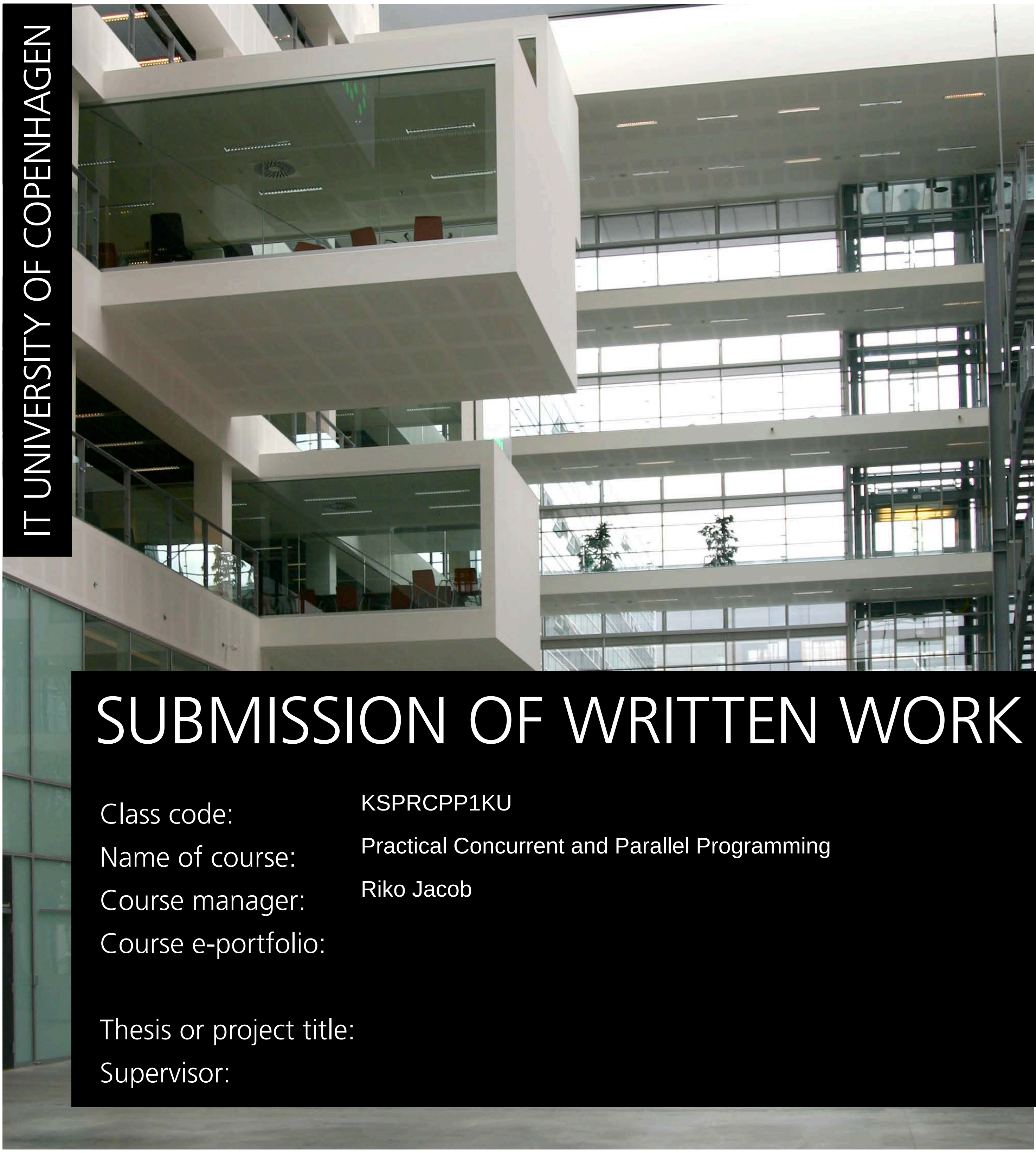
IT UNIVERSITY OF COPENHAGEN



SUBMISSION OF WRITTEN WORK

Class code:	KSPRCPP1KU
Name of course:	Practical Concurrent and Parallel Programming
Course manager:	Riko Jacob
Course e-portfolio:	
Thesis or project title:	
Supervisor:	

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Richard Banyi	03.06.1993	riba@itu.dk
2.			@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

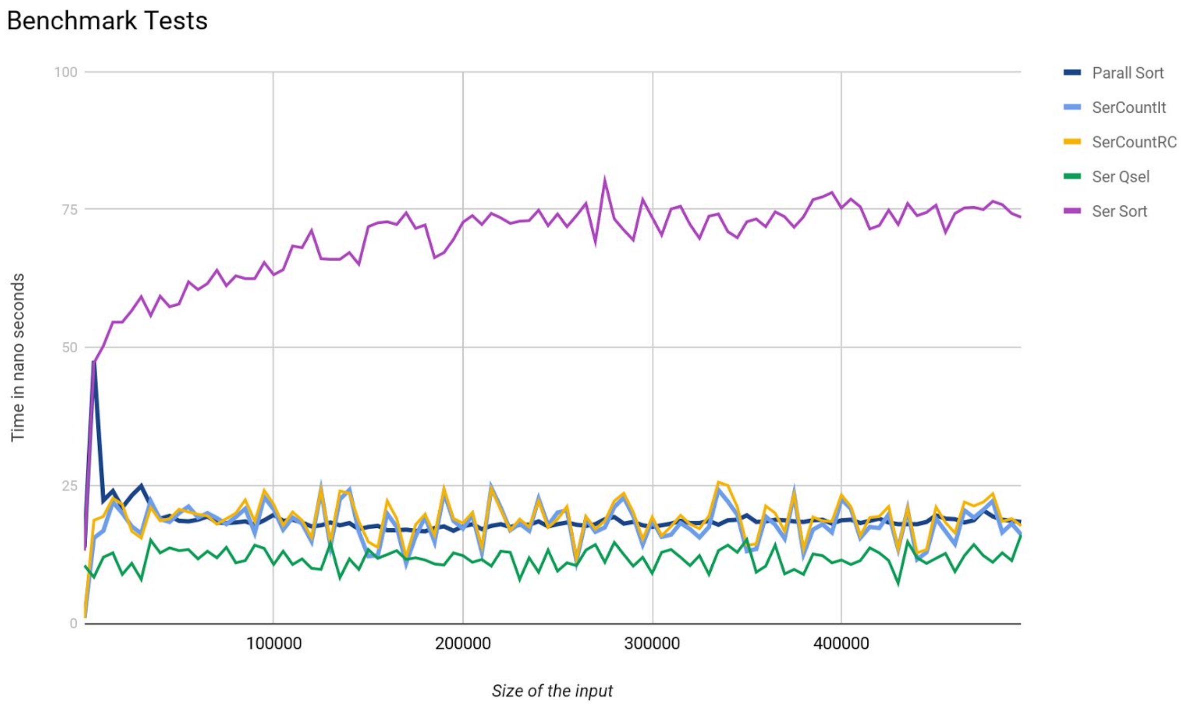
EXAM PCPP 2017 12th of December
Richard Banyi
riba@itu.dk

Machine

# OS: Mac OS X; 10.13.1; x86_64
# JVM: Oracle Corporation; 1.8.0_144
# CPU: null; 8 "cores"
# Date: 2017-12-12T07:17:15+0100

Q1

What I have witnessed is that the Ser Sort performs that the worse from all of the other algorithms. The execution time is best for the SerCountRC, SerCountIt where the size of the throughput does not really affects the execution time.



Q4

Q5

In order to make the ConcurrentStack thread-safe I have used global lock object which is used to lock when calling push or pop, because both of these methods mutate.

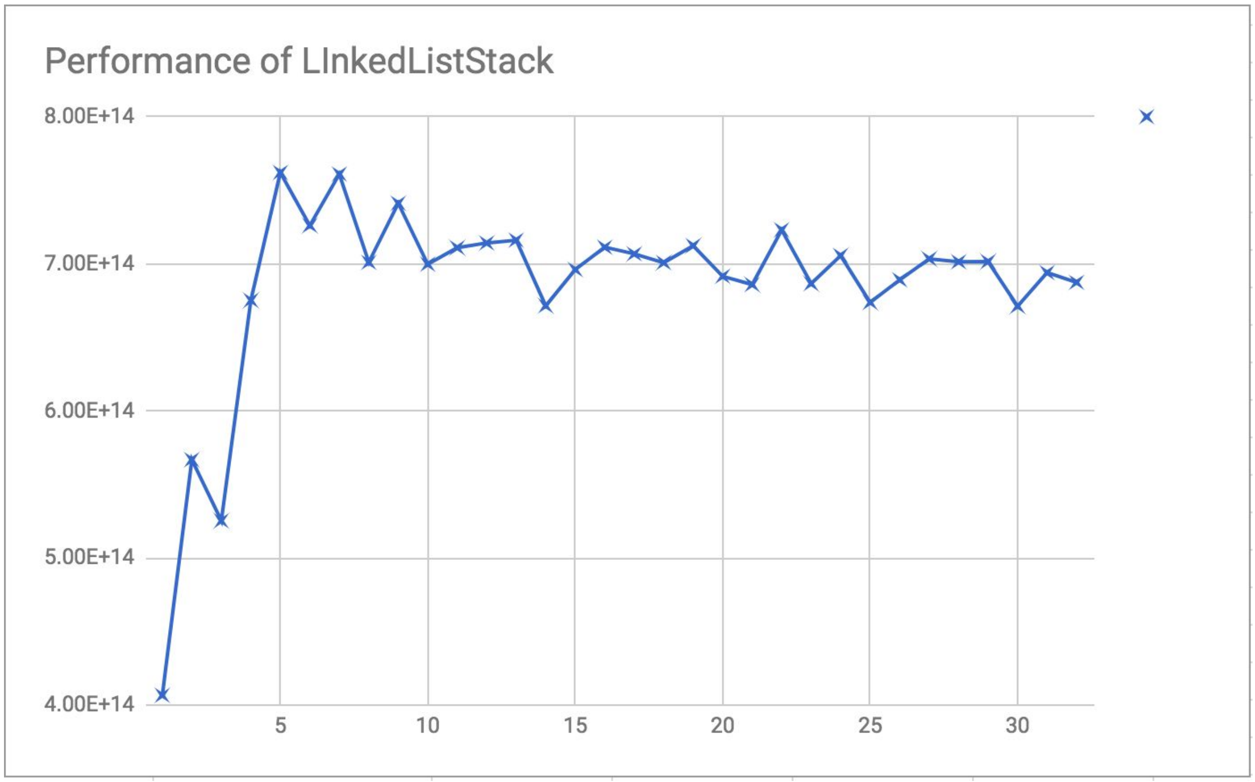
For the correctness of the functional requirements I have created a very
basic single-threaded functional testing while implementing the stack based
linked list, the method name is called *seqTest*.

Also I have creating a *parallelTest* with N-threads with aggregate results.
For this test I have used the **CyclicBarrier(N)** and used the Consumer & Producer
pattern. First I create *npair* of Producers and Consumers. The Producers
pushes *nTrials* random numbers and the consumer pops *nTrials* from the stack.
Afterwards I check the consumed numbers is equals to the sum of the produced
numbers. Both of them are summing a thread-local *sum* variable and than adding
the result to a common AtomicInteger.

I have also tried to implement the lock striping version, but for some reason I have encountered all the time null pointer exceptions, therefore I could run the performance test. See the class called class StripedStack for more information.

Results:

```
Parallel test: class ConcurrentStackImp... passed
```



Y - time in nano seconds
X - number of threads

Code:

```java
import java.util.LinkedList;
import java.util.Random;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.Semaphore;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.function.IntToDoubleFunction;


public class ConcurrentStack {
  public static void main(String[] args) {
    System.out.printf("STACK");
    System.out.println();
    seqTest(new ConcurrentStackImp());
    parallelTest(new ConcurrentStackImp());
    // timeAllMaps();
    //seqStripedTest(new StripedStack(32));
  }

  private static void timeAllMaps() {
  final int bucketCount = 100_000, lockCount = 32;
  for (int t=1; t<=32; t++) {
    final int threadCount = t;
    Mark7(String.format("%-21s %d", "STACK", threadCount),
          i -> timeMap(threadCount, new ConcurrentStackImp()));
    }
  }

  private static double timeMap(int threadCount, final ConcurrentStackImp stack) {
    final int iterations = 5_000_000, perThread = iterations / threadCount;
    final int range = 200_000;
    return exerciseMap(threadCount, perThread, range, stack);
  }

  // TO BE HANDED OUT
private static double exerciseMap(int threadCount, int perThread, int range,
                                  final ConcurrentStackImp stack) {
  Thread[] threads = new Thread[threadCount];
  for (int t=0; t<threadCount; t++) {
    final int myThread = t;
    threads[t] = new Thread(() -> {
      Random random = new Random(37 * myThread + 78);
      for (int i=0; i<perThread; i++) {
        Integer item = random.nextInt(range);
        if (random.nextDouble() < 0.60) {
          stack.push(item);
        }
        else
```

```
          if (random.nextDouble() < 0.02) {
            stack.pop();
          }
        }
      });
    }
    for (int t=0; t<threadCount; t++)
      threads[t].start();
    try {
      for (int t=0; t<threadCount; t++)
        threads[t].join();
    } catch (InterruptedException exn) { }
    return stack.size();
  }

    private static void parallelTest(final ConcurrentStackImp stack) {
      Timer t = new Timer();
      int trials = 10_000;
      int npairs = 10;
      int threadCount = npairs * 2 + 1;
      System.out.printf("%nParallel test: %s", stack.getClass());
      final ExecutorService pool = Executors.newCachedThreadPool();
      new PushPopTest(stack, npairs, trials).test(pool);
      pool.shutdown();
      double time = t.check();
      System.out.println("... passed");
      System.out.printf("time = %10.2f ns; threadCount = %d\n", time, threadCount);
    }

    private static void seqTest(final ConcurrentStackImp stack) {
      System.out.printf("Sequential test %n%s%n", stack.getClass());

      assert stack.size() == 0;
      stack.push(1);
      stack.push(2);
      assert stack.size() == 2;
      assert stack.pop() == 2;
      assert stack.pop() == 1;
      assert stack.pop() == null;
    }

    private static void seqStripedTest(final StripedStack stack) {
      System.out.printf("Sequential test for StipedStac %n%s%n", stack.getClass());
      System.out.println();
      stack.push(10);
    }

    // NB: Modified to show microseconds instead of nanoseconds

  public static double Mark7(String msg, IntToDoubleFunction f) {
```

```
  int n = 10, count = 1, totalCount = 0;
  double dummy = 0.0, runningTime = 0.0, st = 0.0, sst = 0.0;
  do {
    count *= 2;
    st = sst = 0.0;
    for (int j=0; j<n; j++) {
      Timer t = new Timer();
      for (int i=0; i<count; i++)
        dummy += f.applyAsDouble(i);
      runningTime = t.check();
      double time = runningTime * 1e6 / count; // microseconds
      st += time;
      sst += time * time;
      totalCount += count;
    }
  } while (runningTime < 0.25 && count < Integer.MAX_VALUE/2);
  double mean = st/n, sdev = Math.sqrt((sst - mean*mean*n)/(n-1));
  System.out.printf("%-25s %15.1f us %10.2f %10d%n", msg, mean, sdev, count);
  return dummy / totalCount;
}

public static void SystemInfo() {
  System.out.printf("# OS:   %s; %s; %s%n",
                    System.getProperty("os.name"),
                    System.getProperty("os.version"),
                    System.getProperty("os.arch"));
  System.out.printf("# JVM:  %s; %s%n",
                    System.getProperty("java.vendor"),
                    System.getProperty("java.version"));
  // The processor identifier works only on MS Windows:
  System.out.printf("# CPU:  %s; %d \"cores\"%n",
                    System.getenv("PROCESSOR_IDENTIFIER"),
                    Runtime.getRuntime().availableProcessors());
  java.util.Date now = new java.util.Date();
  System.out.printf("# Date: %s%n",
    new java.text.SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssZ").format(now));
}

}

class Tests {
  public static void assertEquals(int x, int y) throws Exception {
    if (x != y)
      throw new Exception(String.format("ERROR: %d not equal to %d%n", x, y));
  }

  public static void assertTrue(boolean b) throws Exception {
    if (!b)
      throw new Exception(String.format("ERROR: assertTrue"));
  }
```

```java
}

class PushPopTest extends Tests {
  protected CyclicBarrier startBarrier, stopBarrier;
  protected final ConcurrentStackImp stack;
  protected final int nTrials, nPairs;
  protected final AtomicInteger putSum = new AtomicInteger(0);
  protected final AtomicInteger takeSum = new AtomicInteger(0);

  public PushPopTest(ConcurrentStackImp stack, int npairs, int ntrials) {
    this.stack = stack;
    this.nTrials = ntrials;
    this.nPairs = npairs;
    this.startBarrier = new CyclicBarrier(npairs * 2 + 1);
    this.stopBarrier = new CyclicBarrier(npairs * 2 + 1);
  }

  void test(ExecutorService pool) {
    try {
      for (int i = 0; i < nPairs; i++) {
        pool.execute(new Producer());
        pool.execute(new Consumer());
      }
      startBarrier.await(); // wait for all threads to be ready
      stopBarrier.await();  // wait for all threads to finish
      assertTrue(stack.isEmpty());
      assertEquals(putSum.get(), takeSum.get());
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }

  class Producer implements Runnable {
    public void run() {
      try {
        Random random = new Random();
        int sum = 0;
        startBarrier.await();
        for (int i = nTrials; i > 0; --i) {
          int item = random.nextInt();
          stack.push(item);
          sum += item;
        }
        putSum.getAndAdd(sum);
        stopBarrier.await();
      } catch (Exception e) {
        throw new RuntimeException(e);
      }
    }
  }
```

```
    class Consumer implements Runnable {
      public void run() {
        try {
          startBarrier.await();
          int sum = 0;
          for (int i = nTrials; i > 0; --i) {
            Integer item = null;
            while (item == null) { item = stack.pop(); }
            sum += item;
          }
          takeSum.getAndAdd(sum);
          stopBarrier.await();
        } catch (Exception e) {
          throw new RuntimeException(e);
        }
      }
    }
  }

  interface ConcurrentStackList {
    void push(int e);
    Integer pop();
    int size();
  }

  class ConcurrentStackImp implements ConcurrentStackList {
    private LinkedList<Integer> stack = new LinkedList<Integer>();
    private final Object lock = new Object();

    public void push(int e) {
      synchronized (lock) {
        stack.push(e);
      }
    }

    public Integer pop() {
      synchronized (lock) {
        if (!(stack.isEmpty())) {
          return stack.pop();
        }
        else return null;
      }
    }

    public int size() {
      return stack.size();
    }

    public boolean isEmpty() {
```

```java
    return stack.isEmpty();
  }
}

class StripedStack {
  ConcurrentStackImp[] buckets;

  public StripedStack(int bucketCount) {
    this.buckets = makeBuckets(bucketCount);
  }

  @SuppressWarnings("unchecked")
  private static ConcurrentStackImp[] makeBuckets(int size) {
  // Java's @$#@?!! type system requires this unsafe cast
    return (ConcurrentStackImp[])new ConcurrentStackImp[size];
  }

  // Protect against poor hash functions and make non-negative
  private static int getHash(Thread t) {
    final int th = t.hashCode();
    System.out.printf("th: %d\n", th);
    return (th ^ (th >>> 16)) & 0x7FFFFFFF;
  }

  public void push(int e) {
    Thread thread = Thread.currentThread();
    final int h = getHash(thread), hash = h % buckets.length;
    System.out.printf("hash: %d\n", hash);
    buckets[hash].push(e);
  }

  public Integer pop() {
    Thread thread = Thread.currentThread();
    final int h = getHash(thread), hash = h % buckets.length;
    Integer item = null;
    while (item == null) {
      item = buckets[hash].pop();
      return  item;
    }
    return null;
  }

  public int size() {
    int count = 0;
    for (int i = 0; i < buckets.length; i++) {
      count++;
    }
    return count;
  }
```

```
}

class Timer {
  private long start, spent = 0;
  public Timer() { play(); }
  public double check() { return (System.nanoTime()-start+spent); }
  public void pause() { spent += System.nanoTime()-start; }
  public void play() { start = System.nanoTime(); }
}
```

Q7

Results:

```
Press return to terminate...
public key: 4
private key: 22
cleartext: SECRET
encrypted: WIGVIX
decrypted: SECRET

Press return to terminate...
public key: 18
private key: 8
cleartext: SECRET
encrypted: KWUJWL
decrypted: SECRET

Press return to terminate...
public key: 17
private key: 9
cleartext: SECRET
encrypted: JVTIVK
decrypted: SECRET
```

Source Code:

```
import java.io.*;
import akka.actor.*;
import java.util.Random;
import java.util.HashMap;

public class SesComSys {
    public static void main(String[] args) {
        final ActorSystem system = ActorSystem.create("SesComSys");
        final ActorRef registry = system.actorOf(Props.create(RegistryActor.class), "Registry");
        final ActorRef sender = system.actorOf(Props.create(SenderActor.class), "Sender");
        final ActorRef receiver = system.actorOf(Props.create(ReceiverActor.class), "Receiver");

        receiver.tell(new InitMessage(registry), ActorRef.noSender());
        sender.tell(new InitMessage(registry), ActorRef.noSender());
        sender.tell(new CommMessage(receiver), ActorRef.noSender());

        try {
          System.out.println("Press return to terminate...");
          System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            system.shutdown();
        }
    }
}

class KeyPair implements Serializable {
    public final int public_key, private_key;
    public KeyPair(int public_key, int private_key) {
        this.public_key = public_key;
        this.private_key = private_key;
    }
}

class Crypto {
    static KeyPair keygen() {
        int public_key = (new Random()).nextInt(25)+1;
        int private_key = 26 - public_key;
        System.out.println("public key: " + public_key);
        System.out.println("private key: " + private_key);
        return new KeyPair(public_key, private_key);
}

  static String encrypt(String cleartext, int key) {
      StringBuffer encrypted = new StringBuffer();
      for (int i=0; i<cleartext.length(); i++) {
          encrypted.append((char) ('A' + ((((int)
              cleartext.charAt(i)) - 'A' + key) % 26)));
      }
      return "" + encrypted;
  }
}

class InitMessage implements Serializable {
  public final ActorRef registry;
  public InitMessage(ActorRef registry) { this.registry = registry; }
```

```java
}

class RegMessage implements Serializable {
  public final ActorRef receiver;
  public RegMessage(ActorRef receiver) { this.receiver = receiver; }
}

class KeyMessage implements Serializable {
  public final KeyPair keypair;
  public KeyMessage(KeyPair keypair) { this.keypair = keypair; }
}

class CommMessage implements Serializable {
  public final ActorRef receiver;
  public CommMessage(ActorRef receiver) { this.receiver = receiver; }
}

class LookupMessage implements Serializable {
  public final ActorRef receiver;
  public final ActorRef sender;
  public LookupMessage(ActorRef receiver, ActorRef sender) {
    this.receiver = receiver;
    this.sender = sender;
  }
}

class PubKeyMessage implements Serializable {
  public final ActorRef receiver;
  public final int public_key;
  public PubKeyMessage(ActorRef receiver, int public_key) {
    this.receiver = receiver;
    this.public_key = public_key;
  }
}

class Message implements Serializable {
    public final String s;
    public Message(String s) { this.s = s; }
}

class SenderActor extends UntypedActor {
  public ActorRef registry;

  public void onReceive(Object o) throws Exception {
    if (o instanceof InitMessage) {
      InitMessage init = (InitMessage) o;
      this.registry = init.registry;
    }
    else if (o instanceof CommMessage) {
      CommMessage comm = (CommMessage) o;
      ActorRef receiver = comm.receiver;
      registry.tell(new LookupMessage(receiver, getSelf()), getSelf());
    }
    else if (o instanceof PubKeyMessage) {
      PubKeyMessage pubkey = (PubKeyMessage) o;
      ActorRef receiver = pubkey.receiver;
      System.out.println("cleartext: SECRET");
      String encrypted = Crypto.encrypt("SECRET", pubkey.public_key);
      System.out.println("encrypted: " + encrypted);
      receiver.tell(new Message(encrypted), ActorRef.noSender());
    }
  }
}

class ReceiverActor extends UntypedActor {
  public ActorRef registry;
  public KeyPair keypair;

  public void onReceive(Object o) throws Exception {
    if (o instanceof InitMessage) {
      InitMessage init = (InitMessage) o;
      this.registry = init.registry;
      this.registry.tell(new RegMessage(getSelf()), getSelf());
    }
    else if (o instanceof KeyMessage) {
      KeyMessage keys = (KeyMessage) o;
      this.keypair = keys.keypair;
    }
    else if (o instanceof Message) {
      Message message = (Message) o;
      String decrypted = Crypto.encrypt(message.s, keypair.private_key);
      System.out.println("decrypted: " + decrypted);
    }
  }
}

class RegistryActor extends UntypedActor {
  public ActorRef receiver;
  HashMap<ActorRef, Integer> pk_keys = new HashMap<ActorRef, Integer>();

  public void onReceive(Object o) throws Exception {
    if (o instanceof RegMessage) {
      RegMessage regmsg = (RegMessage) o;
      this.receiver = regmsg.receiver;
      KeyPair key = Crypto.keygen();
      pk_keys.put(this.receiver, key.public_key);
      this.receiver.tell(new KeyMessage(key), ActorRef.noSender());
    }
    else if (o instanceof LookupMessage) {
      LookupMessage lookup = (LookupMessage) o;
      ActorRef sender = lookup.sender;
      ActorRef receiver = lookup.receiver;
      int public_key = pk_keys.get(lookup.receiver);
      sender.tell(new PubKeyMessage(receiver, public_key), ActorRef.noSender());
    }
  }
}
```