

Arquivos distribuídos

Relatório de implementação do trabalho 2
INE5645 - Programação Paralela e Distribuída.

Giovane Pimentel de Sousa - 22202685
Guilherme Henriques do Carmo - 22201630
Isabela Vill de Aquino - 22201632

December 15, 2024

Introdução

O objetivo deste trabalho é a implementação de uma aplicação cliente/servidor distribuída utilizando sockets TCP para transferência de arquivos. A aplicação simula operações comuns como as realizadas por comandos Unix, como ‘scp’ e ‘cp’, mas com funcionalidades específicas, incluindo suporte para recomeço de transferências interrompidas e controle de múltiplas conexões simultâneas. Adicionalmente, buscou-se aplicar conceitos de programação paralela e distribuída para garantir a eficiência no processamento e atendimento de múltiplos clientes. Este relatório apresenta a arquitetura, os detalhes de implementação e as decisões técnicas tomadas durante o desenvolvimento do sistema.

Como executar

Servidor

A compilação segue o formato básico de qualquer aplicação Go. A execução do servidor segue o formato de aplicações Unix:

```
$ cd daemon && go build . # Este comando produzirá o binário remcp-serv.  
$ ./remcp-serv daemon
```

Caso queira se saber qualquer saída mostrada pelo servidor como logs, conexões, erros em curso, etc., basta executar sem a flag **daemon**, que o servidor será executado utilizando o terminal como standard output.

Com a flag **daemon** o servidor será executado como um daemon, também ficando disponível para aceitar conexões de clientes.

Cliente

O cliente se comporta de maneira semelhante a comandos Unix como ‘scp’ ou ‘cp’, interpretando o IP para determinar a direção da transferência. Exemplos de execução:

```
$ cd remcp && go build . # Este comando produzirá o binário remcp.  
$ ./remcp 127.0.0.1:/home/gio/Downloads/teste.txt /home/gio/arquivos_distribuidos/teste  
$ ./remcp /home/gio/Downloads/teste.txt 127.0.0.1:/home/gio/arquivos_distribuidos/teste
```

- Caso o IP esteja no **primeiro** argumento, a transferência será **servidor => cliente**.
- Caso o IP esteja no **segundo** argumento, a transferência será **cliente => servidor**.

Implementação

A implementação foi realizada em Go 1.23 e explorou várias técnicas e conceitos relevantes para sistemas distribuídos, incluindo:

Arquitetura

A aplicação é dividida em dois componentes principais:

1. **Servidor (*remcp-serv*)**: Executa como um daemon, permitindo o atendimento simultâneo de múltiplos clientes, respeitando o limite máximo configurado.

2. **Cliente (*remcp*)**: Realiza transferências de arquivos, interpretando a direção com base nos argumentos fornecidos.

Configurações

As principais configurações, como número máximo de conexões simultâneas e velocidade de transferência, estão implementadas como constantes no código. Isso simplifica ajustes durante o desenvolvimento.

```
const (
    transferRate = 100000
    maxClients   = 2
)
```

Sendo:

- **transferRate**: taxa transferência máxima, que eventualmente será dividida entre os clientes.
- **maxClients**: o máximo de clientes que o servidor aceita simultaneamente.

Controle de concorrência

Para garantir a integridade durante a execução paralela, mutexes foram utilizados nas regiões críticas. O servidor distribui a banda disponível igualmente entre os clientes conectados, ajustando dinamicamente a velocidade de transferência.

```
mutexCurrentClients.Lock()
if currentClients >= maxClients {
    ack := []byte{0}
    conn.Write(ack)
    conn.Close()
} else {
    ack := []byte{1}
    conn.Write(ack)
    currentClients++
    go handleConn(conn) // Handle of requests
}
mutexCurrentClients.Unlock()
```

Acknowledgments (ACKs)

O uso de "acknowledgments" foi essencial para evitar problemas de mensagens malformadas. Sempre que uma mensagem é enviada entre cliente e servidor, o receptor confirma o recebimento antes de prosseguir.

```

// Acknowledgment to start receive file
ack := make([]byte, 1)
if _, err := conn.Read(ack); err != nil {
    return err
}

```

Mecanismo de retry

Quando o número máximo de conexões é atingido, o cliente tenta se reconectar automaticamente ao servidor. O comportamento segue esta lógica:

- Até 5 tentativas são realizadas, com um intervalo de 5 segundos entre elas.
- Caso o limite persista, a conexão é encerrada com um erro.

```

var conn net.Conn
var err error
attemp := 0
for {
    attempt++
    if attempt > maxAttempts {
        fmt.Println("Timeout!")
        return
    }

    conn, err = net.Dial("tcp", ip+Port)
    if err != nil {
        fmt.Println(err)
        return
    }

// Acknowledgment connection from server
ack := make([]byte, 1)
_, err = conn.Read(ack)
if err != nil {
    fmt.Errorf("Error on read acknowledgment from server. %s", err)
    return
}

if ack[0] != 1 {
    if attempt == 1 {
        fmt.Println("Server overloaded.")
    }

    fmt.Println("Retrying connection in 5 seconds...")
} else {
    break
}

time.Sleep(5 * time.Second)
}

```

Recomeço de transferências

Quando uma transferência é interrompida por qualquer circunstância, elas podem ser retomadas da seguinte forma:

- Durante a inicialização de uma transferência, o cliente verifica a existência de arquivos *.part* e informa ao servidor o ponto de recomeço. Assim o servidor só precisa enviar o restante dos dados, não o todo.

```

func getOrCreateFilePart(filepath, destinationPath string) (*os.File, error) {
    filename := utils.GetFilenameFromPath(filepath)

    // Open file or create if it doesn't exist
    file, err := os.OpenFile("/tmp/"+filename+".part", os.O_RDWR|os.O_CREATE, 0666)
    if err != nil {
        return nil, err
    }

    return file, nil
}

```

- Arquivos parciais (.part) são criados na pasta /tmp, sendo ela utilizada como diretório padrão neste projeto. Quando a transferência é finalizada, o arquivo é movido para o diretório inicialmente escolhido como destino, perdendo a extensão .part do seu nome.

```

file.Close()
filename := utils.GetFilenameFromPath(filepath)
if err := moveFile("/tmp/"+filename+".part", destinationPath+filename); err != nil {
    return err
}

fmt.Println("\nFile successfully received!")
return nil

```

Fluxo de transferências

1. **Servidor => Cliente:** O cliente recebe os dados em partes, armazenando temporariamente até concluir o download.

2. **Cliente => Servidor:** A transferência ocorre de maneira direta, sem throttling ou controle adicional.

```

func handleConn(conn net.Conn) {
    defer decrementClients() // Garantir o decremento dos clientes.
    defer conn.Close()
    flag := make([]byte, 1)
    conn.Read(flag)

    // The flag is what the connection starter wants to.
    // If he wants to receive a file, the server will send the file.
    // If he wants to send a file, the server will receive the file.
    switch flag[0] {
    case 0: // flagReceiveFile
        err :=.sendFile(conn)
        if err != nil {
            fmt.Printf("Error on connection with %s\n", conn.RemoteAddr(), err)
        }
    case 1: // flagSendFile
        err := receiveFile(conn)
        if err != nil {
            fmt.Printf("Error on connection with %s\n", conn.RemoteAddr(), err)
        }
    }
}

```

Figure 1: Controle de fluxo do Servidor

```
if isRemoteFile {
    dir, err := os.Open(destinationPath)
    if os.IsNotExist(err) { // Verifying if the directory of destination exists.
        fmt.Printf("Directory %s doesn't exist.\n", destinationPath)
        return
    }
    dir.Close()

    if err = receiveFile(conn, sourcePath, destinationPath); err != nil {
        fmt.Printf("\n%s", err)
    }
}

return
}

file, err := os.Open(sourcePath)
if os.IsNotExist(err) { // Verifying if the file of source exists.
    fmt.Printf("File %s doesn't exist.\n", sourcePath)
    return
}

if err = sendFile(conn, file, destinationPath); err != nil {
    fmt.Println(err)
}

return
```

Figure 2: Controle de fluxo do Cliente

Parametrizações e testes

Para validar o comportamento da aplicação em diferentes cenários, realizamos uma série de testes variando parâmetros como número máximo de conexões simultâneas, velocidade de transferência e tamanhos dos arquivos enviados. Esses testes tiveram como objetivo assegurar a implementação correta do sistema frente aos requisitos especificados.

Os testes para máximo de conexões foi menor pois como, no geral, o resultado é parecido, não se viu necessário tantos testes.

Foi validado como um servidor pode ficar sobrecarregado conforme muitas requisições/clients chegam. Para resolver esse tipo de problema, limitações de acesso são necessárias, evitando que o servidor se sobrecarregue, que é menos prejudicial do que não responder algumas requisições de imediato.

```
> ./remcp 127.0.0.1:/home/gio/Downloads/imb-1.jpg /home/gio/ > ./remcp 127.0.0.1:/home/gio/Downloads/imb-2.jpg /home/gio/arquivos_distribuidos/teste/  
arquivos_distribuidos/teste/ [=====] 30.73% [=====] 10.48%  
> ./remcp 127.0.0.1:/home/gio/Downloads/imb-3.jpg /home/gio/arquivos_distribuidos/teste/  
arquivos_distribuidos/teste/ Server overloaded...  
Retrying connection in 5 seconds...  
Timeout!  
> ./remcp 127.0.0.1:/home/gio/Downloads/imb-3.jpg /home/gio/arquivos_distribuidos/teste/  
arquivos_distribuidos/teste/ Server overloaded...  
Retrying connection in 5 seconds...  
Retrying connection in 5 seconds...  
|
```

Figure 3: Começo das transferências

Neste teste, configuramos o limite de conexões simultâneas no servidor para 2 clientes. Durante a execução, ao tentar estabelecer uma terceira conexão, o servidor retornou um *acknowledgment* de erro, e o cliente aguardou antes de realizar novas tentativas. Esse comportamento observado estava conforme o esperado.

O arquivo de envio tinha o tamanho de 1MB, sendo transferido a uma taxa de 10.000 bytes por segundo.

```

./remcp 127.0.0.1:/home/gio/Downloads/imb-1.jpg /home/gio/arquivos_distribuidos/teste/
[File successfully received!]
./remcp 127.0.0.1:/home/gio/Downloads/imb-2.jpg /home/gio/arquivos_distribuidos/teste/
[File successfully received!]
./remcp 127.0.0.1:/home/gio/Downloads/imb-3.jpg /home/gio/arquivos_distribuidos/teste/
[Server overloaded.
Retrying connection in 5 seconds...
Timeout!
./remcp 127.0.0.1:/home/gio/Downloads/imb-3.jpg /home/gio/arquivos_distribuidos/teste/
[Server overloaded.
Retrying connection in 5 seconds...
Timeout!
./remcp 127.0.0.1:/home/gio/Downloads/imb-3.jpg /home/gio/arquivos_distribuidos/teste/
[Server overloaded.
Retrying connection in 5 seconds...
[=] 2.16%

```

Figure 4: Novas tentativas do cliente 3 de se conectar

Então, a transferência do cliente 1 finalizou e o cliente 3 enfim consegue se conectar com sucesso. Durante o tempo de espera para uma nova conexão por parte do cliente 3 e a finalização de transferência do cliente 1, o cliente 2 obteve a banda de transferência toda para si, o que tornou, nem que seja por pouco tempo, a velocidade de transferência mais rápida. Porém, sua felicidade durou pouco, pois com o mecanismo de retry, logo que foi possível, a conexão foi estabelecida entre o cliente 3 e o servidor, tendo agora a banda de transferência dividida entre estes dois clientes.

```

./remcp 127.0.0.1:/home/gio/Downloads/imb-1.jpg /home/gio/arquivos_distribuidos/teste/
[File successfully received!]
./remcp 127.0.0.1:/home/gio/Downloads/imb-2.jpg /home/gio/arquivos_distribuidos/teste/
[File successfully received!]
./remcp 127.0.0.1:/home/gio/Downloads/imb-3.jpg /home/gio/arquivos_distribuidos/teste/
[File successfully received!
[=] 100.00%

```

Figure 5: Cliente 2 finaliza e cliente 3 fica com toda a taxa de transferência

Por fim, o cliente 2 finaliza, fazendo com que a banda de transferência fique toda para o cliente 3, tornando o envio do arquivo muito mais rápido em comparação às outras duas transferências para o cliente 1 e o cliente 2.

Conclusões

A implementação da aplicação cliente/servidor proposta atendeu integralmente aos objetivos do trabalho demonstrando a viabilidade de soluções distribuídas para transferência de arquivos. O uso de sockets TCP, em conjunto com mecanismos como controle de concorrência e *acknowledgments*, garantiu a robustez e a funcionalidade esperadas.

Os desafios enfrentados ao longo do desenvolvimento estavam previstos no escopo do trabalho e foram abordados com as técnicas adequadas sugeridas. A divisão dinâmica de banda, a limitação de conexões e o recomeço de transferências foram funcionalidades implementadas com sucesso e validadas em diferentes cenários de teste.

Embora a aplicação tenha cumprido os requisitos, melhorias futuras incluem a modularização de partes do código para facilitar manutenção e reutilização. No entanto, as escolhas feitas durante o desenvolvimento permitiram uma solução eficiente e funcional, alinhada aos objetivos do trabalho.