Relatório

Projeto 1 - Distributed Backup System

Grupo T6G06
José Guerra - up201706421
Martim Silva - up201705205

Procol Enhancements

BACKUP

Enhancement: This scheme can deplete the backup space rather rapidly, and cause too much activity on the nodes once that space is full. Can you think of an alternative scheme that ensures the desired replication degree, avoids these problems, and, nevertheless, can interoperate with peers that execute the chunk backup protocol described above?

A nossa versão do enhancement do backup tenta garantir que chunks que já tenham alcançado o replication degree pretendido não sejam guardados em mais peers, já que isto seria um desperdício de memória. Para isto, antes de iniciar os procedimentos para guardar um chunk, o peer espera um certo tempo aleatório entre 0 e 400 ms, durante o qual pode ou não receber mensagens STORED, depois disto procura num mapa, já utilizado na versão básica, se o replication degree pretendido já foi alcançado. Se tiver sido, abstém-se de guardar o chunk e não envia a mensagem STORED, senão, guarda o chunk e envia a mensagem STORED, como no protocolo normal. É de realçar que, embora não se chegue a guardar o chunk, o peer guarda como na versão normal informação acerca de quem é que guardou este chunk e o sobre o replication degree atual e o desejado. Este método não garante totalmente que não haja chunks com um current replication degree maior que o seu desired replication degree devido a concorrência entre peers e o delay ao receber de mensagens, no entanto, ajuda, já que certos peers acabam por não guardar certos chunks.

Devido a simplicidade do enhancement do backup, a interoperabilidade entre peers com versões diferentes é assegurada.

DELETE

“Enhancement: If a peer that backs up some chunks of the file is not running at the time the initiator peer sends a DELETE message for that file, the space used by these chunks will never be reclaimed. Can you think of a change to the protocol, possibly including additional messages, that would allow reclaiming storage space even in that event?”

A nossa versão do *enhancement* do delete cria um novo tipo de mensagem DELETEACK e tem o seguinte comportamento:

- ➔ O peer iniciador do protocolo backup é também o peer iniciador do protocolo DELETE (como foi dito pelo professor no fórum do moodle).
- ➔ Assim, ao iniciar o protocolo delete, o peer sabe quais os outros peer que fizeram STORED de chunks do ficheiro a ser apagado, através de um mapa já utilizado na versão básica. Assim, cria um novo mapa de valores com os peers que deram STORED de chunks daquele ficheiro e terão de enviar a mensagem DELETEACK (*deleteHistory*)
- ➔ Ao receber uma mensagem DELETE, um peer envia uma mensagem de DELETEACK com destino ao peer que iniciou o protocolo backup.
- ➔ Este peer (o que iniciou o delete) ao receber o ACK retira a entrada correspondente ao peer que envia o ACK do mapa *deleteHistory*.
- ➔ Na eventualidade de um peer que deveria ter estado “à escuta” para receber o DELETE não ter estado e recomeçar mais tarde, o peer que iniciou o delete detecta-o e envia de novo a mensagem DELETE para o canal de controlo.
- ➔ Este aviso é feito através do mapa *deleteHistory*. O peer que iniciou o delete, sempre que recebe uma mensagem, de qualquer peer, confirma no *deleteHistory* se o peer que enviou a mensagem está presente no mapa acima referido. Se tiver terá de o notificar como já referido.

  Esta abordagem tem as suas vantagens e desvantagens. Ponderando ambas consideramos que apesar de estarmos dependentes do peer que inicia o delete estar ativo para ativo, para notificar um peer que falte fazer DELETE, a alternativa seria guardar essa informação em todos os peers o que poderia tornar-se numa operação muito custosa.

  Todos estes passos acima descritos, só são realizados se todos os peers estiverem numa versão diferente da versão 1.0 . Assim a interoperabilidade é garantida com os peers com a versão 1.0.

RESTORE

“**Enhancement:** If chunks are large, this protocol may not be desirable: only one peer needs to receive the chunk, but we are using a multicast channel for sending the chunk. Can you think of a change to the protocol that would eliminate this problem, and yet interoperate with non-initiator peers that implement the protocol described in this section? Your enhancement **must use TCP** to get full credit.”

A nossa versão do enhancement do restore utiliza TCP para transferir os chunks entre os peers e tem o seguinte comportamento:

- ➔ O peer que inicia o protocolo de restore cria uma thread que cria um objeto da classe EnhancementRestore que vai ter um server socket, sendo que este é responsável por esperar por um pedido de ligação por parte de um dos outros peers que tenham chunks do ficheiro que se pretende dar restore.
- ➔ As mensagens GETCHUNK, responsáveis pela reconstrução do ficheiro, são enviadas para o canal de controlo.
- ➔ Nos outros peers, criam-se threads para processar as mensagens GETCHUNK. Caso o chunk esteja presente no peer então, depois de verificar que tanto eles como o peer que enviou a mensagem estão na versão 1.1 (versão em que os enhancements estão ativos), é criado um socket com o endereço IP de onde foi mandado a mensagem GETCHUNK e com a porta do canal MDR (“multicast data recovery channel”). Este socket vai ser usado para enviar a mensagem CHUNK contendo o conteúdo do chunk. Caso os peers não estejam todos na versão 1.1 então o protocolo funciona do mesmo modo que a versão básica. Assim a interoperabilidade é garantida com os peers com a versão 1.0.
- ➔ No peer que inicia o restore, dentro do thread que está a correr o EnhancementRestore, quando um dos outros peers quer enviar um chunk então este aceita a conexão e cria um outro thread para receber o próprio chunk.
- ➔ À medida que os chunks vão sendo recebidos são colocados num mapa, quando todos os chunks são recebidos, o ficheiro é reconstruído e todos os sockets que são necessários fechar são fechados.

Concurrency explained

A cada canal está associado um thread. Isto permite que cada canal seja independente.

```
private void setupChannels(String[] mcAddress, String[] mdbAddress, String[] mdrAddress) {
    this.controlChannel = new MC( peer: this, mcAddress[0], Integer.parseInt(mcAddress[1]));
    this.backupChannel = new MDB( peer: this, mdbAddress[0], Integer.parseInt(mdbAddress[1]));
    this.restoreChannel = new MDR( peer: this, mdrAddress[0], Integer.parseInt(mdrAddress[1]));

    new Thread(this.controlChannel).start();
    new Thread(this.backupChannel).start();
    new Thread(this.restoreChannel).start();
}
```

Cada peer no início do seu construtor chama a função setupChannels() que cria 1 thread para cada canal e chama a função setupExecutors(). Esta última função, aloca 5 threads para o envio de mensagens e 10 para receber mensagens para as variáveis descritas abaixo.

```
private void setupExecutors(){
    this.senderExecutor = Executors.newFixedThreadPool( nThreads: 5);
    this.receiverExecutor = Executors.newFixedThreadPool( nThreads: 10);
}
```

Temos também uma classe chamada *Dispatcher* responsável pela troca de mensagens. Cada objeto da classe Dispatcher tem um “role” associado. Tanto pode ser para a entrega de uma mensagem no canal, e nesse caso simplesmente envia para o canal pretendido, mas também pode ter o papel de “receiver”, no caso de ser construído a partir de um DatagramPacket. De acordo com o “role” associado, o objeto da classe Dispatcher terá diferentes funções.

```
@Override
public void run() {
    switch (this.type) {
        case RECEIVER:
            this.receiveMessageFromChannel();
            break;
        case SENDER:
            this.sendMessageToChannel();
            break;
        default:
            break;
    }
}
```

```java
public void receiveMessageFromChannel() {
    if(Integer.parseInt(this.message.getHeader().getSenderId()) == this.peer.getId()) retur
    this.peer.getDelete().checkIfPeerNeedsToDelete(this.message.getHeader().getSenderId());

    switch (this.message.getHeader().getMessageType()) {
        case PUTCHUNK:
            this.peer.getSpaceReclaim().storePutChunk(message);
            this.peer.getBackup().startStoredProcedure(message);
            break;
        case STORED:
            this.peer.updateRepDegreeInfo(message, increment: true);
            if(!this.message.getHeader().getVersion().equals("1.0"))
                this.peer.removeDeleteHistory(message);
            break;
        case GETCHUNK:
        case ENRESTORE:
            this.peer.getRestore().startChunkProcedure(message);
            break;
        case DELETE:
            this.peer.getDelete().deleteFile(this.message.getHeader().getFileId());
            if(!this.message.getHeader().getVersion().equals("1.0"))
                this.peer.getDelete().sendDeleteAckMessage(message.getHeader().getFileId(),
```

Em qualquer um dos protocolos, sempre que é necessário enviar uma nova mensagem isto é feito da seguinte forma:

```java
/**
 * Sends a DELETE message to all peers
 */
public void sendDeleteMessage() {
    Message request = new Message(DELETE, this.peer.getVersion(), Integer.toString(this.peer.getId()), this.fileId);

    for(int i = 0; i < MESSAGE_RETRIES; i++) {
        Dispatcher dispatcher = new Dispatcher(this.peer, request, this.peer.getControlChannel());
        this.peer.getSenderExecutor().submit(dispatcher);

        try {
            Thread.sleep((long) (MAX_DELAY * Math.random()));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Ou seja, a thread em uso contínua a execução com a programa e é criada uma nova thread para o envio da mensagem. Na nova thread, a mensagem irá então para a função run() do Dispatcher já mostrada acima e irá para a função sendMesssageToChannel().

```java
/**
 * Function responsible for delivering a message to a channel
 */
public void sendMessageToChannel() {
    DatagramPacket packet;
    DatagramSocket socket;

    try {
        socket = new DatagramSocket();
        byte[] buf = this.message.toBytes();
        packet = new DatagramPacket(buf, buf.length, InetAddress.getByName(this.address), this.port);
        socket.send(packet);
    } catch (IOException e) {
        Logs.logError( msg: "Error while sending a message to a channel");
        e.printStackTrace();
    }
}
```

Nesta função a mensagem é enviada para o canal que está no ciclo infinito while(true) a correr, sempre a receber mensagens.

```java
/**
 * Runs the Channel thread that handles received packets
 */
@Override
public void run() {
    while (true) {
        try {
            DatagramPacket received = this.receive();
            Dispatcher handler = new Dispatcher(this.peer, received);
            this.peer.getReceiverExecutor().submit(handler);
        } catch (IOException e) {
            Logs.logError( msg: "Error handling peer" + e);
        }
    }
}
```

Ao receber o DatagramPacket é criado novamente um thread para processar com o pedido recebido e o canal continua a execução para receber novos pedidos. Assim a execução em concorrência em garantida.