Trabalho Prático 2
Comunicações por Computador

Catarina Machado[a81047], Gonçalo Faria[a86264], and João Vilaça[a82339]

Universidade do Minho
Departamento de Informática

Resumo O constante desenvolvimento e optimização das infraestruturas de redes de computador têm desencadeado um aumento significativo da largura de banda destes sistemas e a diminuição da susceptibilidade a erros na transferência de dados. Por esta razão, o *Transmission Control Protocol (TCP)*[4], o principal protocolo da camada de transporte usado na Internet, desenvolvido para os sistemas de redes de há 50 anos atrás, não tem a capacidade de dispor destas novas e melhoradas capacidades de redes da actualidade. Neste trabalho, para fins educativos, no âmbito da unidade curricular de Comunicação por Computador, pretende-se desenvolver um protocolo alternativo, com as mesmas funcionalidades, sobre *User Datagram Protocol (UDP)*, que apresente um desempenho comparativamente superior ao TCP no domínio em que este será usado, ou seja, na nossa aplicação de transferência de ficheiros.

Keywords: UDP · TCP · PDU · Fiabilidade · Controlo de Fluxo · Controlo de Congestão.

1 Introdução

A desafio associado às implementações fiáveis de protocolos do nível de transporte surge na incerteza de entrega íntegra, ordenada, sem erros e única de mensagens de dados em redes de computadores reais. Esta incerteza é superada, no caso do TCP, através da detecção e recuperação de erros na transmissão, por via da numeração das mensagens e mantendo tanto no receptor como no emissor uma representação aproximada do estado da ligação.

A representação aproximada do estado da ligação é constantemente sincronizada, através da entrega e recepção de mensagens de dados e de controlo. Nas mensagens de controlo, para além de várias outras informações, contêm mensagens de confirmação cumulativa que indicam a recepção de mensagens de dados pelo receptor.

Dado que as redes de computador são limitadas quanto à quantidade de dados que podem transportar num dado instante, no estado da ligação, para evitar congestão, consta um mecanismo de regulação do envio de mensagens, denominado controlo de fluxo. A forma como este é implementado é o principal aspecto que define a performance do protocolo de transporte fiável num dado domínio aplicacional. O TCP, como controlo de fluxo, usa um mecanismo de

janela, que define um conjunto de dados em transmissão que, após o envio do último pacote desta, espera pela recepção de uma mensagem de confirmação para continuar o envio. Esta espera por sincronização, frequentemente, origina uma perda de performance.

Na eventualidade de perda ou corrupção de dados, o mecanismo usado nas principais implementações de TCP é, após o envio, esperar por um *timeout*, baseado no *round-trip-time*, e reenviar até que a mensagem de dados seja recebida com sucesso. No entanto, este mecanismo demora uma quantidade de tempo considerável e, para além disso, pode levar a retransmissões indevidas.

As retransmissões indevidas contribuem para o congestionamento da rede. Este congestionamento é agravado pois estas retransmissões indevidas introduzem mensagens inúteis que o receptor já recebeu. As retransmissões indevidas podem ser o resultado de um emissor agressivo, que decide retransmitir precocemente, ou um especulativo, que supõe que todos os pacotes de um dado conjunto foram perdidos. Extensões ao TCP, como o TCP SACK,[3], resolvem este problema, embora com um custo associado que limita a sua aplicabilidade.

O protocolo desenvolvido, para além da implementação de funcionalidades que garantem a entrega fiável de mensagens, obtém altas taxas de transferências para a aplicação, mesmo quando na presença de grandes atrasos e perdas na rede. Adicionalmente, para além do mecanismo de controlo de fluxo baseados em janela, tal como o constante nas implementações de TCP, implementamos um mecanismo baseado em controlo de taxa de envio, semelhante aos presente em [1,2], desta forma tornando independentes os mecanismos de controlo de falhas e de controlo de fluxo, permitindo a transmissão constante independentemente do estado dos dados previamente enviados.

2 Especificação do protocolo

2.1 Inicio de Ligação

O socket GCVTCP permite 2 tipos de ligações *peer-to-peer* e *peer-to-server*.

Ao iniciar o socket GCVTCP, este verifica se o *daemon* do protocolo encontra-se activo e, caso não esteja, este é ativado. O *daemon* do protocolo é responsável por escalonar os pedidos de ligação, recebidos no port 8626, e mantêm por referência o conjunto de sockets GCVTCP em ligação no sistema.

Ligação peer-to-peer Quando um socket A pretende iniciar uma ligação, este envia uma mensagem de início de ligação para o port 8626, o port por defeito do protocolo, do socket destino B. Nesse port como se encontra o *daemon* do protocolo à escuta este, ao receber a tentativa de ligação, caso o socket B estiver a tentar estabelecer ligação com o socket A, é enviada uma confirmação de ligação e a comunicação inicia-se.

Ligação peer-to-server Quando um socket A pretende iniciar uma ligação com um serversocket, este envia uma mensagem especial de inicio de ligação

para o port por defeito do protocolo que contém o identificador de um dado serversocket B. Caso no sistema B contenha o respectivo serversocket à escuta, um socket é criado e a ligação com A é estabelecida.

Tanto no caso do início de ligação *peer-to-peer* como no caso do início de ligação *peer-to-server*, se a confirmação de inicio de ligação for perdida, ao receber de novo a tentativa de inicio de ligação de um socket GCVTCP a mensagem de confirmação perdida é reenviada.

2.2 Fim de ligação

O processo de término de ligação pode ocorrer sempre que o socket em questão não tenha recebido uma mensagem num dado tempo de tolerância. O tempo de tolerância é o equivalente a 8 vezes o resultado da estimativa de *round-trip-time* mais 4 vezes a variância dessa estimativa. Para forçar o término da ligação, sempre que um socket é fechado este envia o pacote de controlo **BYE**. Caso o destinatário deste pacote **BYE** não o receba a sua extremidade da ligação termina da forma por defeito, quando o tempo de tolerância expirar.

2.3 Formato das mensagens protocolares (PDU)

O tamanho máximo de pacote por defeito do protocolo é 1460.

Pacote de Dados O pacote de dados destina-se a encapsular as mensagens de dados.

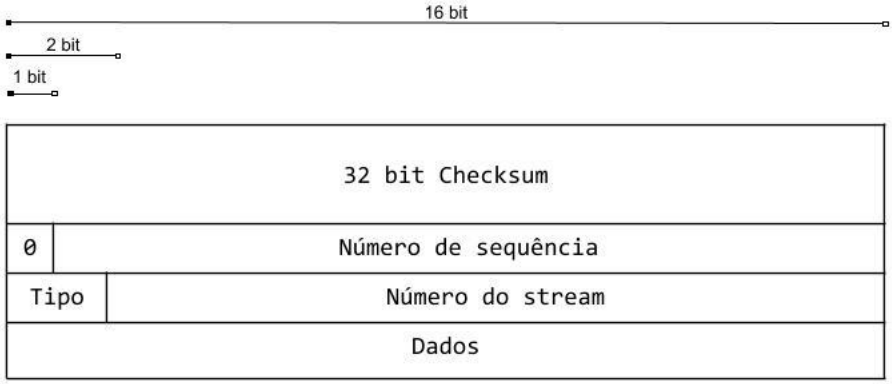

Figura 1. Pacote de Dados

O número de sequência é atribuído pelo emissor e introduz a ordem das mensagens enviadas.

O campo tipo caracteriza o pacote de dados enviado no contexto do *stream*. Se for do tipo **SOLO**, então o pacote é o único pacote de um dado *stream*. Se

for do tipo **FIRST**, então o pacote marca o início de um *stream*. Se for do tipo **LAST**, então o respectivo pacote marca o término de um *stream*. Por fim, o tipo **MIDDLE**, corresponde a um pacote que é membro de um *stream* mas precede um pacote do tipo **FIRST** e antecede um do topo **LAST**.

O número de *stream*, identifica o *stream* em que este pacote se encontra.

Pacote de Controlo O pacote de controlo destina-se a encapsular mensagens de controlo e está divididos em 6 tipos cada um desenhado para satisfazer um tipo de funcionalidades específicas. A informação particular a cada tipo de pacote de controlo encontrar-se no campo informação de controlo.

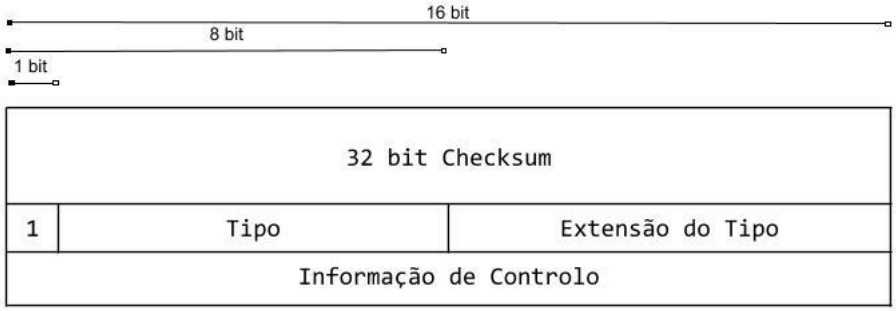

Figura 2. Pacote de Controlo

Os pacotes de controlo estão divididos em 6 tipos cada um desenhado para satisfazer um tipo de funcionalidades específicas. A informação particular a cada tipo de pacote de controlo encontrar-se no campo informação de controlo.

Tipos de pacotes de controlo: HI - Pacote associado ao início de ligação. Corresponde ao pacote de controlo com o campo tipo com valor 0. Na informação de controlo consta o tamanho máximo que as mensagens de dados terão, o número de sequência inicial do emissor deste pacote e o número máximo de *burst*, que é uma particularidade do mecanismo de controlo de fluxo.

OK - Pacote associado à confirmação cumulativa de pacotes recebidos. Corresponde ao pacote de controlo com o campo tipo com valor 1. Na informação de controlo consta o maior número de sequência, em pacotes de dados, que o emissor deste pacote já recebeu, o espaço livre no buffer que será usado também para limitar o tamanho de *burst*, o tempo de ida e volta e a variância do tempo de ida e volta.

SURE - Pacote associado à confirmação cumulativa de pacotes **OK**. Corresponde ao pacote de controlo com o campo tipo com o valor 2 e, conjuntamente com o pacote **OK**, é a principal ferramenta para o cálculo do tempo de ida e volta. Na informação de controlo consta o maior número de sequência que o emissor recebeu em pacotes de confirmação.

NOPE - Pacote associado à comunicação de pacotes perdidos. Corresponde ao pacote de controlo com o campo tipo com o valor 3. Na informação de controlo consta um conjunto de números de sequência respectivos aos pacotes de dados em falta pelo emissor da mensagem. Em alternativa a representar o conjunto de número de sequência extensivamente foi usada uma lista intervalar. Cada elemento da lista são dois inteiros que representam um intervalo de número de sequências.

BYE - Pacote associado ao fim de ligação. Corresponde ao pacote de controlo com o campo tipo com o valor 4.

FORGET IT - Pacote associado ao pedido para ignorar os dados transmitidos de um dado *stream*. Corresponde ao pacote de controlo com o campo tipo com o valor 5. O campo informação de controlo contém o número do *stream* que se pretende que seja ignorado.

SUP - Pacote associado ao pedido para a preservação da ligação. Corresponde ao pacote de controlo com o campo tipo com valor 6. Este pacote não leva informação adicional sendo que apenas se destina a provocar o recomeço do cronómetro de *timeout*.

2.4 Controlo de Entrega

Como mecanismo para assegurar a ordem dos dados enviados assim como detecção de perdas, tal como o usado no TCP, é usada a numeração das mensagens de dados. Esta numeração é atribuída ao campo número de sequência, constante em cada um dos pacotes de dados.

À medida que os pacotes de dados são recebidos, o receptor vai mantendo o valor do último pacote contíguo recebido. Este valor é alterado unicamente no momento em que o pacote com o número de sequência com uma unidade superior é recebido.

Para que o emissor tenha conhecimento de quais os pacotes o receptor já recebeu, é enviado, a cada período de controlo fluxo, caso este tenha sido alterado, uma confirmação cumulativa, o pacote **OK**, com o valor do último pacote contíguo recebido. Adicionalmente, para efeitos do cálculo da estimativa de *round-trip-time*, é usado um pacote de confirmação de confirmação, o pacote **SURE**, que, tal como o **OK**, é cumulativo e é possivelmente enviado no final de cada período de controlo de fluxo. Não é necessário que para todos os **OK** seja gerado um correspondente **SURE**.

Nós decidimos vincular o envio dos **OK**s e dos **SURE**s ao fim de cada período de controlo de fluxo pois desta forma a quantidade de tráfego na rede devido a pacotes de controlo é independente do débito desta, permitindo assim maximizar o rendimento do protocolo em redes com grandes débitos

2.5 Detecção de Erros

Através da análise dos pacotes no *wireshark* concluímos que o *datagramSocket* e *datagramPacket* não implementam o mecanismo de checksum indicado no UDP.

Por esta razão, introduzimos o checksum Adler-32. Escolhemos o Adler-32 devido à performance superior comparativamente aos outros algoritmos de 32 bits como, por exemplo, o CRC-32. No entanto, de acordo com a descrição presente em [5], o algoritmo apresenta desvantagens quanto à capacidade de detecção de alguns erros quando comparado com o CRC-32, sendo no entanto superior ao mecanismo por defeito do UDP e TCP.

2.6 Reordenação de Pacotes

Para que pacotes que chegam fora de ordem não sejam considerados perdidos e para que desta forma seja prevenido o envio de pedidos de retransmissão no buffer de recepção consta uma estrutura de dados que assegura a ordem dos pacotes à espera para serem processados.

A estrutura em uso corresponde ao que nós designamos por lista intervalar. A lista intervalar é composta por elementos que são intervalos de pacotes de dados. A motivação para a escolha desta estrutura foi simultaneamente a facilidade que esta apresenta para calcular a lista de intervalos em falta, a lista intervalar dual, e também a representação condensada que esta possui que permite a expressão de uma enorme quantidade de números de sequência com poucos elementos. A referida representação condensada é extremamente útil pois possibilita que pedidos de retransmissão contenham, quando necessário, o pedido de um número elevado de pacotes.

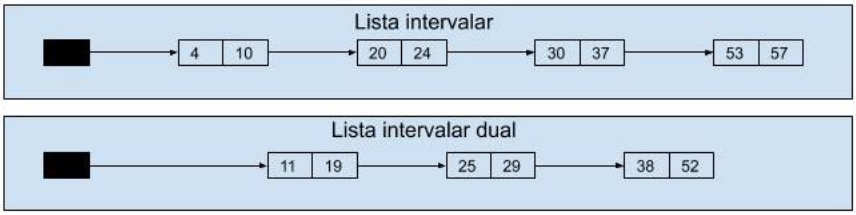

Figura 3. Listas intervalares

2.7 Pacotes Duplicados

Quando múltiplas cópias da mesma mensagem de dados são recebidas, à excepção da primeira, são todas descartadas.

2.8 Controlo de Fluxo

O mecanismo de controlo de fluxo que implementamos no nosso protocolo, em semelhança aos presente nos protocolos NETBLT[1] e UDT[2], é baseado no controlo de taxa de envio. De uma forma bastante distinta do TCP, existe um *burst*

de pacotes de dados que é enviado em cada período. Esse período é fixo, denominamos de período de controlo de fluxo e foi atribuído o valor de 10 milissegundos. O fator a determinar neste algoritmo de controlo de fluxo é, dependendo da presente capacidade da rede para transportar dados assim como da disponibilidade do buffer do receptor de os receber, calcular o tamanho do *burst* a usar.

Por defeito, o número de pacotes que compõem o *burst* inicial é 1. No final de cada período de controlo de fluxo, enquanto o modo de controlo de congestão está desactivado, o valor do *burst* é actualizado para o mínimo entre a soma do presente valor e o número de pacotes de dados confirmados no decorrer do período de controlo de fluxo passado, e o espaço livre no buffer do receptor.

O modo de controlo de congestão é activado ora quando é recebido um pedido de retransmissão do receptor, ou quando o *burst* atinge um valor máximo determinado no momento de estabelecimento da ligação

2.9 Controlo de Congestão

O mecanismo que nós introduzimos para lidar com a possível insatisfação da procura dos recursos da rede quando esta se encontra exposta a um excessivo tráfego de dados, foi o presente no TCP, *additive increase multiplicative decrease*(AIMD).

Quando activo, o algoritmo de controlo de congestão implementado, no final de cada período de controlo de fluxo acciona ora o *multiplicative decrease*, o *additive increase* ou nenhum dos dois.

O *multiplicative decrease*, que é accionado quando o último pacote **OK** recebido foi há mais tempo que a soma entre o valor do intervalo de controlo de fluxo e a soma da previsão de *round-trip-time* com o desvio dessa previsão, é atribuído ao *burst* o resultado da multiplicação de um factor constante b pelo mesmo, tal que $0 < b < 1$. O valor de b de momento usado é 0.6 .

O *additive increase*, que é accionado quando é recebida uma mensagem de confirmação no intervalo de controlo de fluxo passado, está definido como sendo o incremento do *burst* numa unidade.

Na eventualidade de nem a condição de *multiplicative decrease*, nem a de *additive increase* se verificar então o tamanho do *burst* é preservado.

Adicionalmente, foi introduzido um mecanismo que permite a saída do modo de controlo de congestão que é accionado no momento em que foram processados mais de 10 *additive increase* sucessivos ou quando o número de mensagens de dados confirmadas no passado intervalo de congestão é maior que o dobro do *burst*.

2.10 Controlo de Perdas

Para controlo de perdas, em alternativa ao mecanismo do TCP de, através de um temporizador no emissor ou de espera de confirmações duplicadas proceder com uma retransmissão, decidimos usar um mecanismo semelhante ao constante em [3,1,2]. Este mecanismo atribui o encargo de decidir explicitamente quais os pacotes a retransmitir ao receptor.

A recuperação de erros, por via de transmissão de perdas, aparenta ser um mecanismo superior ao presente nas implementações comuns do TCP pois, a qualquer momento, o receptor tem acesso a quais pacotes chegaram e quais os que não, evitando desta forma, quase que completamente, retransmissões indevidas.

Embora conceptualmente semelhante ao mecanismo presente em [3,1,2], o nosso utiliza uma distinta representação da lista de pacotes em falta. Dada a natureza da lista intervalar usada no buffer de pacotes de dados do receptor, no momento de envio de um **NOPE**, o pacote correspondente ao pedido de retransmissão, enviará a lista intervalar dual, que contém os intervalos de números de sequência em falta.

O pedido de retransmissão é realizado quando as seguintes duas condições se verificam, o último pedido de retransmissão ocorreu à mais de que a soma da média móvel do *round-trip-time* com a média móvel do desvio padrão e o último **OK** mandado foi há mais tempo que a soma da média móvel do desvio padrão e o período de controlo de fluxo.

Ao receber o pedido de retransmissão, o emissor activa o evento correspondente que, consequentemente, submete o executor a colocar na frente da fila de envio, ordenadamente, os pacotes especificados por esse pedido. Como os pacotes são postos na fila de envio o emissor trata o tráfego provocado pela retransmissão da mesma forma que o tráfego de pacotes de dados não outrora enviados.

Devido a esta decisão, todo o tráfico introduzido é uniformemente controlado pelos algoritmos de controlo de congestão e fluxo, permitindo a independência das funcionalidades de recuperação de dados e controlo de fluxo, assim como, o controlo da carga provocada pelo emissor na rede.

Para alguns casos em que, por exemplo, o emissor não tem mais dados para mandar e os últimos pacotes de dados mandados foram perdidos, é possível que a retransmissão destes pacotes não ocorra com o mecanismo de retransmissão normal. No entanto, dado que é necessária fiabilidade mesmo nestas circunstâncias, por vezes, o emissor também efectua retransmissões mesmo sem receber um **NOPE**, embora invulgarmente.

2.11 Multiplexação de Canal

A capacidade de multiplexação de várias ligações no mesmo socket possibilita a eliminação da sobrecarga associada ao início de uma nova ligação, a redução do esforço computacional necessário para o suporte da hierarquia de classes do socket GCVTCP e a melhor exploração dos mecanismos de controlo de congestão, de fluxo e de erros do nosso protocolo.

Esta funcionalidade foi satisfeita através da introdução do conceito de *stream*. Os pacotes pertencentes a um *stream* devem essencialmente ser entregues em conjunto, embora não necessariamente em simultâneo. Mais precisamente, em cada invocação no socket das funcionalidades de envio, um novo *stream* é criado e é associado a todos os pacotes de dados enviados nessa invocação. Sendo que é possível que ocorram em simultâneo várias invocações de funcionalidades de envio no socket, é possível obter multiplexação da ligação que é processada uniformemente no nível de transporte.

No receptor, como os pacotes de dados estão identificados com o número de *stream* a que pertencem, após retirar os pacotes de dados, ordenados pelo número de sequência, do buffer de recepção estes são colocados no *pipe* do *stream* a que pertencem. Dado estes pacotes de dados estão identificados pelo tipo **SOLO**, **FIRST**, **MIDDLE** e **LAST**, ao receber um **FIRST** ou **SOLO** é aberto um novo *pipe* e ao receber um **LAST** ou **SOLO** este é fechado.

3 Implementação

Depois de definida uma API estável para a camada de transporte, foi iniciado o desenvolvimento da camada aplicacional, totalmente agnóstica ao tipo de transporte utilizado, que implementa então a transferência propriamente dita dos ficheiros e uma listagem de ficheiros disponíveis.

Esta camada é constituída por 3 blocos de destaque:

1. Encriptação de comunicação
2. Autenticação
3. Processamento de pedidos
4. Atualização global do sistema

Para cada exemplo consideremos uma conexão entre a Alice e o Bob.

3.1 Encriptação de comunicação

A Alice pretende trocar alguns ficheiros com o Bob, e por isso incia uma ligação. Após esta ser estabelecida (Alice: `connect(String host)`; Bob: `GCVSocket listen()` é, em todos os casos, executado um processo de troca de chaves criptográficas.

Depois de geradas as chaves, a Alice envia a sua chave pública assimétrica (RSA, 2048 bits) para o Bob, que a recebe e guarda. Em seguida, o Bob envia a sua chave assimétrica pública (RSA, 2048 bits) e a uma chave simétrica (AES, 128 bits), gerada única e exclusivamente para esta conexão, encriptada com a chave pública assimétrica da Alice, para que só ela e o Bob consigam ter acesso aos dados da mensagem. A Alice desencripta a chave assimétrica e guarda-a.

Listing 1.1. Encriptação recorrendo a chave simétrica

```
public String encrypt(String value) {
    try {
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);

        byte[] encrypted = cipher.doFinal(value.getBytes());
        return new String(Base64.getEncoder().encode(encrypted));
    } catch (Exception ex) {
        ex.printStackTrace();
    }

    return value;
}
```

Daqui em diante, todas as mensagens são assinadas digitalmente, por exemplo, quando a Alice envia um pedido ao Bob, assina esse pedido com uma assinatura feita com a sua chave assimétrica privada. Assim o Bob, tendo a chave pública da Alice, consegue garantir que foi ela que enviou aquela mensagem e que a mesma não foi alterada.

Listing 1.2. Exemplo do método sign

```
public static String sign(PrivateKey privateKey, String data) {
    try {
        Signature rsa = Signature.getInstance("SHA1withRSA");
        rsa.initSign(privateKey);
        rsa.update(data.getBytes());
        return Base64.getEncoder().encodeToString(rsa.sign());
    } catch (Exception e) {
        e.printStackTrace();
    }

    return data;
}
```

Para além disso, todas as mensagens são encriptadas pela chave simétrica e a partir daí não sofrem mais nenhuma alteração.

3.2 Autenticação

No que diz respeito à autenticação foi feita uma implementação individual por máquina na rede, ou seja, cada uma das aplicações mantém a sua própria lista de utilizadores conhecidos e faz a sua própria gestão de autenticação e acessos. Isto permite simplificar o tráfego na rede e acima de tudo a gestão de concorrência, quer em termos registo de utilizadores e na atualização dos seus dados por todas as aplicações na rede.

Fazer esta simplificação é então vantajoso e não traz nenhum problema em termos globais de gestão de utilizadores. Como é feita sempre uma autenticação no início de qualquer ligação, qualquer que seja o peer, o utilizador pode utilizar as credenciais que bem entender para cada uma destas ligações sem prejuízo da rapidez do serviço porque teria sempre de verificar a sua identidade independentemente do método com que o faça.

Todos os utilizadores têm então duas maneiras de colocar ficheiros em determinado peer, podem simplesmente enviar o ficheiro para o peer e ele fica publicamente disponível para acesso por parte de qualquer utilizador com acesso permitido a essa aplicação, ou então pode fazer uma transferência que fique ”privada”entre os dois participantes na ligação.

private static Map¡String, User¿ users

Listing 1.3. Autenticação

```
private static Map<String, User> users;

private void autenticacao(String username, String password) {
    if (users.containsKey(username)) {
        User u = users.get(username);
        if (u.getPassword().equals(password)) {
            loggedInUser = user;
        }
```

```
    } else {
        users.put(username, new User(username, password));
        path += username + "/";
        File directory = new File(path);
        if (!directory.exists()){
            directory.mkdirs();
        }
    }
}
}
```

3.3 Processamento de pedidos

Nesta secção vai ser omitido todo o processo de codificação, encriptação, e assinatura digital mencionado anteriormente (Encriptação de comunicação) pois o mesmo é sempre efetuado e de maneira muito semelhante para todos os tipos de pedidos seguintes.

ConnectionType.PUT A Alice envia um pedido ao Bob para ele armazenar um determinado ficheiro ou lista de ficheiros. O Bob armazena o ficheiro e comunica aos restantes peers que possui aquele ficheiro.

ConnectionType.GET A Alice envia ao Bob um pedido para que ele lhe envie um ou vários ficheiros, e o Bob responde-lhe com todos os ficheiros que ela pediu que existam e que ela tenha permissões para aceder.

ConnectionType.LIST, ConnectionType.INFORM A Alice envia ao Bob um pedido LIST para listar todos os ficheiros a que ela tenha acesso que o Bob possua. O Bob envia-lhe como resposta um INFORM que contém uma lista com os nomes desses ficheiros.

ConnectionType.ASK, ConnectionType.INFORM A Alice envia ao Bob um pedido a perguntar todos os locais que o Bob conhece onde ela poderá adquirir o ficheiro que ela procura. Para depois a Alice ser capaz de descarregar fragmentos desse ficheiro de vários servidores em simultâneo.

ConnectionType.FRAG A Alice envia ao Bob um pedido de um fragmento de um ficheiro (por exemplo, fragmento 3, número de fragmentos 7). O Bob envia-lhe então apenas essa pequena porção do ficheiro. Assim a Alice pode descarregar de vários servidores em simultâneo cada um dos fragmentos do ficheiro.

ConnectionType.SHARE O Bob informa todos os peers que conhece, enviando um pedido SHARE, que a partir daquele momento ele possui determinado ficheiro.

3.4 Atualização global do sistema

Todos os peers vão mantendo ao longo do seu tempo de atividade no sistema uma espécie de mapa da constituição da rede em termos de outras aplicações ativas e ficheiros que elas têm disponíveis. Quando um peer é iniciado pode ser passada como parâmetro uma lista de outras máquina atualmente ativas. Antes deste estar pronto a receber pedidos comunica então a todos este peers que ele a partir de agora existe na rede.

3.5 Bibliotecas de Suporte Usadas

- **java.net.DatagramSocket** - Canal UDP utilizado
- **java.net.InetAddress** - Resolução de nomes e IP's
- **java.util.zip** - Compressão dos dados a serem transferidos
- **javax.crypto** e **java.security** - Encriptação dos Dados, quer RSA, quer AES
- **java.util.Base64** - Codificação e descodificação dos conteúdos dos ficheiros a serem transferidos

4 Testes e resultados

4.1 Testes funcionais e de carga

Ao longo de todo o processo fizemos também bastantes teste quer na camada de transporte quer na camada aplicacional para garantir que a atomicamente os métodos implementados tinham o comportamento esperado e que resultados obtidos fossem corretos.

No fim, de maneira mais global, desenvolvemos testes funcionais e de carga para a aplicação como um todo de forma a garantir que os requisitos definidos para esta solução de software estejam efetivamente a ser respondidos.

Listing 1.4. Tratamento de uma grande quantidade de dados

```
for (int i = 1000000; i >= 1; i/=10) {
    char[] c = new char[i];
    Arrays.fill(c, 'a');
    String a = System.currentTimeMillis() + "-" + i + "-";
    String s = new String(c);
    Connection.send(cs, (a + s).getBytes());
}
```

4.2 Resultados e comparações

A seguinte análise estatística resultou da simulação no emulador CORE da transferência de uma quantidade de dados de 100MB. Os tamanhos das amostras foram variáveis entre estados de rede diferente mas os mesmos entre GCV e TCP. As estimativas para os estados "unlimited"e "1Gbps, delay: 500us, loss: 15%,duplicate: 15%"resultaram de 100 observações, enquanto para estado "bandwidth: 64kpbs, delay: 80us"de apenas 20 observações.

GCV

Estado da Rede / Propriedade estatística	unlimited	bandwidth: 64kpbs delay: 80us	1Gbps delay: 500us loss: 15% duplicate: 15%
média	58.2 ms	88790.66 ms	1524,9 ms
variãncia	14.49	24453.06	1283.83

TCP

Estado da Rede / Propriedade estatística	unlimited	bandwidth: 64kpbs delay: 80us	1Gbps delay: 500us loss: 15% duplicate: 15%
média	152.36 ms	181397 ms	8263,33 ms
variãncia	7.57	4348.56	1604.91

5 Conclusões

Foi desenvolvida com sucesso uma solução que resolve de forma eficaz o desafio associado às implementações fiáveis de protocolos do nível de transporte no que toca à incerteza de entrega íntegra, ordenada, sem erros e única de mensagens de dados em redes de computadores reais.

O protocolo desenvolvido, para além da implementação de funcionalidades que garantem a entrega fiável de mensagens, obtém altas taxas de transferências para a aplicação, mesmo quando na presença de grandes atrasos e perdas na rede. Adicionalmente, para além do mecanismo de controlo de fluxo baseados em janela, tal como o constante nas implementações de TCP, implementamos um mecanismo baseado em controlo de taxa de envio, semelhante aos presente em [1,2], desta forma tornando independentes os mecanismos de controlo de falhas e de controlo de fluxo, permitindo a transmissão constante independentemente do estado dos dados previamente enviados.

Referências

1. D. D. Clark, M. L. Lambert, and L. Zhang. NETBLT: a high throughput transport protocol. *ACM SIGCOMM Computer Communication Review*, 2004.
2. Y. Gu and R. L. Grossman. UDT: UDP-based data transfer for high-speed wide area networks. *Computer Networks*, 2007.
3. M. Mathis, Mahdavi, S. Floyd, LBNL, and A. Romanow. RFC 2018, 1996.
4. J. POSTEL. Transmission Control Protocol (TCP). 1981.
5. R. Stewart, J. Stone, and D. Otis. RFC 3309: Stream Control Transmission Protocol (SCTP) Checksum Change, 2002.