Descrição do Projeto de Disciplina Sistemas Distribuídos - PROCC/UFS -2013/1: **Decifrador de Senhas Distribuído**

Uma tradução de Felipe Carvalho da proposta de Randy Bryant
com a adaptação para Java do Prof. Tarcisio Rocha

1 Visão Geral

O objetivo desse trabalho é desenvolver um sistema distribuído simples capaz de explorar o poder de vários processadores para acelerar o cumprimento de uma tarefa que exige computação intensiva. Em sua implementação, você irá incorporar recursos necessários para criar um sistema robusto, que é capaz de lidar com pacotes perdidos ou duplicados e com as possíveis falhas de clientes e servidores. Você também irá aprender o valor de criar um conjunto de abstrações em camadas na ligação entre protocolos de rede de baixo nível e aplicações de alto nível.

Seu sistema irá implementar um decifrador de senhas distribuído. A maioria dos sistemas minimiza os casos em que uma senha é armazenada ou transmitida de forma explícita, uma vez que uma violação menor de segurança permitiria que um invasor obtivesse uma cópia da senha e tivesse pleno acesso a muitos recursos do sistema. Em vez disso, uma função de *hash* segura h é aplicada à senha P para dar uma assinatura de *hash* $H = h(P)$. A função h é projetada de tal maneira que não possa ser facilmente invertida. Ou seja, saber o valor de H não fornece nenhuma informação real sobre o valor de P. Em vez de armazenar e distribuir senhas explicitamente, os sistemas reforçam a segurança de seu gerenciamento de senhas apenas armazenando suas assinaturas de *hash*.

Como exemplo, nos primeiros sistemas Unix, informações sobre todas as contas de usuário, incluindo as assinaturas *hash* de suas senhas, eram armazenadas em um arquivo não criptografado e desprotegido `/etc/passwd`. Quando um usuário fazia o login e informava a senha P, o sistema calculava $h(P)$ e verificava se este correspondia à assinatura armazenada no arquivo para aquele usuário. A segurança desta abordagem baseou-se na suposição de

que um atacante seria incapaz de determinar o valor de uma senha dada sua assinatura *hash*. Hoje em dia, os sistemas Unix armazenam as assinaturas de *hash* em locais menos acessíveis, porque se tornou prática a quebra de senhas usando a abordagem de força bruta, semelhante à qual você irá implementar.

Uma função *hash* segura comumente usada é a SHA-1, desenvolvida pela *National Security Agency* (Agência de Segurança Nacional). Ela gera uma assinatura *hash* de 20 *bytes* de uma sequência arbitrária de *bytes*. Aqui estão alguns exemplos da aplicação de SHA-1 em *strings* de texto:

P	$h(P)$
12344	420df50a0a436cabe48e1597a9508a2b5449d35e
12345	8cb2237d0679ca88db6464eac60da96345513964
12346	94ae0a96d83a445d72a93417b63ac90d79db5eca

Como estes exemplos ilustram, apesar da similaridade entre as *strings* de texto, seus valores de *hash* são muito diferentes.

Um ataque simples em um esquema que se baseia na dificuldade de inverter uma função de *hash* segura, seria executar uma busca de força bruta, na qual nós enumeramos possíveis senhas e vemos se o *hash* delas gera H. Nossas medições mostram que uma típica máquina de Andrew Linux pode calcular *hashes* SHA-1, a uma taxa de cerca de 10.000 por segundo. Executando em sequência, um decifrador de força bruta necessitaria de cerca de 28 horas para tentar todas as senhas possíveis compostas por 9 dígitos decimais. Mas, se pudéssemos aproveitar o poder de 100 máquinas, então poderíamos reduzir este tempo a cerca de 17 minutos.

A qualquer momento que seja necessário que uma aplicação execute em de dezenas ou centenas de máquinas, precisamos encontrar maneiras de tornar o sistema robusto. Inevitavelmente, algumas dessas máquinas vão parar de trabalhar, ou vamos encontrar casos em que máquinas são desconectadas. Portanto, escrever um decifrador de senhas distribuído te dá a oportunidade de aprimorar suas habilidades na concepção de sistemas distribuídos confiáveis.

O projeto é dividido em duas partes:

A: Implementar o *Live Sequence Protocol*, um protocolo próprio para fornecer comunicação confiável entre APIs (*Application Program Interfaces*) cliente e servidora simples em cima do protocolo UDP da Internet.

B: Implementar a aplicação decifradora de senhas. Você vai descobrir que a interface abstrata fornecida pelo LSP irá tornar o desenvolvimento dessa parte muito mais simples do que seria se você apenas estivesse utilizando código UDP diretamente.

2 Parte A: Simple and Reliable Protocol

O IP (Internet Protocol) de baixo nível proporciona o que é referido como um serviço de "datagrama não confiável", permitindo que uma máquina envie uma mensagem para outra como um pacote, mas com a possibilidade de que este pacote possa ser perdido ou duplicado. Além disso, como um pacote IP salta de um nó de rede para outro, o seu tamanho é limitado a um número máximo de *bytes*, conhecido como Unidade de Transmissão Máxima (*Maximum Transmission Unit*, MTU). Normalmente, os pacotes de até 1500 *bytes* podem seguramente ser transmitidos ao longo de qualquer caminho de roteamento, mas, um tamanho maior que esse se tornar problemático.

Poucas aplicações fazem uso do serviço de IP diretamente. Em vez disso, elas são escritas em termos de um dos seguintes protocolos:

UDP: É também um serviço de datagrama não confiável, mas permite que os pacotes possam ser direcionados para destinos lógicos diferentes numa única máquina, conhecidos como portas. Isto torna possível a execução de vários clientes ou servidores numa única máquina.

TCP: Um serviço de *streaming* de confiança, em que uma série de mensagens de tamanho arbitrário é transmitida através da quebra, na fonte, de cada mensagem em vários pacotes, onde depois estes são remontados no local de destino. TCP lida com questões como perda de pacotes, pacotes duplicados, e impede que o remetente sobrecarregue a largura de banda de Internet e as capacidades de *buffering* no destino.

O *Live Sequence Protocol* (LSP) oferece recursos que se encontram de certa forma entre o UDP e TCP, mas também tem características que não são encontradas em nenhum desses protocolos.

- Ao contrário do UDP ou TCP, o LSP é especializado para apoiar um modelo de comunicação cliente-servidor.
- O servidor pode manter conexões com vários clientes ao mesmo tempo, cada um dos quais é identificado por um código numérico único identificador de conexão.
- A comunicação entre o servidor e um cliente consiste de uma sequência de mensagens discretas em cada direção.
- O tamanho das mensagens é limitado para caber dentro de pacotes UDP individuais (cerca de 1000 *bytes*).
- As mensagens são enviadas de forma confiável: exatamente uma cópia de cada mensagem é recebida, e as mensagens são recebidas na mesma ordem em que foram enviadas.

- O servidor e os clientes monitoram o status de suas conexões e detectam quando o outro lado se desconecta.

2.1 LSP: Perspectiva da Rede

Iremos descrever inicialmente o LSP em termos de fluxo de mensagens entre o servidor e um dos seus clientes. Cada mensagem LSP contém três valores:

- ***Message Type***: Um código de 16 *bits* que descreve o tipo de mensagem. Vamos considerar apenas três tipos diferentes. Cada um tem um código numérico, mas vamos nos referir a estes por nomes:
  - ***Connect*** (**Código 0**): Enviado por um cliente para estabelecer uma conexão com o servidor.
  - ***Data*** (**Código 1**): Enviado por um cliente ou servidor para transmitir informações (dados).
  - ***Ack*** (**Código 2**): Enviado por um cliente ou servidor para reconhecer uma mensagem *Connect* ou *Data*.
- ***Connection ID***: Único, de 16 *bits* e diferente de zero, atribuído pelo servidor para identificar a conexão. Sua implementação pode escolher qualquer esquema para atribuir IDs. Nossa implementação simplesmente atribui IDs sequencialmente, começando com 1.
- ***Sequence Number***: Um número de 16 *bits*, que é incrementado com cada mensagem de dados enviada, começando do número 0 para a conexão inicial e, em seguida, incrementando módulo 65536.
- ***Payload***: Uma sequência de *bytes*, com um formato e interpretação determinados pela aplicação.

Usaremos a seguinte notação para representar os diferentes pacotes possíveis:

(Connect, 0, 0): Solicitação de conexão. Deve ter ID 0 e número de sequência 0.

(Data, id, sn, D): Mensagem de dados com o ID id, número de sequência sn, e *payload* D.

(Ack, id, sn): Mensagem de reconhecimento *Ack* com ID id, e número de sequência sn.

Ambas as mensagens *Connect* e *Ack* têm valores de *payload* `nil`.

A Figura 1 ilustra como a conexão é estabelecida. Nesta figura, as linhas verticais com setas para baixo denotam a passagem do tempo tanto no cliente como no servidor, enquanto que as linhas que atravessam horizontalmente denotam mensagens enviadas entre os dois. O cliente inicia a conexão enviando uma solicitação de conexão para o servidor. O servidor atribui um ID de conexão e responde ao cliente com uma mensagem de reconhecimento (Ack) contendo este ID, o número de sequência 0, e um *payload* vazio.

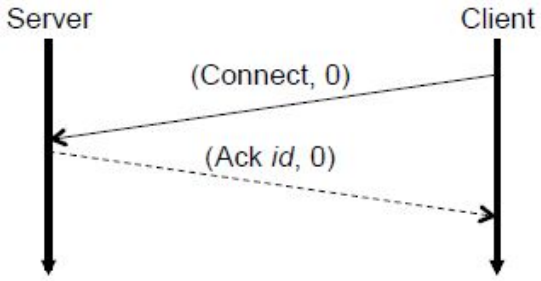

Figura 1: Estabelecendo conexão. O servidor atribui o ID de conexão e responde ao cliente com uma mensagem de reconhecimento (exibida como uma linha tracejada.)

A Figura 2 ilustra uma sequência normal de comunicação entre o servidor e um cliente. Mensagens de dados podem ser enviadas em ambas direções, e diferentes séries de números de sequência são mantidas para cada sentido, começando com um número de sequência 1. A figura ilustra a transmissão dos valores de dados D_i e D_{i+1}, tendo números de sequência i e $i+1$ a partir do cliente para o servidor. Estes são seguidos por um valor de transmissão de dados D_j, tendo número de sequência j, do servidor para o cliente.

Observe que cada mensagem de dados é seguida por uma mensagem de reconhecimento na direção oposta. Cada lado deve aguardar o reconhecimento ser recebido antes que possa enviar outra mensagem de dados. Note, no entanto, que é perfeitamente possível que um lado receba uma mensagem de dados a partir do outro lado enquanto aguarda o reconhecimento de uma mensagem de dados que já enviou - os dois lados operam de forma assíncrona, e o IP não garante que os pacotes chegam na mesma ordem em que são enviados.

Podemos ver que o protocolo básico ilustrado nas Figuras 1 e 2 não é completamente robusto. Por um lado, a presença de números de sequência torna possível detectar quando uma mensagem foi perdida ou duplicada. No entanto, se qualquer mensagem - seja de solicitação de conexão, de dados, ou de reconhecimento - for perdida, a conexão em uma ou em ambas as direções irá parar de funcionar, com ambos os lados à espera de mensagens do outro.

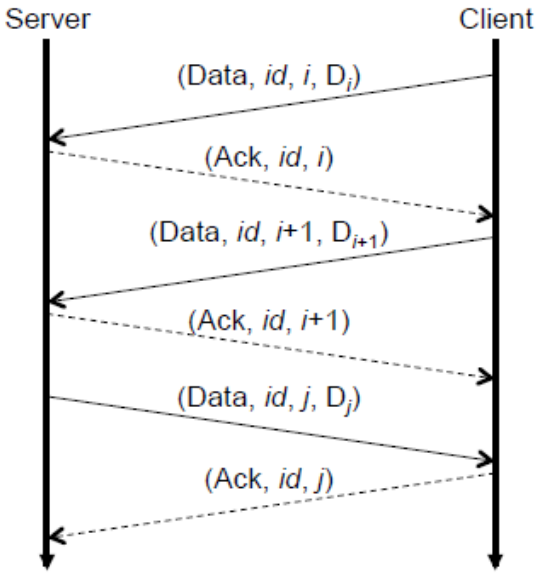

Figura 2: Comunicação normal. A mensagem número i deve ser reconhecida antes da mensagem $i + 1$ poder ser enviada. Mensagens de reconhecimento são exibidas como linhas tracejadas.

Para tornar o LSP robusto, incorporamos um simples gatilho de tempo para os servidores e clientes. Isto é, temporizadores disparam periodicamente nos clientes e servidores, dividindo-se o fluxo do tempo para cada processo em uma sequência de "épocas". Vamos nos referir a este intervalo de tempo como a "duração de época", a qual denominamos com o símbolo δ. Nosso valor padrão para δ é de dois segundos, embora possa ser variado.

Cada vez que o temporizador de época dispara, um cliente executa as seguintes ações:

- Reenvia o pedido de conexão, caso o pedido de conexão original ainda não tenha sido reconhecido.
- Se uma mensagem de dados foi enviada, mas ainda não foi reconhecida, então reenvia a mensagem de dados.
- Se qualquer das mensagens de dados foi recebida, então reenvia uma mensagem de reconhecimento para a mensagem de dados mais recentemente recebida.
- Se o pedido de conexão foi enviado e reconhecido, mas não foram recebidas mensagens de dados, então envia um reconhecimento (Ack) com número de sequência 0.

O servidor executa um conjunto semelhante de ações para cada uma de suas conexões:

- Se uma mensagem de dados foi enviada, mas ainda não reconhecida, então reenvia a mensagem de dados.
- Reconhece a solicitação de conexão, se nenhuma mensagem de dados foi recebida.
- Reconhece a mensagem de dados recebida mais recentemente, se houver.

A Figura 3 ilustra a forma como os eventos de época compensam as falhas da comunicação normal. Nós mostramos a ocorrência do temporizador de época como uma forma oval grande e preta em cada linha de tempo. Neste exemplo, o cliente tenta enviar os dados D_i, mas a mensagem de confirmação acaba sendo perdida. Além disso, o servidor tenta enviar os dados D_j, mas a mensagem de dados acaba sendo perdida. Quando o temporizador de época dispara no cliente, ele irá enviar um reconhecimento da mensagem de dados $j-1$, a última mensagem de dados recebida, e irá reenviar os dados D_i.

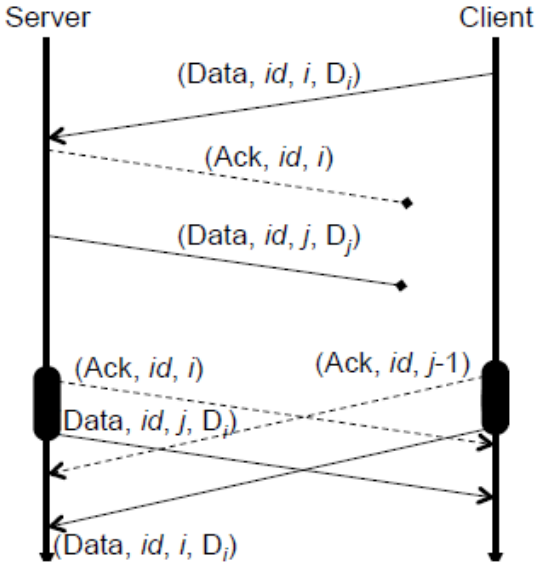

Figura 3: Atividades de Época. Ambos os lados enviam reconhecimentos (mostrados como linhas tracejadas) para a mensagem de dados mais recentemente recebida e (possivelmente) reenviam qualquer mensagem de dados não reconhecida.

Assumindo que o servidor tem um evento de época por volta do mesmo tempo (não há nenhuma exigência de que eles ocorrem simultaneamente), podemos perceber que o servidor irá enviar um reconhecimento para os dados D_i, e irá reenviar os dados D_j.

Podemos ver neste exemplo que mensagens duplicadas podem ocorrer: por exemplo, o servidor recebeu duas cópias de D_i. Na maioria dos casos, podemos usar números de sequência para detectar eventuais duplicações. Isto é, cada lado mantém um contador que indica que número de sequência é esperado e descarta qualquer mensagem que não corresponde ao número esperado. No entanto, um caso de duplicação requer uma atenção especial. É possível que o cliente envie vários pedidos de conexão, com um ou mais pedidos ou reconhecimentos sendo perdidos. O servidor deve rastrear o endereço do *host* e o número da porta de cada pedido de conexão e descartar aqueles para o qual essa combinação de *host* e porta já tenha uma conexão estabelecida.

Uma característica do trabalho com épocas é que haverá pelo menos uma mensagem transmitida em cada direção entre um cliente e um servidor em cada época. Como um recurso final, iremos rastrear na extremidade de cada conexão o número de épocas que se passaram desde que uma mensagem (de qualquer tipo) foi recebida na outra extremidade. Uma vez que essa contagem atingir um limite especificado de épocas, o qual denotamos com o símbolo K, vamos declarar que a conexão foi perdida e notificar à aplicação que ela perdeu essa conexão. Nossa implementação utiliza um valor padrão de 5 para K. Assim, se nada for recebido a partir da outra extremidade ao longo de um período total de $K \cdot \delta$, tendo um valor padrão de 10 segundos, então a conexão será encerrada.

2.2 LSP: Perspectiva da API

Vamos agora fornecer uma vista do LSP a partir da perspectiva de um programador da linguagem *Java*. Precisamos que você implemente exatamente a mesma API mostrada aqui para facilitar os testes automatizados, e para garantir a compatibilidade entre as diferentes implementações do protocolo.

2.2.1 Parâmetros LSP

Tanto para o cliente como para o servidor, a API fornece um mecanismo para especificar o limite de épocas K e a duração de época δ, quando um cliente ou servidor é inicialmente criado. Estes parâmetros são encapsulados na seguinte classe:

```
1 public class LspParams {
```

```
2     private int epoch;
3     private int epochLimit;
4
5     public LspParams(int epoch, int epochLimit){
6       this.epoch = epoch;
7       this.epochLimit = epochLimit;
8     }
9
10    public int getEpoch() {
11       return epoch;
12    }
13
14    public int getEpochLimit() {
15       return epochLimit;
16    }
17
18 }
```

2.2.2 API do Cliente

Uma aplicação instancia a classe `LspClient` para configurar e iniciar as atividades de um cliente LSP. O construtor dessa classe é declarado da seguinte forma:

```
1 LspClient(String host, int port, LspParams params)
```

onde `host` indica o nome ou ip do servidor, `host` indica o número da porta do servidor e `params` é uma referência a um objeto `LspParams` que contém os parâmetros LSP. Quando esta referência é `null`, o cliente deve usar os valores padrão para estes parâmetros. Este construtor deve retornar depois que uma conexão com o servidor tenha sido estabelecida, por exemplo, que o cliente tenha recebido uma mensagem de *Ack* em resposta ao seu pedido de conexão. Ela deve retornar uma exceção caso a conexão não possa ser estabelecida.

A aplicação cliente simplesmente lê e escreve mensagens de dados. Ela também pode encerrar uma conexão, fazendo com que o cliente encerre a transmissão ou o recebimento de mensagens. Todos os detalhes do estabelecimento de uma conexão, reconhecimento de mensagens e manipulação de épocas são ocultados do programador da aplicação. A função a seguir define esta interface:

```
1 /**
2 * Devolve o Id da conexão
3 */
4 public int getConnId();
5
```

```
6  /**
7  * Devolve um vetor de bytes de uma mensagem enviada pelo lado servidor.
8  * Devolve null se a conexão for perdida.
9  */
10 public byte[] read();
11
12 /**
13 * Envia uma mensagen para o lado servidor como um vetor de bytes.
14 * Devolve exceção se a conexão for perdida.
15 */
16 public void write(byte[] payload);
17
18 /**
19 * Encerra a conexão.
20 */
21 public void close();
```

Como os comentários indicam, caso a conexão com o servidor tenha sido perdida, a operação `read` devedevolver `null` e a operação `write` deve devolver uma exceção. Alguns detalhes sobre essas três chamadas devem ser observados:

- `read` operação bloqueante – deve retornar quando os dados forem recebidos com sucesso a partir do servidor ou quando a conexão com o servidor for perdida.
- `write` operação não bloqueante – deve retornar imediatamente, ou seja não bloqueia a linha de execução do chamador até que a mensagem chegue. Deve lançar exceção caso a conexão tenha sido perdida.
- `close` deve retornar depois que qualquer mensagem pendente para o servidor tenha sido enviada e reconhecida, ou se a conexão com o servidor for perdida.
- O cliente não deve tentar chamar `read`, `write` ou `close` depois de chamar `close`.
- Uma vez que a chamada a `close` tenha sido completada, quaisquer rotinas *Java* associadas com o cliente devem encerrar.

2.2.3 API Servidor

A API para o servidor é semelhante à de um cliente, exceto pelo fato de que cada mensagem é marcada pelo seu ID de conexão. O servidor é configurado e iniciado instanciando a classe `LspServer`, através do seguinte construtor:

```
1 public LspServer(int port, LspParams params)
```

Seus argumentos são o número da porta que o servidor deve usar e os parâmetros LSP. Quando `params` é `null`, o servidor deve usar os valores padrão para estes parâmetros.

A aplicação tipo servidor também pode ler e escrever mensagens, além de encerrar conexões. Como um servidor pode estar conectado a vários clientes, a diferenciação entre esses clientes se dá pelo ID de conexão.

```
1  /**
2  * Lê dados da fila de entrada do servidor. Se não houver dados
3  * recebidos, bloqueia o chamador
4  * até que dados sejam recebidos. Os dados estão encapsulados
5  * pela classe Pack.
6  */
7  public Pack read();
8
9  /**
10 * Envia dados para um determinado cliente.
11 * Deve Devolver exceção se a conexão estiver encerrada.
12 */
13 public void write(Pack pack);
14
15 /**
16 * Encerra uma conexão com o identificador connId.
17 */
18 public void closeConn(int connId);
```

Nas operações de leitura e escrita, esse ID está encapsulado na classe `Pack`. Como pode ser visto a seguir, essa classe encapsula, além do vetor de bytes que está sendo lido ou escrito, o ID da conexão de forma a diferenciar o cliente envolvido:

```
1  public class Pack {
2     protected short connId;
3        protected byte[] payload;
4
5        public Pack(short connId, byte[] payload) {
6           this.connId = connId;
7           this. payload = payload;
8        }
9
10       public short getConnId() {
11          return connId;
12       }
13
14       public void setConnId(short connId) {
15          this.connId = connId;
16       }
17
18       public byte[] getPayload() {
```

```
19              return payload;
20          }
21  }
```

A operação `read` retorna um `Pack` que encapsula: o ID de conexão indicando o cliente a partir do qual a mensagem foi recebida, o *payload* da mensagem. A operação `write` recebe um `Pack` como parâmetro que encapsula o ID da conexão do cliente para o qual a mensagem deve ser enviada e o `playload` da mensagem. A operação `closeConn` inclui um ID de conexão como um argumento, indicando qual conexão deve ser encerrada.

Encerrar uma conexão indica que o servidor deve parar de aceitar (ou reconhecer) mensagens para esta conexão. Podem ainda haver mensagens para esta conexão que foram escritas pela aplicação, mas que ainda não foram enviadas para o cliente. A aplicação deve continuar enviando (e reenviando) mensagens até que 1) todas tenham sido enviadas e reconhecidas, ou 2) a conexão com o cliente tenha sido perdida.

Além disso, uma chamada final permite que o servidor encerre todas as conexões ativas:

```
1  public void closeAll();
```

Alguns detalhes sobre a API do servidor devem ser observados:

- `read` deve retornar quando os dados foram recebidos de um cliente com sucesso.
- `write` deve retornar imediatamente.
- `closeConn` deve retornar depois que qualquer mensagem pendente para o cliente tenha sido enviada e reconhecida ou a conexão com o cliente tenha sido perdida.
- `closeAll` deve retornar depois que, para cada cliente, qualquer mensagem pendente para o cliente tenha sido enviada e reconhecida ou a conexão com o cliente tenha sido perdida.
- O servidor não deve tentar chamar `write` ou `closeConn` para uma conexão depois de chamar `closeConn` para esta conexão.
- Deve ser possível chamar `closeAll` mesmo quando não há conexões ativas sem pendência.
- O servidor não deve tentar chamar qualquer operação depois de chamar `closeAll`.
- Uma vez que a chamada a `closeAll` tenha sido completada, quaisquer rotinas *Java* associadas com o servidor devem encerrar.

2.3 Exemplo de Aplicação LSP

```
1  /**
2   * Cliente de echo simples. Envia uma mensagem ao servidor,
3   * e obtém com resposta do mesmo
4   * a mesma mensagem que foi enviada.
5   * @author tarcisio.rocha
6   */
7  public class ClientEcho {
8     public static void main(String[] args){
9        // Cliente se conecta com um servidor
10       LspClient client = new LspClient("localhost", 4455,
11                                        new LspParams(5000, 5));
12       // Define mensagem a enviar
13       String mensagem = "Olá!!";
14       // Envia mensagem
15       client.write(mensagem.getBytes());
16       // Recebe resposta do servidor de echo
17       byte [] payload = client.read();
18       // Converte a resposta de bytes para string
19       String recebido = new String(payload);
20       // Exibe resposta
21       System.out.println(recebido);
22    }
23 }
```

Figura 4: *Echo* do Cliente baseado em LSP.

Como uma ilustração de como as aplicações podem ser escritas para usar LSP, mostramos as funções essenciais para um simples *echo* de cliente e servidor.

O código do cliente de *echo* é mostrado na Figura 4. Uma conexão com um servidor que se encontra na máquina local ("localhost") escutando a porta 4455 é criada. Em seguida é criada uma mensagem é chamada a operação `write` para enviar a mensagem para o servidor e `read` para receber de volta a mensagem do mesmo servidor em *bytes*. Por fim, a mensagem é convertida de `bytes` para `String` e exibida na tela.

O código do servidor de *echo* é mostrado na Figura 5. O lado servidor é instanciado e fica apto a receber conexões. Dentro de seu *loop*, procede-se com a leitura das mensagens recebidas na forma de um `Pack` através da operação `read`. Cada `Pack` encapsula os bytes da mensagem e o ID da conexão do cliente que a enviou. Em seguida, o conteúdo do `Pack` é exibido (ID da conexão + mensagem) e a mesma mensagem recebida é enviada para o mesmo cliente através da operação `write`.

```java
/**
 * Classe que implementa um servidor de echo simples.
 * Continuamente recebe pacotes
 * de clientes, exibe os dados na tela no formato
 * String e devolve ao cliente o
 * mesmo pacote que ele enviou.
 * @author tarcisio.rocha
 */
public class ServerEcho {
   public static void main(String[] args){
      // Servidor fica apto a receber conexões na porta 4455
      LspServer server =  new LspServer(4455, new LspParams(5000, 5));
      while (true){
         //Lê próxima mensagem
         Pack p = server.read();
         //Exibe conteúdo da mensagem
         System.out.println("ConnID: "+p.connId +
                            " Data: "+ new String(p.getPayload()));
         //Devolve a mesma mensagem recebida ao cliente
         server.write(p);
      }
   }
}
```

Figura 5: *Echo* do Servidor baseado em LSP.

2.4 Dicas

- Você é livre para formular seu próprio projeto, porém uma abordagem é visualizar a implementação do lado cliente (que pode ser uma classe chamada de `LspClient`) consistindo de pelo menos três linhas de execução independentes:

  Manipulador de pacotes de entrada: Lê os pacotes que estão chegando da conexão UDP, encaminhando cada pacote para ser tratado de acordo com o seu tipo (ex: ACK, DATA...).

  Disparador de épocas: A cada δ unidades de tempo, faz as checagens e toma as medidas necessárias para a época.

  Manipulador de envio de pacotes: Verifica de forma cíclica se a aplicação cliente solicitou o envio de algum pacote. Caso positivo, envia o pacote (caso o pacote anteriormente enviado já tenha sido reconhecido com um ACK – lembrar que um novo pacote só é enviando quando o anterior já tiver sido reconhecido).

  Cada um desses manipuladores é executado como uma thread *Java*,

coordenando e se comunicando com os outras através de referências e estruturas de dados globais compartilhadas. Existem duas estruturas de dados mantidas pelo manipulador de eventos: uma para manter as mensagens que foram escritas pela aplicação, mas ainda não foram enviadas através da rede, e outra que contém as mensagens que tenham sido recebidas a partir da rede, mas ainda não foram lidas pela aplicação. Uma classe Java que pode ser usada para representar essas duas estruturas é a `LinkedBlockingQueue`.

O servidor pode ser implementado por pelo menos duas classes principais. Uma chamada `LspServer` que consiste na rotina de execução principal do servidor e outra chamada `LspConnection` que representa uma conexão com um dado cliente. Um dos principais papeis da rotina principal seria o de receber os pacotes que estão chegando e, caso seja um pacote de pedido de conexão, criar uma nova instância do LspConnection para tratá-la criando para ela um novo ID. Caso esse pacote seja um pacote de dados ou de ACK, encaminhar o pacote para a respectiva instância de `LspConnection` para que o mesmo seja tratado baseando-se no ID da conexão que cada pacote deve carregar. Uma dica para fazer com que o `LspServer` consiga manter todas as suas instâncias de `LspConnection` ao mesmo tempo é o uso de uma tabela hash onde cada instância de `LspConnection` vai poder ser identificada com base no ID da conexão. Uma estrutura Java que pode ser usada para isso é a `HashMap`. Pode-se instanciar uma `HashMap` no `LspServer` com a forma

```
1 HashMap<Short, LspConnection> tabelaDeConexoes =
2                 new HashMap<Short, LspConnection>()
```

tendo como chave o ID da conexão como um tipo Java `short` e o valor como uma instância do tipo `LspConnection`. Com isso, pode-se (i) adicionar o par <chave, valor> na tabela através do método `put` e (ii) obter uma `LspConnection` (ou seja, um valor da tabela) que corresponde a uma chave (ID da conexão) através da operação `get` da tabela. Você deve consultar a documentação Java para mais detalhes de como usar um `HashMap`.

A implementação da `LspConnection` precisará ter, pelo menos duas linhas de execução independentes o "Disparador de épocas" e o "Manipulador de envio de pacotes" que toma, no lado servidor, medidas semelhantes as que toma o `LspClient` no lado cliente. Ele só não possuiria uma linha para o "Manipulador de pacotes de entrada", já que quem faz esse papel é o `LspServer`. O `LspServer` pode entregar os

pacotes que chegam para a respectiva `LspConnection` através da chamada de um método da `LspConnection` que deve ser implementado.

3 Parte B: Decifrador de Senhas

(A descrição do decifrador de senhas que usa o LSP será disponibilizada em breve)