PCD2018 - 20

ESERCIZIO 2

DIFFIE–HELLMAN KEY EXCHANGE

- Alice e Bob pubblicano p, g
- Alice sceglie a, ed invia a Bob $A = g^a \bmod p$
- Bob sceglie b, ed invia ad Alice $B = g^b \bmod p$
- Ora condividono $s = B^a \bmod p = A^b \bmod p$

Speaker notes

Per essere efficace, p è un numero primo. g deve essere una radice primitiva di p. Dettagli su https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange.

Quindi, durante uno scambio fra Alice e Bob sono pubblici:

$$p, g, A, B$$

Mentre sono di difficile calcolo, e segreti:

$$a, b, s$$

Scopo dell'esercizio è impersonare un'agente malevolo, Charlie, che intende recuperare i segreti di Alice e Bob con un attacco di forza bruta.

Speaker notes

intende cioè provare tutte le combinazioni di a e b finché non trova quelle che verificano l'equazione di s, in modo da ottenere le chiavi private di Alice e Bob.

Ai fini di completare l'esercizio prima della morte termica dell'universo, i numeri (e quindi la sicurezza) dello scambio sono molto, molto bassi.

Speaker notes

una chiave di livello normale ha solitamente almeno 1024 bit. Il nostro esempio ne ha 24.

La classe da realizzare deve rendere verde un test che, dati i seguenti parametri:

$$p=128504093,\ g=10009$$

$$A=69148740,\ B=67540095$$

verifica che tutte le coppie prodotte dalla funzione da completare soddisfino la condizione di calcolo di s.

Speaker notes

dato il basso numero di bit, sono possibili collisioni; la soluzione è molto probabilmente non unica. Ma questo è un problema di Charlie.

Charlie è inoltre a conoscenza che, imprudentemente, Alice e Bob hanno scelto:

$$0 <= a <= 65536$$

$$0 <= b <= 65536$$

Le coppie da verificare quindi sono solo 2^32.

Speaker notes

c'è un modo abbastanza veloce di controllare tutte le coppie, che non dovrebbe essere difficile da individuare.

OBIETTIVO DELL'ESERCIZIO:

organizzare il lavoro di attraversamento dello spazio di ricerca in modo da utilizzare più thread contemporaneamente, e occupare tutti i core disponibili.

La consegna dell'esercizio consiste in una mail a `michele.mauro@unipd.it` con:

- oggetto: [pcd18] [123456] Esercizio 2
- allegato: DiffieHellman_123456.java
- testo: spiegazione dell'organizzazione del codice e della suddivisione del lavoro scelta

Va sostituito `123456` con la propria matricola.

Speaker notes

la mancanza o l'inesattezza di uno di questi elementi può penalizzare o invalidare la consegna.

La parte più importante è l'organizzazione del codice che suddivide il lavoro.

Criteri di valutazione saranno:

- facilità di lettura
- chiarezza della struttura
- scelta delle primitive corrette

Speaker notes

la correttezza del risultato è secondaria. Copiare è inutile, anche perché abbiamo notato che tendenzialmente, le soluzioni più copiate sono spesso quelle sbagliate.

pcd2018.exe2.DiffieHellman

```
public class DiffieHellman {

  /**
   * Limite massimo dei valori segreti da cercare
   */
  private static final int LIMIT = 65536;

  public List<Integer> crack(long publicA, long publicB) {
    List<Integer> res = new ArrayList<Integer>();

    return res;
  }
}
```

CHANNELS

Per completare la panoramica sui metodi di gestione dei `Socket` ci manca un'astrazione che ci permetta di ascoltare e reagire a più richieste di connessione.

Nella revisione dei metodi di I/O introdotta con il package `java.nio` in Java 1.4 nel 2002, viene introdotto un'intero albero di tipi dedicati alla gestione della comunicazione nel modo più generico.

```
/**
 * A nexus for I/O operations.
 *
 **/
public interface Channel extends Closeable
```

Un `Channel` rappresenta un canale di I/O, che può essere aperto o chiuso.

```
/**
 * A channel to a network socket.
 *
 **/
public interface NetworkChannel extends Closeable
```

Un `NetworkChannel` rappresenta una comunicazione su di una rete. Può:

- esssere legato (con l'operazione `bind`) ad un'indirizzo
- dichiarare le opzioni che supporta.

```
/**
 * An asynchronous channel for stream-oriented
 * listening sockets.
 *
 **/
public abstract class AsynchronousServerSocketChannel
  implements AsynchronousChannel, NetworkChannel
```

Un `AsynchronousServerSocketChannel` è un canale asincrono basato su di una server socket.

Ci permette, in modo asincrono, di accettare connessioni e gestirle.

```
/**
 * A handler for consuming the result of an asynchronous
 * I/O operation.
 *
 * @param V The result type of the I/O operation
 * @param A The type of the object attached to the
 *   I/O operation
 **/
interface CompletionHandler<V,A>
```

Speaker notes

abbiamo bisogno di specificare anche questa interfaccia. Essendo la modalità asincrona, questa interfaccia ci permette di indicare al sistema l'azione da effettuare al completamento della successiva interazione.

`CompletionHandler` è l'interfaccia che deve implementare un oggetto che gestisce la ricezione di un'operazione di I/O asincrona.

```
/**
 * Invoked when an operation has completed.
 *
 * @param result The result of the I/O operation
 * @param attachment The type of the object attached to
 *   the I/O operation when it was initiated
 **/
void completed(V result, A attachment)
```

Il compito del metodo `completed` è gestire l'interazione relativa ai dati ricevuti, ed eventualmente predisporre l'operazione successiva.

```
/**
 * Invoked when an operation fails.
 *
 * @param exc The exception to indicate why the I/O
 *   operation failed
 * @param attachment The type of the object attached to
 *   the I/O operation when it was initiated
 **/
void failed(Throwable exc, A attachment)
```

Il compito del metodo `failed` è, ovviamente, gestire il caso in cui un'interazione ha incontrato una eccezione.

Implementando un `CompletionHandler` possiamo esprimere il comportamento del server alla prossima interazione, in maniera asincrona.

Speaker notes

asincrona nel senso non sincronizzata con il nostro codice. Indichiamo al sistema quale codice eseguire quando si completa un certo evento.

Il parametro generico `attachment` ci permette di far circolare le informazioni di contesto riguardo allo stato della conversazione.

I vari handler potrebbero essere chiamati da `Thread` diversi, in momenti imprevedibili; da qui la necessità di gestire esplicitamente il passaggio del contesto.

Speaker notes

perché le varie linee di esecuzione non avrebbero altrimenti modo di scambiarsi dati correttamente.

La gestione delle operazioni di I/O richiede quindi di specificare sempre l'attachment da far circolare ed il `CompletitionHandler` che gestisce il completamento.

```
/**
 * (from AsynchronousServerSocketChannel)
 * Accepts a connection.
 *
 * @param A The type of the attachment
 * @param attachment The object to attach to the I/O
 *   operation; can be null
 * @param handler The handler for consuming the result
 **/
public abstract <A> void accept(A attachment,
  CompletionHandler<AsynchronousSocketChannel,? super A>
  handler)
```

```
/**
 * (from AsynchronousSocketChannel)
 * Reads a sequence of bytes from this channel into the given
 * buffer.
 *
 * @param A The type of the attachment
 * @param dst The buffer into which bytes are to be
 *   transferred
 * @param attachment The object to attach to the I/O op.
 * @param handler The completion handler
 **/
public final <A> void read(ByteBuffer dst, A attachment,
  CompletionHandler<Integer,? super A> handler)
```

```
/**
 * (from AsynchronousSocketChannel)
 * Writes a sequence of bytes to this channel from the given
 * buffer.
 *
 * @param A The type of the attachment
 * @param src The buffer from which bytes are to be
 *   retrieved
 * @param attachment The object to attach to the I/O op.
 * @param handler The completion handler object
 **/
public final <A> void write(ByteBuffer src, A attachment,
  CompletionHandler<Integer,? super A> handler)
```

Questo richiede di riorganizzare (pesantemente) il nostro codice, ma ci permette di gestire molte più connessioni.

Un'alternativa all'uso di un `CompletionHandler` è data dalla versione dei metodi che ritorna un `Future`.

```
/**
 * (from AsynchronousServerSocketChannel)
 * Accepts a connection.
 *
 * @return a Future object representing the pending result
 **/
public abstract Future<AsynchronousSocketChannel>
  accept()
```

```
/**
 * (from AsynchronousSocketChannel)
 * Reads a sequence of bytes from this channel into the given
 * buffer.
 *
 * @param dst The buffer into which bytes are to be
 *   transferred
 * @return A Future representing the result of the operation
 **/
public abstract Future<Integer> read(ByteBuffer dst)
```

```java
/**
 * (from AsynchronousSocketChannel)
 * Writes a sequence of bytes to this channel from the given
 * buffer.
 *
 * @param src The buffer from which bytes are to be
 *   retrieved
 * @return A Future representing the result of the operation
 **/
public abstract Future<Integer> write(ByteBuffer src)
```

La struttura a cui questo approccio porta è duale alla precedente: il contesto è dato dal blocco in cui viene eseguito gestito il `Future`.

La principale differenza è che in questo caso, se il blocco di codice è unico per tutta la conversazione, il thread che la gestisce è unico e rimane allocato per l'intera durata della conversazione.

ESEMPIO

pcd2018.channels.Server

```
ExecutorService pool = Executors.newFixedThreadPool(4);
AsynchronousChannelGroup group=
  AsynchronousChannelGroup.withThreadPool(pool);
AsynchronousServerSocketChannel serverSocket =
  AsynchronousServerSocketChannel.open()
  .bind(new InetSocketAddress("127.0.0.1", GAME_PORT), 16);

pool.submit(() -> {
  serverSocket.accept(
    new GameAttachment(1, new Game(), serverSocket, group),
    new AcceptPlayerO());
});
```

Speaker notes

Il main prepara le risorse e istanzia l'attachment vuoto per iniziare alla ricezione della prima connessione. La prossima operazione è `AcceptPlayerO`. Notate che possiamo scegliere noi il tipo di `ExecutorService` che il sistema deve usare.

pcd2018.channels.AcceptPlayerO

```
@Override
public void completed(AsynchronousSocketChannel result,
  GameAttachment attachment) {
  System.out.println(Thread.currentThread().getName() +
    " : game " + attachment.id + " connected player O");
  attachment.server.accept(attachment.playerO(result),
    new WriteFirstStatus());
}
```

Speaker notes

`AcceptPlayerO` viene richiamata alla ricezione della prima connessione. Annotiamo il socket collegato nell'attachment, e programmiamo per la prossima azione `WriteFirstStatus`

pcd2018.channels.WriteFirstStatus

```
public void completed(AsynchronousSocketChannel result,
  GameAttachment attachment) {

  attachment = attachment.playerX(result);
  GameResult status = attachment.game.status();
  AsynchronousSocketChannel socket =
    attachment.players[status.next];
  byte[] bytes = (status.toString() + "\n").getBytes();
  socket.write(wrap(bytes), attachment, new ReadPlayer());
```

Speaker notes

Annotiamo la seconda connessione, inviamo lo stato sulla prima e leggiamo la mossa del primo giocatore.

```
  // more games?
  if (attachment.id <= 5) {
    attachment.server.accept(new GameAttachment(attachment.id
      new Game(), attachment.server), new AcceptPlayerO());
  } else {
    attachment.group.shutdown();
  }
}
```

Speaker notes

se abbiamo raggiunto il numero desiderato di partite, segnaliamo al gruppo del canale di cominciare a considerare la chiusura del sistema. Altrimenti predisponiamo, alla ricezione di una nuova connessione, l'apertura di una nuova partita.

pcd2018.channels.ReadPlayer

```
public void completed(Integer result,
    GameAttachment attachment) {
  GameResult status = attachment.game.status();
  AsynchronousSocketChannel socket =
    attachment.players[status.next];
  attachment.readBuf.clear();
  socket.read(attachment.readBuf, attachment,
    new WriteStatus());
}
```

Speaker notes

ci mettiamo in attesa della mossa dalla connessione del giocatore che deve muovere, e programmiamo come risposta l'invio dello stato all'altro giocatore.

pcd2018.channels.WriteStatus

```
String input = new String(attachment.readBuf.array(),
  0, result).trim();
Integer move = Integer.parseInt(input);
GameResult initial = attachment.game.status();
GameResult status =
  attachment.game.move(initial.next, move);
```

Speaker notes

leggiamo i dati di input e calcoliamo lo stato dopo la mossa ricevuta.

```
if (!status.end) {
  // the game goes on
  AsynchronousSocketChannel socket =
    attachment.players[status.next];
  byte[] bytes = (status.toString() + "\n").getBytes();
  socket.write(wrap(bytes), attachment, new ReadPlayer());
```

Speaker notes

se la partita non è terminata ancora terminata, prepariamo il messaggio per il prossimo giocatore, e impostiamo la prossima azione su `ReadPlayer`.

```
} else if (status.valid) {
  attachment.players[status.next].write(
    wrap("You won.".getBytes()), attachment,
    new CloseSocket(status.next));
  int loser = (status.next + 1) & 0x1;
  attachment.players[loser].write(
    wrap("You lost.".getBytes()), attachment,
    new CloseSocket(loser));
```

Speaker notes

oppure, se abbiamo un vincitore, predisponiamo entrambi i messaggi (il sistema li manderà quando avrà un thread a disposizione) e programmiamo la chiusura dei socket una volta completato l'invio

```
} else {
  // we have a tie
  attachment.players[0].write(
    wrap("Tied.".getBytes()), attachment,
    new CloseSocket(0));
  attachment.players[1].write(
    wrap("Tied.".getBytes()), attachment,
    new CloseSocket(1));
}
```

Speaker notes

in modo simile, se abbiamo una parità.

pcd2018.channels.CloseSocket

```
try {
  attachment.players[idx].close();
} catch (IOException e) {
  e.printStackTrace();
}
```