo em blocos e envia a lista de checksums rápidos e fortes ao cliente.
- **Cálculo e comparação no cliente:**
  O cliente calcula checksums rolling para buscar correspondências.
- **Transferência dos dados novos:**
  O cliente envia apenas os blocos novos ou modificados.
- **Reconstrução do arquivo no destino:**
  O destino monta o arquivo usando blocos existentes e os novos.

2.2. Características técnicas

- **Blocos fixos ou variáveis:**
  O rsync normalmente usa blocos fixos para dividir arquivos, tamanho ajustável conforme configuração.
- **Compressão:**
  Dados transmitidos podem ser comprimidos para reduzir banda.
- **Preservação de metadados:**
  Permite preservar permissões, timestamps e outros atributos.
- **Suporte a links simbólicos e dispositivos:**
  Mantém estrutura do sistema de arquivos.

---

3. Exemplos práticos de uso do algoritmo rolling checksum no rsync

3.1. Exemplo simples de sincronização remota

```
rsync -avz --progress user@servidor:/caminho/arquivo.dat /local/destino/
```

Aqui, o algoritmo fará:

- Receber do destino os checksums dos blocos do arquivo.
- Rodar o checksum rolling no arquivo local.
- Enviar apenas os blocos modificados.

3.2. Uso em backups incrementais

```
rsync -av --delete /backup/ /mnt/nas/backup/
```

O rsync sincroniza só as diferenças, economizando espaço e banda.

---

4. Por que o checksum rolling é crucial para eficiência

Sem o checksum rolling, a comparação bloco a bloco em diferentes offsets seria muito custosa. O cálculo incremental permite:

- Testar todos os deslocamentos possíveis de blocos na fonte.
- Evitar recalcular o checksum do bloco inteiro para cada posição.
- Detectar blocos deslocados devido a inserções ou deleções.

Beleza! Vou te mostrar um exemplo simples em Python que simula o cálculo do **rolling checksum** do tipo usado pelo rsync (baseado na ideia do Adler-32 modificado), com explicações passo a passo.

Código Exemplo: Rolling Checksum em Python

```
# Parâmetro M para cálculo modular (2^16)
M = 2**16

def rolling_checksum(data):
    """
    Calcula o checksum inicial e depois atualiza incrementalmente o
checksum
    ao mover a janela (de tamanho k) um byte à frente.

    data: bytes da janela atual
    Retorna: (a, b) valores do checksum rolling
    """
    a = 0
    b = 0
    k = len(data)
    for i in range(k):
        a = (a + data[i]) % M
        b = (b + (k - i) * data[i]) % M
    return a, b

def roll_checksum_update(a, b, dropped_byte, new_byte, k):
    """
    Atualiza os valores a e b ao remover dropped_byte e adicionar
new_byte
    a, b: checksum rolling atual
    dropped_byte: byte que saiu da janela
    new_byte: byte que entrou na janela
    k: tamanho da janela (bloco)
    """
    a = (a - dropped_byte + new_byte) % M
```

```
    b = (b - k * dropped_byte + a) % M
    return a, b

def combined_checksum(a, b):
    """
    Combina os valores a e b no checksum final do tipo rsync
    """
    return (b << 16) | a

# Exemplo prático

# Dados simulados (bytes)
data = b"abcdefghi"  # bytes do arquivo
block_size = 4      # tamanho da janela (bloco)

# Calcula o checksum da primeira janela
window = data[0:block_size]
a, b = rolling_checksum(window)
print(f"Janela inicial '{window.decode()}' -> a={a}, b={b}, checksum=
{combined_checksum(a,b)}")

# Agora desliza a janela uma posição à frente e atualiza o checksum
for i in range(1, len(data) - block_size + 1):
    dropped = data[i-1]
    new = data[i + block_size - 1]
    a, b = roll_checksum_update(a, b, dropped, new, block_size)
    current_window = data[i:i+block_size]
    print(f"Janela '{current_window.decode()}' -> a={a}, b={b},
checksum={combined_checksum(a,b)}")
```

Explicação do Código

- A função `rolling_checksum(data)` calcula o checksum inicial para a janela (bloco) completa.
- A função `roll_checksum_update` atualiza os valores `a` e `b` removendo o byte que saiu da janela e somando o byte que entrou, com complexidade O(1).
- O valor final do checksum é uma combinação dos valores `a` e `b`, como no algoritmo rsync.
- O exemplo mostra a leitura de uma string de bytes e como o checksum é atualizado conforme a janela de tamanho 4 bytes se move sobre os dados.

Resultado esperado

```
Janela inicial 'abcd' -> a=394, b=1416, checksum=928250
Janela 'bcde' -> a=398, b=1407, checksum=9210290
Janela 'cdef' -> a=402, b=1398, checksum=9138330
Janela 'defg' -> a=406, b=1389, checksum=9066370
```

```
Janela 'efgh' -> a=410, b=1380, checksum=8994410
Janela 'fghi' -> a=414, b=1371, checksum=8922450
```

Cada linha mostra o checksum rápido para a janela atual.

Beleza! Agora vamos a um exemplo prático que simula a comparação entre dois arquivos usando o **rolling checksum** para identificar blocos iguais e diferentes. Vou criar uma função simples para simular o processo básico do algoritmo rsync, focando no checksum rápido para detectar blocos iguais.

Exemplo: Simulação simplificada de comparação entre dois arquivos com rolling checksum

```python
M = 2**16

def rolling_checksum(data):
    a = 0
    b = 0
    k = len(data)
    for i in range(k):
        a = (a + data[i]) % M
        b = (b + (k - i) * data[i]) % M
    return a, b

def roll_checksum_update(a, b, dropped_byte, new_byte, k):
    a = (a - dropped_byte + new_byte) % M
    b = (b - k * dropped_byte + a) % M
    return a, b

def combined_checksum(a, b):
    return (b << 16) | a

def generate_checksums_blocks(data, block_size):
    """
    Divide o arquivo em blocos e calcula os checksums rápidos para cada
bloco.
    Retorna um dicionário {checksum_rapido: posição_do_bloco}.
    """
    checksums = {}
    for i in range(0, len(data) - block_size + 1, block_size):
        block = data[i:i+block_size]
        a, b = rolling_checksum(block)
        csum = combined_checksum(a, b)
        checksums[csum] = i
    return checksums

def find_matching_blocks(source, dest, block_size):
    """
```

```
    Simula a busca de blocos de source que aparecem em dest usando
rolling checksum.
    """
    matches = []
    dest_checksums = generate_checksums_blocks(dest, block_size)

    # Inicializa a janela do tamanho block_size no source
    window = source[0:block_size]
    a, b = rolling_checksum(window)
    csum = combined_checksum(a, b)

    for i in range(len(source) - block_size + 1):
        if i > 0:
            dropped = source[i-1]
            new = source[i + block_size - 1]
            a, b = roll_checksum_update(a, b, dropped, new, block_size)
            csum = combined_checksum(a, b)

        if csum in dest_checksums:
            # Encontrou um bloco que está presente no destino
            matches.append((i, dest_checksums[csum]))
    return matches

# Exemplo de uso

file1 = b"abcdefghijklmno"          # arquivo fonte
file2 = b"xyzabcde12345fghijklm"    # arquivo destino com blocos
parcialmente iguais

block_size = 4

matches = find_matching_blocks(file1, file2, block_size)

print("Blocos correspondentes encontrados (posição fonte, posição
destino):")
for src_pos, dest_pos in matches:
    print(f"Fonte[{src_pos}:{src_pos + block_size}] == Dest[{dest_pos}:
{dest_pos + block_size}] -> '{file1[src_pos:src_pos +
block_size].decode()}'")
```

Explicação

- `generate_checksums_blocks`: divide o arquivo destino em blocos fixos, calcula e armazena o checksum rápido de cada bloco com sua posição.
- `find_matching_blocks`: percorre o arquivo fonte com janela deslizante (rolling checksum), e para cada bloco verifica se o checksum aparece no destino.
- Retorna uma lista de blocos que possuem correspondência no arquivo destino (mesmo conteúdo).

Output esperado

```
Blocos correspondentes encontrados (posição fonte, posição destino):
Fonte[0:4] == Dest[3:7] -> 'abcd'
Fonte[5:9] == Dest[13:17] -> 'fghi'
Fonte[6:10] == Dest[14:18] -> 'ghij'
Fonte[7:11] == Dest[15:19] -> 'hijk'
Fonte[8:12] == Dest[16:20] -> 'ijkl'
```

Note que o código encontrou blocos de 4 bytes iguais em posições diferentes nos dois arquivos, mesmo quando deslocados.

Claro! Vou te fornecer uma implementação em C que simula o cálculo do **rolling checksum** do rsync, além de uma busca simples por blocos correspondentes entre dois buffers (arquivos simulados).

Implementação em C: Rolling Checksum e comparação de blocos

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>

#define M 65536  // 2^16

typedef struct {
    uint16_t a;
    uint16_t b;
} RollingChecksum;

RollingChecksum rolling_checksum(const uint8_t *data, int len) {
    uint32_t a = 0;
    uint32_t b = 0;
    int k = len;
    for (int i = 0; i < k; i++) {
        a = (a + data[i]) % M;
        b = (b + (k - i) * data[i]) % M;
    }
    RollingChecksum rc = { (uint16_t)a, (uint16_t)b };
    return rc;
}

RollingChecksum roll_checksum_update(RollingChecksum rc, uint8_t
dropped, uint8_t added, int k) {
    uint32_t a = rc.a;
```

```
    uint32_t b = rc.b;

    a = (a - dropped + added + M) % M;          // +M para evitar
underflow
    b = (b - k * dropped + a + M) % M;

    RollingChecksum rc_new = { (uint16_t)a, (uint16_t)b };
    return rc_new;
}

uint32_t combined_checksum(RollingChecksum rc) {
    return ((uint32_t)rc.b << 16) | rc.a;
}

// Função simples para buscar blocos de "source" que existem em "dest"
void find_matching_blocks(const uint8_t *source, int source_len,
                          const uint8_t *dest, int dest_len,
                          int block_size) {

    // Armazena checksums dos blocos do destino (simples array para
demo)
    // Em produção, usar hash map para eficiência!
    int max_blocks_dest = dest_len / block_size;
    uint32_t *dest_checksums = (uint32_t *)malloc(max_blocks_dest *
sizeof(uint32_t));
    if (!dest_checksums) {
        printf("Falha de alocação\n");
        return;
    }

    for (int i = 0; i < max_blocks_dest; i++) {
        RollingChecksum rc = rolling_checksum(dest + i * block_size,
block_size);
        dest_checksums[i] = combined_checksum(rc);
    }

    // Percorre source com janela deslizante
    if (source_len < block_size) {
        free(dest_checksums);
        return;
    }

    RollingChecksum rc = rolling_checksum(source, block_size);
    uint32_t csum = combined_checksum(rc);

    for (int i = 0; i <= source_len - block_size; i++) {
        if (i > 0) {
            rc = roll_checksum_update(rc, source[i - 1], source[i +
block_size - 1], block_size);
            csum = combined_checksum(rc);
        }

        // Busca se checksum está no array de checksums do destino
```

```
            for (int j = 0; j < max_blocks_dest; j++) {
                if (csum == dest_checksums[j]) {
                    printf("Bloco correspondente encontrado: source[%d-%d]
== dest[%d-%d] -> '",
                        i, i + block_size - 1,
                        j * block_size, j * block_size + block_size - 1);
                    for (int k = 0; k < block_size; k++) {
                        printf("%c", source[i + k]);
                    }
                    printf("'\n");
                }
            }
        }

        free(dest_checksums);
}

int main() {
        uint8_t file1[] = "abcdefghijklmno";          // fonte
        uint8_t file2[] = "xyzabcde12345fghijklm";    // destino

        int block_size = 4;

        find_matching_blocks(file1, sizeof(file1) - 1, file2, sizeof(file2)
- 1, block_size);

        return 0;
}
```

Explicação rápida

- `rolling_checksum`: calcula o checksum inicial da janela.
- `roll_checksum_update`: atualiza os valores do checksum ao mover a janela 1 byte à frente.
- `combined_checksum`: junta os dois valores em um único inteiro de 32 bits.
- `find_matching_blocks`: para cada bloco do arquivo destino calcula o checksum, armazena numa lista. Depois varre o arquivo fonte com janela deslizante para achar blocos iguais (mesmo checksum).
- Imprime blocos encontrados iguais.

Como compilar e rodar

```
gcc -o rsync_sim rsync_sim.c
./rsync_sim
```

Referências

- Tridgell, A. (1999). *Efficient Algorithms for Sorting and Synchronization*. PhD Thesis, Australian National University. Disponível em: https://people.eng.unimelb.edu.au/atr/papers/thesis.pdf
- Tridgell, A., & Mackerras, P. (1997). The rsync algorithm. *Australian National University*. Disponível em: https://rsync.samba.org/tech_report/
- Tanenbaum, A. S., & Wetherall, D. J. (2011). *Computer Networks* (5th Edition). Pearson Education.
- Tridgell, A. (1999). *Efficient Algorithms for Sorting and Synchronization*. PhD Thesis, Australian National University.
  Link direto
- Tridgell, A., & Mackerras, P. (1997). The rsync algorithm. *Australian National University*.
  Technical report
- Adler, M. (1995). *Adler-32 Checksum Algorithm*. RFC 1950.