UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

IMPLEMENTACIÓN DE UN ALGORITMO DISTRIBUIDO DE GENERACIÓN DE CLAVES RSA CON UMBRAL

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERA CIVIL EN COMPUTACIÓN

CATERINA MUÑOZ VILDÓSOLA

PROFESOR GUÍA:
JAVIER BUSTOS JIMÉNEZ

MIEMBROS DE LA COMISIÓN:
ALEJANDRO HEVIA ANGULO
CLAUDIO GUTIÉRREZ GALLARDO

Este trabajo ha sido parcialmente financiado por Nic Chile Research Labs

SANTIAGO DE CHILE
2016

La presente memoria detalla el proceso de implementación de un algoritmo distribuido de generación de claves RSA umbral.

RSA es un sistema criptográfico de clave privada. Esto significa que se requieren dos claves distintas para realizar las operaciones criptograficas: una privada y una pública. En un sistema RSA la clave privada consiste en un valor privado (llamado exponente de firma o desencriptación) mientras que la clave pública consiste en dos valores (el exponente de verificación o encriptación, más un valor denominado "módulo"). En un sistema RSA tanto el exponente privado como la factorización del módulo deben mantener secretos para garantizar la seguridad.

En RSA umbral, la operación privada está distribuida entre n nodos. De los n nodos mencionados, se requieren t para poder realizar una operación criptográfica privada de manera exitosa.

Dado que la operación privada está distribuida, la clave privada también debe estarlo. Además, tanto la claves privadas como la factorización del módulo RSA deben seguir siendo secretos. Dado lo anterior, al momento de generar las claves ningún nodo debe tomar conocimiento ni de las claves privadas ajenas, ni de la factorización del módulo RSA.

El trabajo de esta memoria consistió en implementar el algoritmo de generación distribuida de claves RSA umbral propuesto por D. Boneh y M. Franklin en [6]. Dicho algoritmo logra generar un módulo RSA y un conjunto de claves privadas umbral sin que ningún nodo obtenga información sobre la factorización del módulo ni sobre las claves ajenas. A diferencia de trabajos previos, el algoritmo logra lo anterior sin requerir de un actor confiable que genere y distribuya las claves. Cabe destacar que el tiempo de ejecución del algoritmo es aleatorizado, por lo que no se puede predecir cuánto tomará en ejecutarse. A pesar de lo anterior, hay un tiempo de ejecución esperado.

Se realizaron pruebas que comprobaron que la implementación estaba correcta y se comportaba de acuerdo a lo especificado en el algoritmo original. Además, se pudo comprobar que el promedio de los tiempos de ejecución medidos fueron menores al tiempo de ejecución esperado.

A mi familia. Y a la computación, por ser bacán.

Agradecimientos

A mi familia: a mi mamá, a todos mis hermanos, a Claudio y a mi Nonna. También a mi papá y a Mónica. Ellos me han apoyado ininterrumpidamente desde que comencé esta etapa hasta ahora que estoy terminándola.

A mis profesores guía y a todos los integrantes y ex-integrantes de NicLabs, por acompañarme en la recta final y enseñarme cosas que no hubiera aprendido en ningún otro lugar.

A todos los amigos que hice en la universidad: los amigos de la sección 8, los amigos de la sección 1, los Curiosillos, mis amigos del DCC, mis amigos de NicLabs y los que no caen en ninguno de los grupos anteriores. Todos ellos hicieron que mi paso por la universidad se pasara volando.

Al Benja, por la incondicionalidad, las conversaciones sin contenido y las guerras de cosquillas.

Finalmente, a mi tía Amy Bardi, por corregirme la redacción y ortografía de este documento.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

1.1. Sobre Buzones y Encriptación de Clave Pública

En muchos sentidos, se puede decir que un sistema de encriptación de clave pública es como un buzón con llave. Cualquier persona con acceso físico al buzón puede introducirle una carta, pero sólo el poseedor de la llave puede abrir el buzón para leer los mensajes. De manera análoga, cualquier persona con acceso a la clave pública del sistema criptográfico puede cifrar un mensaje. Sin embargo, la única forma de recuperar el mensaje original es descifrar el mensaje encriptado usando la clave privada asociada a la clave pública que se ocupó anteriormente.

El problema es el mismo para ambos contextos: ¿qué pasa si alguien roba la llave o la clave? El ladrón podrá ver los mensajes destinados al dueño original. En el caso del sistema criptográfico, el problema se puede resolver agregándole un comportamiento umbral.

Volviendo a la analogía inicial, un buzón-umbral sería un buzón con un conjunto de n cerraduras en vez de una. Sin embargo, no es necesario abrir las n cerraduras para poder abrir el buzón, sino que basta con abrir un subconjunto cualquiera de t cerraduras. Ahora el sistema es más seguro, ya que las n llaves pueden esconderse en lugares distintos. Más aún, algunas de las llaves pueden perderse y el buzón sigue siendo funcional.

Un sistema de encriptación de clave pública umbral es similiar. La clave pública sigue estando disponible para cifrar mensajes. La diferencia radica en la clave privada, que en este caso no es sólo una, sino varias. Las n claves privadas están distribuidas en n nodos y para descifrar un mensaje se requieren t claves (en realidad, se requieren los mensajes desencriptados parciales de t claves, los cuales se unen con una operación de *join* para generar el mensaje desencriptado final). Las ventajas del sistema son análogas a las mencionadas con el buzón umbral.

1.2. Motivación

El sistema RSA (ver sección 2.2) es uno de los esquemas criptográficos de clave pública más usados en el mundo. Dada su popularidad, es natural que se hayan propuesto varios esquemas criptográficos que extienden RSA para que tenga un comportamiento umbral como el descrito en la sección anterior. Así, trabajos como los de Tal Rabin [12] o Victor Shoup [14], proponen esquemas de criptografía umbral RSA.

Ahora bien, ambos esquemas mencionados tienen un factor en común: al momento de generar las claves, se debe contar con un participante externo confiable, que se encargue de generar y repartir las claves privadas. Pero, ¿qué pasaría si ese participante fuera atacado y revelara información de las claves? El sistema se volvería completamente vulnerable, ya que se hubiera perdido la privacidad de las claves.

Para solucionar el problema anterior, Dan Boneh y Matthew Franklin propusieron un algoritmo capaz de generar las claves de manera distribuida y segura [6]. Utilizando este algoritmo, los n nodos encargados de guardar las claves privadas serán también quienes las generen (también generan la clave pública). La gran ventaja del algoritmo, es que en cada momento de la ejecución, cada nodo sólo conoce información concerniente a su propia clave privada, y nunca de las claves de los otros.

El algoritmo en cuestión fue implementado por Michael Malkin, Thomas Wu y Dan Boneh en [11]. Sin embargo, su implementación no está disponible para el público. Por consiguiente, la motivación de este trabajo es proveer una implementación disponible públicamente del algoritmo propuesto por Malkin, Wu y Boneh.

1.3. Objetivos

1.3.1. Objetivo General

El objetivo principal de esta memoria es implementar un algoritmo de generación distribuida de llaves RSA umbral basado en el algoritmo propuesto en [6] y la implementación descrita en [11].

1.3.2. Objetivos Específicos

- Definir detalladamente el algoritmo a implementar.
- Diseñar e implementar un sistema de comunicación para que los actores del algoritmo interactúen.
- Implementar el algoritmo definido.
- Cuidar que la implementación sea modular y extensible.

- Probar que la implementación del algoritmo funcione de acuerdo a su comportamiento esperado.
- Medir el tiempo de ejecución de la implementación del algoritmo con distintos parámetros.

1.4. Organización del Documento

En términos generales, el presente documento relata el proceso llevado a cabo para implementar un algoritmo de generación distribuida de llaves criptográficas para un sistema RSA umbral. Dado eso, el documento incluye, entre otras cosas, la descripción completa del algoritmo implementado, la explicación de la implementación realizada y las pruebas que se le realizaron a la implementación.

En el capítulo 2 (Antecedentes), se explican los conceptos que se deben conocer para poder entender el algoritmo y su implementación. Entre los tópicos que se introducen están: criptografía asimétrica, encriptación RSA y criptografía umbral.

Luego, en el capítulo 3 (Análisis y Diseño), se describe de manera general y detallada el algoritmo que se implementó. También se mencionan y justifican las decisiones de diseño que se tomaron antes de la implementación.

A continuación, en el capítulo 4 (Implementación), se detalla la estructura de la implementación y cada una de sus módulos. Se da especial énfasis a los aspectos distribuidos de la implementación.

Posteriormente, en el capítulo 5 (Resultados), se especifican las pruebas que se le realizaron a la implementación para comprobar su correcto funcionamiento y los resultados obtenidos de dichas pruebas.

Finalmente, en el capítulo 6 (Conclusiones), se presentan las conclusiones y experiencias que dejó el proceso del trabajo en su totalidad.

Capítulo 2

Antecedentes

Para entender el trabajo realizado, se requieren conocer previamente varios conceptos de criptografía, tales como: Criptografía Asimétrica, Sistema Criptográfico RSA y Criptografía Umbral. Para cada concepto, esta sección entrega una pequeña explicación que es suficiente para entender el resto del trabajo.

2.1. Criptografía Asimétrica

También llamada criptografía de clave pública. En este tipo de Criptografía se ocupan claves distintas para realizar las operaciones criptográficas (encriptación y desencriptación, firmado y verificación). Una de las claves es pública y está disponible para que se realice la operación criptográfica "pública"; la otra clave es privada y se ocupa para realizar la operación criptográfica "privada". Se recomienda ver [15] para más información.

2.1.1. Sistemas Criptográficos de clave Pública

1. **Encriptación de Clave Pública** [9]
    - **Operación Pública:** El remitente ocupa la clave pública del destinatario para cifrar un mensaje.
    - **Operación Privada:** El destinatario es el único que conoce la clave privada, y por ende, el único que puede desencriptar los mensajes encriptados con su clave pública.
2. **Firmas Digitales** [8]
    - **Operación Privada:** El remitente puede firmar un mensaje con su clave privada.
    - **Operación Pública:** Cualquier persona con acceso a la clave pública del remitente puede verificar sus mensajes, autentificando al remitente.

De ahora en adelante, para mantener la generalidad, se usarán lo conceptos de operación critográfica privada y operación criptográfica pública, sin referirse particularmente ni a cifrado ni a firmas.

2.2. Sistema Criptográfico RSA

Un sistema criptográfico de clave pública. Fue propuesto por Rivest, Shamir y Adleman en [13]. Su seguridad radica en la dificultad del problema de factorización de números enteros. La clave pública es el par (e, N), donde N es el módulo RSA y e es el exponente público. La clave privada es el par (d, N), donde N es el mismo módulo RSA de la clave pública y d es el exponente privado.

2.2.1. Funcionamiento de RSA

A continuación se explicará el uso de un sistema RSA para cifrar y descifrar un mensaje. Se supondrá que el mensaje a cifrar es un entero m (existen métodos reversibles para tranformar de un string a un entero, por lo que este ejemplo es genérico).

Para cifrar un mensaje m, simplemente se debe usar la clave pública para calcular:

$$c = m^{\mathrm{e}} \mod N$$

Para descifrar el mensaje encriptado, se ocupa la clave privada de manera análoga:

$$m = c^{\mathrm{d}} \mod N$$

2.2.2. Generación de Claves

Para que lo explicado anteriormente funcione, las claves se deben generar de tal manera que:

$$(m^{\mathrm{e}})^{\mathrm{d}} \mod N = m$$

Para que esto se cumpla, las claves se generan de la siguiente manera:

1. Elegir dos primos al azar, p y q.
2. Calcular $N = p \cdot q$
3. Calcular $\phi(N) = N - p - q - 1$
4. Elegir el exponente público e, con $1 < \mathrm{e} < \phi(N)$ y $mcd(\mathrm{e}, \phi(N)) = 1$. Uno de los valores más usados para e es $2^{16} + 1 = 65537$
5. Calcular el exponente privado d despejando la siguiente ecuación: $\mathrm{e} \cdot \mathrm{d} = 1 \mod \phi(N)$.

Nótese que además de mantener en secreto d, también se debe de hacer lo mismo con p, q y $\phi(N)$.

2.2.3. Pequeño Ejemplo

Este es un ejemplo de RSA con números pequeños. Se ocupará $p = 73$, $q = 83$ y e $= 17$.

Generación de Claves

1. $N = p \cdot q = 6059$
2. $\phi(N) = N - p - q + 1 = 5904$
3. $\mathrm{d} = \mathrm{e}^{-1} \mod \phi(N) = 3473$

Cifrado y Descifrado

1. $m = 1000$
2. $encr = m^{\mathrm{e}} \mod N = 3628$
3. $decr = encr^{\mathrm{d}} \mod N = 1000 = m$

RSA en la Práctica

Hoy en día, los sistemas RSA son más complejos que lo explicado anteriormente. Esto se debe a que se ocupan en conjunto con OAEP(*Optimal Asymmetric Encryption Padding*)[3], que es un sistema de *padding* aleatorizado.

2.3. Criptografía Umbral

Tiene como objetivo distribuir una funcionalidad criptográfica entre n nodos. El sistema debe funcionar de manera tal que un conjunto de t nodos sea capaz de realizar exitosamente la operación privada y cualquier conjunto de tamaño menor no pueda hacerlo. El valor t es el parámetro umbral del sistema (t es un entero y además cumple: $1 \leq t \leq n$). Ver [10] para más información.

2.3.1. Criptografía RSA Umbral

Un sistema criptográfico umbral cuya seguridad radica en la dificultad del problema de factorización de enteros. La operación pública se realiza de la misma manera que en un esquema RSA simple, utilizando la clave pública (e, N) (que está públicamente disponible). Sin embargo, la operación privada está distribuida entre n nodos, t de los cuales son necesarios para concretar exitosamente una operación privada.

La distribución de la operación privada se ve reflejada en la distribución de la clave privada. Así, en vez de tener una única clave d, se tienen varias claves $\{\mathrm{d_i}\}_{\mathrm{i}=1}^{n}$. Cada una de las claves privadas es conocida únicamente por un nodo. Cada nodo sólo puede realizar la operación privada parcialmente (utilizando la clave que conoce), su resultado parcial se debe unir con otros $t-1$ resultados parciales de otros nodos (utilizando un algoritmo adecuado) para obtener un resultado final y lograr exitosamente la operación privada.

El conjunto de claves $\{\mathrm{d_i}\}_{\mathrm{i}=1}^{n}$ debe ser generado de tal manera que con cualquiera de sus subconjuntos de tamaño t se pueda realizar una operación privada. Este último problema, de generación de claves RSA umbral, es justamente el tema de esta memoria.

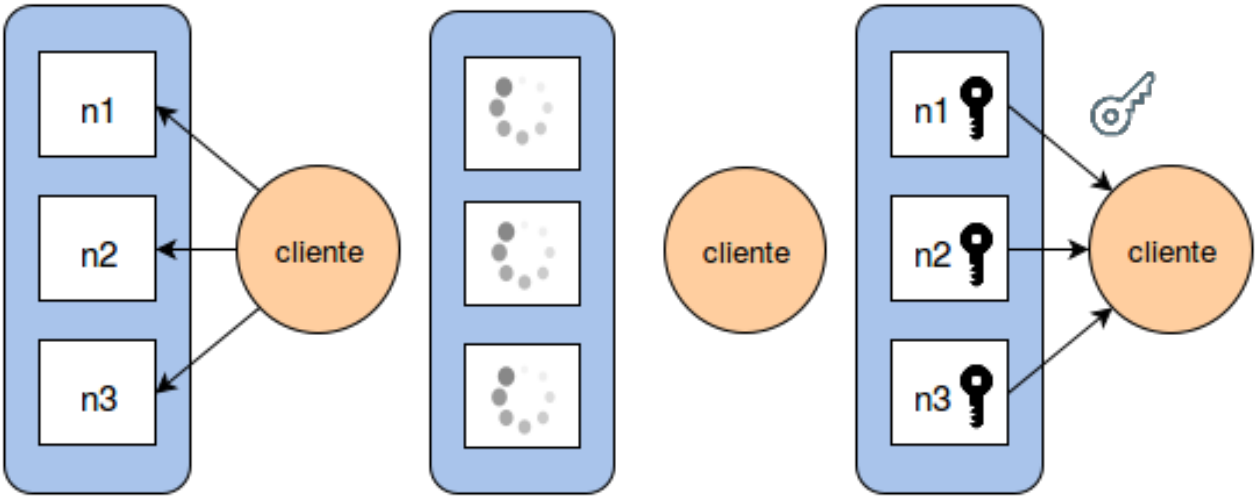
Capítulo 3

Análisis y Diseño

3.1. Actores del Algoritmo y su Interacción

En esta sección se describen los actores que participan en el algoritmo y la manera en que interactúan entre ellos.

El algoritmo tiene como objetivo la generación distribuida de claves RSA con umbral. Como se explicó en la sección 2.3, lograr el objetivo implica generar una clave pública y un conjunto de claves privadas. La tarea de generar las claves la lleva a cabo un conjunto de máquinas llamadas nodos. Los nodos trabajan cooperativamente para generar la clave pública y las claves privadas. Para que el usuario del algoritmo interactúe con los nodos, existe otra máquina llamada cliente. El cliente es el intermediario entre el usuario y los nodos, por lo que en principio tiene dos funciones: informar a los nodos de la solicitud del cliente e informar al cliente del resultado de los nodos.



Paso 1. Paso 2. Paso 3.

Figura 3.1: Esquema de Interacción entre Cliente y Nodos

En términos generales, este es el esquema se sigue la ejecución del algoritmo. Los pasos 1-3 están ilustrados en la Figura 3.1:

0. Los nodos están esperando una solicitud de ejecución.

1. Aparece un cliente. El cliente se comunica con todos los nodos y les solicita ejecutar el algoritmo con ciertos parámetros.

2. Los nodos trabajan cooperativamente entre ellos para ejecutar el algoritmo y obtener un resultado (esto está explicado con detalle en la próxima sección). Como se ha dicho anteriormente, ese resultado consiste en el conjunto de claves privadas y la clave pública.

3. Los nodos le informan al cliente que la ejecución terminó y le entregan la clave pública que obtuvieron. El conjunto de claves privadas queda distribuido entre los nodos.

4. Los nodos están listos para usar sus claves privadas y realizar operaciones privadas de criptografía umbral (verificar firmas o desencriptar mensajes).

3.2. Descripción del Algoritmo

En esta sección se brinda una descripción completa del algoritmo implementado. Esta descripción abarca desde que los nodos reciben la solicitud de ejecución por parte del cliente hasta que los nodos logran calcular las claves privadas. Los pasos de solicitud por parte del cliente hacia los nodos y de reporte desde los nodos al cliente están omitidos dado que se explicaron en la sección anterior.

3.2.1. Especificaciones de Parámetros de Entrada y Resultados del Algoritmo

Parámetros de Entrada

Hay algunos parámetros que se deben entregar al algoritmo para que se pueda ejecutar:

- La cantidad de nodos que participarán: n, con $n \geq 3$.
- El parámetro umbral: t, con $t > \frac{n}{2}$
- El tamaño en bits esperado del módulo RSA: b
- El exponente público: e.

Resultados del Algoritmo

Al terminar la ejecución del algoritmo, se tiene lo siguiente:

- Una clave pública RSA compuesta por el exponente público definido anteriormente (e) y el módulo RSA N. El valor N debe ser el producto de dos números primos y tener un tamaño de a lo más b bits y a lo menos $b-4$ bits.
- Una clave privada RSA d distribuida entre los n nodos participantes. La clave está distribuida de tal manera que se puede realizar criptografía umbral; esto es, basta que un subconjunto cualquiera de t nodos realice la operación privada parcial para poder realizar una operación privada exitosa. Así, el nodo i−ésimo guarda la clave privada parcial $\mathrm{d_i}$.

3.2.2. Descripción General del Algoritmo

La siguiente es una descripción de muy alto nivel de los pasos que sigue el algoritmo. En la próxima sección está la explicación detallada de cada uno de los pasos.

1. El nodo i elige p_i y q_i.
2. Los nodos usan sus p_i y q_i para calcular un candidato N de acuerdo a la siguiente fórmula:

$$N = p \cdot q = \Big(\sum_{\mathrm{i}=1}^{n} p_\mathrm{i}\Big) \cdot \Big(\sum_{\mathrm{i}=1}^{n} q_\mathrm{i}\Big)$$

   Este cálculo se realiza sin revelar los p_i y q_i.
3. Los nodos le realizan un test de biprimalidad a N. Si N es biprimo (producto de dos primos), se avanza al paso siguiente; si no, el algoritmo vuelve a comenzar desde el paso 1. Cabe destacar que no es necesario calcular p ni q explícitamente para determinar si N es producto de dos primos.
4. A partir de N y e se calcula un conjunto de claves preliminar $\{\mathrm{d}'_\mathrm{i}\}$. Estas claves cumplen que:

$$\mathrm{d} = \sum_{\mathrm{i}=1}^{n} \mathrm{d}'_\mathrm{i}$$

   Con este conjunto preliminar no se puede hacer criptografía umbral. Dada la propiedad mencionada anteriormente, a este conjunto de claves se le llamará conjunto aditivo de d.
5. Usando el conjunto $\{\mathrm{d}'_\mathrm{i}\}$, se calcula el conjunto final de claves $\{\mathrm{d_i}\}$. A diferencia del conjunto preliminar, este conjunto final sí permite realizar criptografía umbral. A este conjunto se le llamará conjunto umbral de d.

La Figura 3.2 ilustra el flujo que siguen los pasos del algoritmo.

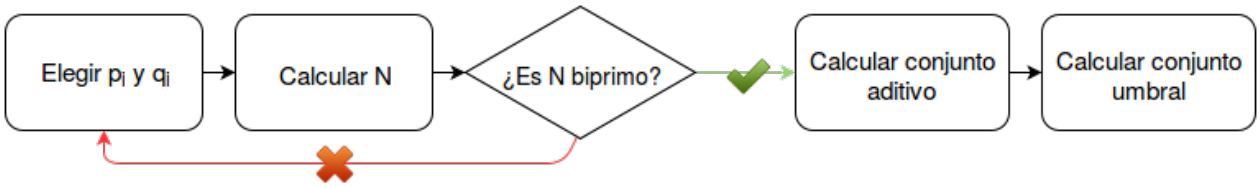


Figura 3.2: Esquema de los Pasos del Algoritmo

3.2.3. Descripción Completa de los Pasos del Algoritmo

A continuación se explica con detalle lo que se hace en cada uno de los pasos mencionados en la descripción anterior. La mayoría de los "pasos" descritos son en realidad algoritmos en sí.

Elección de Candidatos

En este paso, cada nodo elige dos números al azar p_i y q_i. Estos números son los "fragmentos" que se ocuparán para calcular el módulo N en el próximo paso de acuerdo a la fórmula:

$$N = p \cdot q = \left(\sum_{i=1}^{n} p_i\right) \cdot \left(\sum_{i=1}^{n} q_i\right)$$

Así, en este paso, cada nodo i hace lo siguiente:

1. Elegir dos números al azar p'_i y q'_i en el rango $[\frac{2^{a-2}}{n}, \frac{2^{a-4}}{n}]$, donde $a = \frac{b}{2}$ y b es el tamaño deseado del módulo N en bits.

2. Calcula p_i y q_i de acuerdo a lo siguiente:

   - Si i $= 1$, $p_i = 4p'_i - 1$ y $q_i = 4q'_i - 1$
   - Si i $\neq 1$, $p_i = 4p'_i$ y $q_i = 4q'_i$.

Al elegir de esta manera los p_i y q_i, se garantizan dos cosas:

a) El tamaño de p y q será a lo más a bits y de a lo menos $a - 2$ bits. Dado eso, el tamaño del módulo N será de a lo más b bits y a lo menos $b - 4$ bits.

b) Tanto p como q son 3 módulo 4. Gracias a esto, $N - p - q + 1$ será 0 módulo 4. Esta condición es necesaria para poder ejecutar el paso 3: Test de Biprimalidad.

Cálculo Distribuido del Módulo N

El objetivo de este paso es calcular el módulo RSA candidato N utilizando los fragmentos p_i y q_i que cada nodo eligió en el paso anterior. El cálculo se debe hacer de tal forma que

ningún nodo conozca los fragmentos de los otros nodos. A primera vista, esto puede sonar como una tarea imposible, sin embargo, se puede lograr utilizando una versión simplificada del método distribuido de computación privada BGW (Ben-Or, Goldwasser, Wigderson) [5].

Antes de calcular N, se deben cumplir las siguientes condiciones:

- El nodo i−ésimo es el único que conoce p_i y q_i. Esto es un resultado del paso anterior.
- Todos los nodos conocen un primo P de gran tamaño definido anteriormente. P debe cumplir que: $N < (n2^b)^2 < P$, donde b es el tamaño en bits deseado para N. Para efectos de esta implementación, se ocupó $P = M_{4423} = 2^{4423} - 1$.

Bajo el supuesto de que las condiciones anteriores se cumplen, se siguen los siguientes pasos para calcular N de manera distribuida y privada (sin exponer los fragmentos de cada nodo).

Recordar que:

$$N = p \cdot q = \Big(\sum_{\mathrm{i}=1}^{n} p_\mathrm{i}\Big) \cdot \Big(\sum_{\mathrm{i}=1}^{n} q_\mathrm{i}\Big)$$

Cada nodo i hace lo siguiente:

1. Calcular $l = \lfloor \frac{n-1}{2} \rfloor$.
2. Elegir tres polinomios en $\mathbb{Z}_P$: $f_\mathrm{i}(x)$, $g_\mathrm{i}(x)$ y $h_\mathrm{i}(x)$. $f_\mathrm{i}(x)$ y $g_\mathrm{i}(x)$ son de grado l, con $f_\mathrm{i}(0) = p_\mathrm{i}$ y $g_\mathrm{i}(0) = q_\mathrm{i}$. $h_\mathrm{i}(x)$ es de grado $2l$ y $h_\mathrm{i}(0) = 0$.
3. Para $j = 1 \ldots n$
   - Calcular $p_{\mathrm{i},j} = f_\mathrm{i}(j)$, $q_{\mathrm{i},j} = g_\mathrm{i}(j)$ y $h_{\mathrm{i},j} = h_\mathrm{i}(j)$
   - Enviarle el mensaje $\{p_{\mathrm{i},j}, q_{\mathrm{i},j}, h_{\mathrm{i},j}\}$ al nodo j.
4. Recibir los n mensajes $\{p_{j,\mathrm{i}}, q_{j,\mathrm{i}}, h_{j,\mathrm{i}}\}$, para $j = 1 \ldots n$ (mensajes enviados en el paso anterior desde los otros nodos y el mismo nodo i)
5. Computar
   $$N_\mathrm{i} = \Big(\sum_{j=1}^{n} p_{j,\mathrm{i}}\Big) \cdot \Big(\sum_{j=1}^{n} q_{j,\mathrm{i}}\Big) + \sum_{j=1}^{n} h_{j,\mathrm{i}} \mod P$$
   y hacer *broadcast* del mensaje $\{N_\mathrm{i}\}$
6. Recibir los n mensajes $\{N_j\}$. Notar que cada N_j corresponde a un punto de un polinomio $\alpha(x)$ evaluado en j, donde
   $$\alpha(x) = \Big(\sum_{j=1}^{n} f_j(x)\Big) \cdot \Big(\sum_{j=1}^{n} g_j(x)\Big) + \sum_{j=1}^{n} h_j(x) \mod P$$
   Además, $\alpha(x)$ es un polinomio de grado $2l$

7. Interpolar el polinomio α y calcular $\alpha(0)$ (se cumple que el polinomio es de grado $2l$ y el nodo tiene $n > 2l$ puntos). $\alpha(0) = N \mod P$ y como $N < P$, $\alpha(0) = N$

De esta manera, se puede computar N de manera distribuida y privada, sin tener que calcular explícitamente p ni q, y sin que ningún nodo exponga sus fragmentos p_i y q_i. Notar que como todos los nodos ejecutan los pasos anteriores, al final del algoritmo todos conocen el valor de N.

Test de Biprimalidad

En este punto del algoritmo se tiene el N candidato calculado del paso anterior. Se sabe que N es el producto de dos números (p y q), pero se quiere comprobar que dichos números sean primos. Para lograr esto, se hace un test de biprimalidad sobre N. El test indica si N es o no el producto de dos números primos. En el caso de que N sea biprimo (producto de dos primos), se continúa con el siguiente paso del algoritmo. En la situación contraria, se vuelve a comenzar desde el primer paso de Elección de Candidatos.

La ventaja de este test es que permite indentificar si p y q son primos sin tener que calcularlos explícitamente y sin que ningún nodo tenga que exponer sus segmentos p_i y q_i. De esta manera, se mantienen las mismas condiciones de privacidad del paso anterior.

El test de biprimalidad sigue los siguientes pasos:

1. Los nodos se ponen de acuerdo en un $g \in \mathbb{Z}_N^*$. Además se debe cumplir que $\left(\frac{g}{N}\right) \neq 1$ (el símbolo de Jacobi [4] de g con N no puede ser 1).

   Para elegir g, cada nodo hace lo siguiente:

   a) Elegir un $g_\text{i} \in [1, N-1]$.

   b) Hacer *broadcast* del mensaje $\{g_\text{i}\}$.

   c) Recibir todos los g_j y calcular $g = \sum_{j=1}^{n} g_j \mod N$.

   d) Verificar que $mcd(g, N) = 1$. Si no lo es, volver a empezar desde el paso a). Si lo es, continuar con el siguiente paso.

   e) Verificar que $\left(\frac{g}{N}\right) = 1$ (símbolo de Jacobi [4]). Si no lo es, volver a empezar desde el paso a). Si lo es, el algoritmo termina y retorna el g encontrado.

2. Cada nodo calcula un valor v_i de acuerdo a lo siguiente:

   $v_1 = g^{\frac{N-p_1+q_1+1}{4}} \mod N$

   $v_\text{i} = g^{\frac{-p_\text{i}-q_\text{i}}{4}} \mod N$ (inverso modular), i $\neq 1$.

   Los exponentes de los v_i son números enteros. Esto se debe a la forma en que los p_i y q_i fueron elegidos en el paso de Elección de Candidatos.

3. Los nodos comparten sus v_j entre ellos y comprueban si:

$$\prod_{j=1}^{n} v_j \stackrel{?}{=} \pm 1 \mod N$$

4. Si lo anterior se cumple, se declara que N es el producto de dos primos y se prosigue con el algoritmo. Si no, se vuelve a comenzar desde el paso de Elección de Candidatos.

La demostración del correcto funcionamiento del test descrito anteriormente se puede encontrar en [6].

Generación del Conjunto Aditivo de d

Si se alcanzó este paso, entonces se obtuvo un N que es el producto de dos primos. A partir de ahora, N deja de ser un candidato y pasa a ser el módulo RSA de las claves que se generarán.

Como e es dado y se encontró un N que cumple con las condiciones necesarias, la clave pública queda definida como el par (e, N). Queda entonces generar las claves privadas $\mathrm{d_i}$.

En este paso, los nodos generan un conjunto preliminar de claves aditivas $\{\mathrm{d'_i}\}_{\mathrm{i}=1}^{n}$. El conjunto está distribuido entre los nodos: cada nodo i sólo conoce el valor $\mathrm{d'_i}$ (además, en ningún momento del algoritmo conoce otros valores $\mathrm{d'}_{j\neq \mathrm{i}}$).

Estas claves cumplen que:

$$\sum_{\mathrm{i}=1}^{n} \mathrm{d'_i} = \mathrm{d}$$

Donde d sería la clave privada de un esquema RSA normal usando el N encontrado como módulo y el e dado como exponente público (ver Sección 2.2). En otras palabras:

$$\mathrm{d} = \mathrm{e}^{-1} \mod \phi(N)$$

La generación de este conjunto preliminar es necesaria para poder generar las claves finales, que sí permiten realizar criptografía umbral.

Estos son los pasos que sigue cada nodo i para cumplir el objetivo de esta etapa:

1. Calcular ϕ_i según el valor de i:

   - $\phi_1 = N - p_1 - q_1 + 1$
   - $\phi_\mathrm{i} = p_\mathrm{i} + q_\mathrm{i}$, $\mathrm{i} \neq 1$

2. Generar un *sharing aditivo* $\{\gamma_{\mathrm{i},j}\}_{j=1}^{n}$ de ϕ_i. En otras palabras, generar un conjunto $\{\gamma_{\mathrm{i},j}\}_{j=1}^{n}$ que cumpla: $\sum_{j=1}^{n} \gamma_{\mathrm{i},j} = \phi_\mathrm{i}$.

3. Para $j = 1 \ldots n$

- Enviarle el mensaje $\{\gamma_{\mathrm{i},j}\}$ al nodo j (cada nodo envía n mensajes).

4. Recibir los n mensajes $\{\gamma_{\mathrm{i},j}\}$, para $j = 1 \ldots n$ y calcular $\alpha_{\mathrm{i}} = \sum_{j=1}^{n} \gamma_{j,\mathrm{i}}$

5. Hacer *broadcast* del mensaje $\{\alpha_{\mathrm{i}}\}$.

6. Recibir los n α_j y calcular $\alpha = \sum_{j=1}^{n} \alpha_j$. Notar que:

$$\alpha = \sum_{j=1}^{n} \alpha_j = \sum_{j=1}^{n} \sum_{k=1}^{n} \gamma_{k,j} = \sum_{j=1}^{n} \sum_{k=1}^{n} \gamma_{k,j} = \sum_{k=1}^{n} \sum_{j=1}^{n} \gamma_{k,j} = \sum_{k=1}^{n} \phi_k$$

   De esta manera, se logró calcular $\sum_{k=1}^{n} \phi_k = N - p - q + 1$. Este valor es necesario para invertir e distribuidamente.

7. Calcular $l = \alpha \mod \mathrm{e}$ y $\xi = l^{-1} \mod \mathrm{e}$

8. Definir $\mathrm{d}'_{\mathrm{i}} = \lfloor \frac{-\xi \cdot \phi_{\mathrm{i}}}{\mathrm{e}} \rfloor$

9. Realizar ajuste de d'_1 (ver explicación a continuación).

El último paso consiste en ajustar d'_1 en un error r que se pudo haber generado al aplicar los pisos en el paso 8. Para lograr esto, hay que probar ajustando d'_1 con todos los valores posibles que puede tener r: $[0...(n-1)]$ y comprobar si el valor es correcto efectuando una operación de descifrado de prueba.

Estos son los pasos que siguen los nodos en conjunto para ajustar d'_1:

1. El nodo 1 elige un número aleatorio $0 < m < N$. El valor m será el mensaje que se cifrará para hacer la prueba.

2. El nodo 1 cifra el mensaje usando la clave pública. $C = m^{\mathrm{e}} \mod N$.

3. El nodo 1 le manda a todos los otros nodos el mensaje $\{C\}$ junto con una solicitud de descifrado.

4. Todos los nodos menos el nodo 1 ($\mathrm{i} = 2...n$), calculan el descifrado parcial de su clave: $D_{\mathrm{i}} = C^{\mathrm{d}'_{\mathrm{i}}} \mod N$ y se lo envían al nodo 1.

5. El nodo 1 recibe los descifrados parciales del resto de los nodos y calcula el producto módulo N. $P = \prod_{\mathrm{i}=2}^{n} D_{\mathrm{i}} \mod N$

6. El nodo 1 prueba todos los valores posibles de r:

   para $r = 0...(n-1)$

   - $\mathrm{d}^*_1 = \mathrm{d}1' + r$
   - $D_1 = C^{\mathrm{d}^*_1} \mod N$
   - $T = (D_1 \cdot P) \mod N$
   - si $T = m$, interrumpir el ciclo y redefinir $\mathrm{d}'_1 = \mathrm{d}^*_1$

Así, al final de este paso, cada nodo i tiene el elemento d'_i del conjunto $\{\mathrm{d}'_\mathrm{i}\}_{\mathrm{i}=1}^{n}$ que cumple que: $\sum_{\mathrm{i}=1}^{n} \mathrm{d}'_\mathrm{i} = \mathrm{d} = \mathrm{e}^{-1} \mod \phi(N)$.

Generación del Conjunto Umbral de d

En este punto, cada nodo i conoce el valor d'_i del conjunto de claves preliminar. Lo único que falta es generar el conjunto de claves final que permitirá realizar criptografía umbral. Para lograr este objetivo, los nodos realizan los pasos detallados a continuación (la técnica ocupada está especificada en [12]).

Cada nodo i hace lo siguiente:

1. Elegir un polinomio en $\mathbb{Z}_N$: $f_\mathrm{i}(x)$, $f_\mathrm{i}(x)$ es de grado $t-1$, con $f_\mathrm{i}(0) = \mathrm{d}'_\mathrm{i}$.
2. Para $j = 1 \ldots n$
   - Calcular $f_{\mathrm{i},j} = f_\mathrm{i}(j) \mod N$
   - Enviarle el mensaje $\{f_{\mathrm{i},j}\}$ al nodo j.
3. Recibir los n mensajes $\{f_{j,\mathrm{i}}\}$, para $j = 1 \ldots n$ (mensajes enviados en el paso anterior desde los otros nodos y el mismo nodo i)
4. Calcular:
$$\mathrm{d}_\mathrm{i} = \sum_{j=1}^{n} f_{j,\mathrm{i}} \mod N$$

Notar que los d_i son valores de un polinomio $F(x) = \sum_{\mathrm{i}=1}^{n} f_\mathrm{i}(x)$ de grado $t-1$. Teniendo esto en cuenta (y que los identificadores de los nodos son conocidos), se necesitarían t claves d_i para poder interpolar el polinomio y recuperar d (la clave privada que está distribuida).

Cabe mencionar que cuando se están usando las claves de los nodos para realizar operaciones privadas, se ocupa una operación especial (llamada *join*) para unir los resultados parciales de los nodos que entregaron una respuesta. El *join* es capaz de generar un resultado total a partir de los resultados parciales sin exponer ni d ni los d_i.

De esta manera, al final de este paso, cada nodo conoce uno de los valores d_i. Así, la clave privada queda compuesta por el conjunto de claves $\{\mathrm{d}_\mathrm{i}\}_{\mathrm{i}=1}^{n}$, distribuido entre los nodos, y el módulo N, que queda público.

3.3. Tiempo de Ejecución Aleatorizado del Algoritmo

Se pueden reconocer dos fases importantes en el algoritmo: la primera consiste en encontrar un N que sea producto de dos primos y que tenga el tamaño adecuado; la segunda es generar las claves privadas d_i.

Una vez que se encuentra un N adecuado, generar las claves privadas es directo y toma un tiempo marginal respecto a la ejecución total del algoritmo. Sin embargo, la primera fase toma un tiempo aleatorizado. La razón de esto es que el algoritmo debe encontrar dos primos de manera simultánea:

1. Los nodos eligen sus p_i y q_i. Al hacer esto, eligen de manera implícita a p y q.

2. Se calcula $N = p \cdot q$ y se comprueba que sea el producto de dos primos.

Debido al teorema de los números primos[7], la probabilidad de que un número de b bits sea primo es aproximadamente:

$$\frac{1}{b \cdot \ln 2}$$

Consecuentemente, la probabilidad de encontrar dos primos de b bits de manera simultánea es alderedor de:

$$\left(\frac{1}{b \cdot \ln 2}\right)^2$$

Estos resultados están explicados en más detalle en [6].

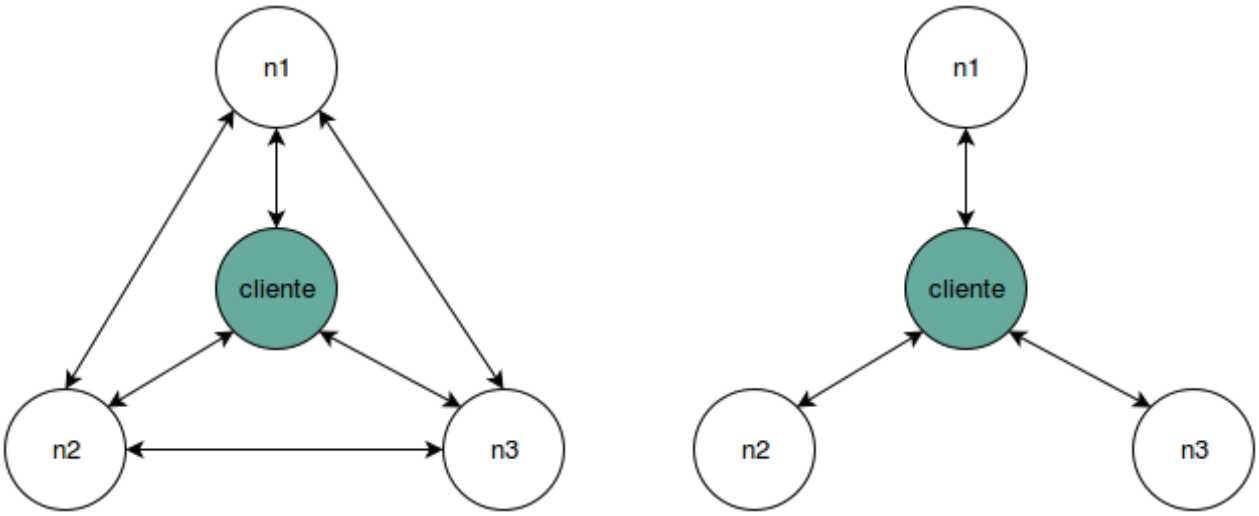
Dado lo que se ha dicho, en promedio hay que hacer $(b \ln 2)^2$ elecciones de p y q para encontrar un módulo N de $2b$ bits. Para un N de 1024 bits, $b = 512$, la cantidad de intentos promedio debería ser 125948.

Esto representa un deterioro cuadrático de rendimiento respecto a generar las claves en un sólo computador (lo que toma $2 \cdot b \ln 2$ intentos en promedio).

3.4. Capa de Comunicación

Como se pudo ver en la Sección 3.2, el algoritmo requiere que los nodos pueden comunicarse entre sí. En particular, cada nodo debe ser capaz de mandarle un mensaje a cualquier otro nodo, mandarle un mensaje a todos los nodos (*broadcast*), y recibir correctamente los mensajes que le lleguen.

Además, al comienzo y al final del algoritmo, los nodos deben comunicarse con el cliente: para recibir la solicitud para comenzar el algoritmo, para entregar el resultado final, y reportar que la ejecución del algoritmo terminó. La opción más obvia es usar un esquema completo (*clique*) como el que se muestra en la Figura 3.3a. El problema de este modelo es que cada participante debe mantener conexiones con cada uno de los otros participantes. Además de ser complejo en términos de implementación, este esquema implica que los nodos manejen más información de la que deberían (ya que deben conocer los identificadores de todos los nodos, y no sólo del cliente).



(a) Esquema Completo (b) Esquema Central

Figura 3.3: Esquemas de Comunicación

La otra opción es que toda la comunicación pase por el cliente. Así, el cliente actúa como un *router* para los nodos. Un ejemplo de este esquema central se puede ver en la Figura 3.3b. Con este modelo, se pierde ligeramente el carácter distribuido del sistema, sin embargo, la comunicación es considerablemente más simple y los nodos no saben cómo acceder directamente a sus pares, lo cual hace que el sistema sea más difícil de corromper. Además, el punto único de falla se puede remediar replicando la funcionalidad de comunicación que reside en el cliente.

Dado lo expuesto anteriormente, y para simplificar la implementación, se decidió ocupar la segunda opción, el esquema central.

3.5. Lenguaje de Programación: Erlang

El lenguaje de programación que se eligió para implementar el algoritmo fue Erlang[2]. La principal razón de esta decisión fue que Erlang es un lenguaje orientado a la concurrencia, lo que significa que posee primitivas para paralelizar y distribuir programas de manera muy sencilla.

La unidad de concurrencia en este lenguaje son los procesos. Usando la función `spawn`, se pueden lanzar procesos nuevos fácilmente que ejecutan las funciones que el programador elige. Dichos procesos pueden comunicarse entre sí a través del envío y recepción de mensajes, independientemente de si están en la misma máquina o en máquinas distintas. En general, los procesos se identifican con identificadores de procesos o *pids* (estos identificadores son los que se ocupan para enviar mensajes), sin embargo, los procesos también se pueden registrar con un nombre de proceso. De esta manera, cualquier otro proceso puede enviarle un mensaje al proceso registrado sin tener que conocer su *pid*.

Generalmente, lo que se usa para comunicar procesos o hilos concurrentes es memoria compartida. Sin embargo, hay muchos problemas asociados a ocupar memoria compartida que dificultan considerablemente el desarrollo, como *deadlocks* y hambruna. Como Erlang ocupa paso de mensajes para comunicar procesos, los problemas asociados a memoria compartida no existen y el desarrollo es más fluido.

Erlang corre sobre su propia máquina virtual: *BEAM*. La implementación de *BEAM* permite que dos procesos corriendo en instancias distintas de la máquina virtual se comuniquen de manera sencilla, casi como si pertenecieran a la misma máquina virtual. Además, las máquinas virtuales pueden estar en la misma máquina física o en máquinas físicas distintas (conectadas por la red). Esta característica es la que permite que Erlang sea un lenguaje distribuido.

Los archivos que contienen código Erlang ocupan la terminación `.erl`. Para ejecutar dichos archivos, primero hay que compilarlos usando el compilador del lenguaje, Erlang Compiler. El compilador compila los archivos `.erl` a archivos `.beam`. Los archivos `.beam` son los que se ejecutan con la máquina virtual *BEAM*.

Otras características de Erlang son que es un lenguaje funcional, dinámico, sin mutación (asignación única de valores en variables) y con evaluación estricta (evaluación *eager*).

Capítulo 4

Implementación

Luego de especificar el algoritmo implementado y tomar las decisiones de diseño mencionadas en el capítulo anterior, se procedió a implementar la solución. En este capítulo se describe cómo se llevó a cabo la implementación y algunos detalles particularles del software.

4.1. Módulo de Comunicación

El módulo de comunicación (`comm.erl`) es el que está encargado de administrar toda la comunicación que ocurre entre los nodos y el cliente. Como se dijo en la Sección 3.4, se eligió ocupar un modelo de comunicación central en el que toda la comunicación entre nodos es vía el cliente. Dado lo anterior, este módulo tiene funciones para que los nodos puedan enviar y recibir mensajes y para que el cliente pueda distribuir correctamente los mensajes entre los nodos.

Estas son las principales funciones contenidas en el módulo:

a) Cliente:

- `start_coord`: Se encarga de iniciar la comunicación en los nodos y el cliente. Genera un proceso con la función `coord_loop` en el cliente y un proceso con la función `node_loop` en cada nodo. De esta manera, todos los participantes quedan con un proceso que recibe y procesa mensajes.
- `coord_loop`: Un loop que recibe mensajes desde los nodos y los redirige al destinatario especificado en el mensaje.

b) Nodos:

- `send_to_node`: Una función que permite que un nodo le mande un mensaje al proceso de otro nodo. Para lograr esto, se debe proveer el identificador del nodo destinatario y el nombre del proceso destinatario debe ir dentro del mensaje (el proceso destinatario debe estar registrado con un nombre).

- `node_loop`: Un loop que va recibiendo los mensajes desde el cliente y los entrega al proceso indicado en el mensaje.
- `broadcast`: Una función que hace lo mismo que `send_to_node`, pero le envía el mensaje a todos los nodos (incluyendo el nodo mismo).

La Figura 4.1 explica cómo fluye la comunicación entre procesos y nodos.

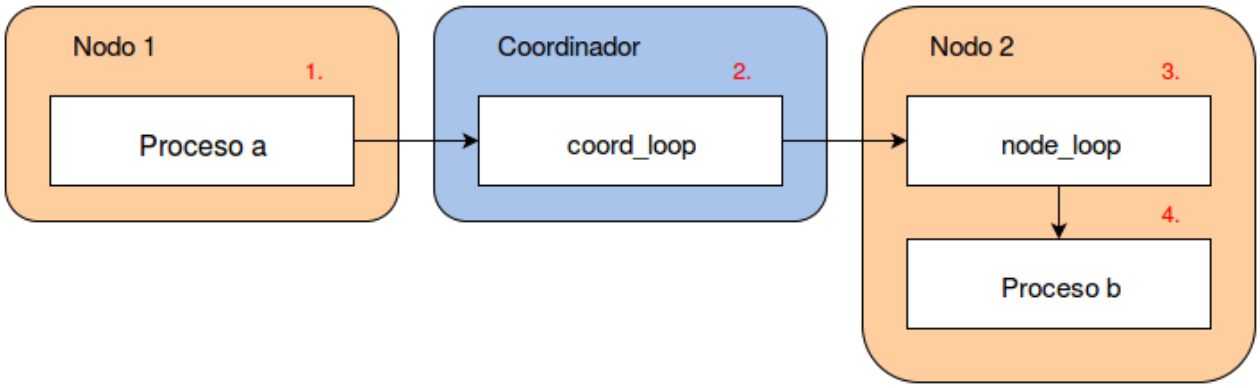


Figura 4.1: Flujo de Comunicación entre procesos

1. El `Proceso a` del nodo 1 le quiere enviar un mensaje al `Proceso b` del nodo 2. Ocupa la función `send_to_node`, especificando el nodo al que le quiere enviar el mensaje y qué mensaje quiere enviar. Además debe incluir el nombre del proceso destinatario dentro del mensaje.

2. La función le envía un mensaje al proceso que está corriendo `coord_loop` en el cliente.

3. El cliente le redirige el mensaje al proceso que está corriendo `node_loop` en el nodo destinatario.

4. El proceso de `node_loop` le envía el mensaje al proceso destinatario, que está indicado dentro del mensaje.

4.2. Módulos de Pasos

Aclaración sobre Nombres de Variables

Hasta ahora, se ha hecho referencia a las variables y parámetros propios del algoritmo utilizando el estilo *math mode* de LaTeX. Así, el número de nodos ha sido n, el módulo RSA generado ha sido N, el exponente público ha sido e, etc.

De ahora en adelante, el documento se referirá a las variables y parámetros propios de la implementación del algoritmo y no del algoritmo en sí. Para marcar la diferencia entre variables del algoritmo y variables de la implementación, las variables de la implementación estarán escritas con tipografía monoespaciada. Siguiendo los ejemplos anteriores, el número de nodos será `NNodes`, el módulo RSA será `N` y el exponente público será `E`. La variable

`NNodes` es la única que cambia su nombre, debido a la ambigüedad que se generaría con el módulo RSA.

Probablemente se notará que todos los nombres de variables propios de la implementación comienzan con letras mayúsculas. La razón de esto es que en Erlang las variables siempre empiezan con mayúscula. Esta condición es exigida por la sintáxis (y no por convención, como podría esperarse).

4.2.1. Pasos

Los pasos del algoritmo explicados en la Sección 3.2.3 están implementados en los módulos `step1.erl` - `step5.erl`. Como es de esperarse, cada uno de los módulos corresponde a uno de los pasos mencionados en orden de ejecución.

Se debe tomar en cuenta que todos los pasos conocen los valores que el algoritmo recibe como input: el tamaño en bits del módulo RSA a generar, la cantidad de nodos participantes, el exponente público dado y el parámetro umbral que tendrá el sistema.

Cabe destacar también que las funciones descritas a continuación son de naturaleza distribuida. Cuando una de las funciones siguientes se ejecuta, se debe ejectuar en todos los nodos del sistema a la vez. Los nodos trabajan cooperativamente para lograr resultados conjuntos utilizando estas funciones.

Dicho lo anterior, se presenta la descripción general de la función principal de cada módulo de paso.

- `step1.erl`: Elección de Candidatos de la Sección 3.2.3.

  Se encarga de generar los valores `Pi` y `Qi`. Para generar los valores de manera que cumplan las condiciones especificadas, ocupa una función auxiliar.
- `step2.erl`: Cálculo Distribuido del Módulo N de la Sección 3.2.3.

  Dados los valores `Pi` y `Qi`, calcula un candidato a módulo RSA `N`. Ocupa el patrón general de comunicación mencionado en la Sección 4.2.2 para compartir puntos de polinomios y valores.
- `step3.erl`: Test de Biprimalidad de la Sección 3.2.3.

  Dados los valores `Pi`, `Qi` y `N`, determina si el valor `N` dado es un biprimo o no. También ocupa el patrón de la Sección 4.2.2 para compartir valores.
- `step4.erl`: Generación del Conjunto Aditivo de d de la Sección 3.2.3.

  Dado un módulo RSA `N` y un exponente público `E`, genera un conjunto aditivo de claves privadas RSA. Al final de la ejecución, cada nodo queda con uno de los valores de las claves, `Di`. En la subsecciońn correspondiente de la Sección 3.2.3, se menciona que se debe realizar un ajuste a la clave `Di` del nodo con identificador 1. Para realizar ese ajuste, se ocupa un módulo auxiliar, `trial_decrypt.erl`
- `step5.erl`: Generación del Conjunto Umbral de d de la Sección 3.2.3.

  Dado un parámetro umbral `T` y el valor `Di` (perteneciente a un conjunto aditivo de una clave privada), genera otro valor `Dii`. El valor `Dii` es parte de un conjunto umbral (con

parámetro umbral `T`), que se genera a partir del conjunto aditivo.

4.2.2. Patrón General de Comunicación

Hay un patrón de comunicación que se repite en muchos de los pasos especificados en la Sección 3.2.3. Dicho patrón consiste en que cada nodo le debe enviar un fragmento de información particular al resto de los nodos, y a su vez recibir los fragmentos análogos que vienen de los otros nodos. Finalmente, todos los nodos terminan con la misma información y la pueden usar para calcular un resultado agregado. Dicho resultado puede ser una suma, un producto, una lista de puntos de un polinomio, el valor en un punto de un polinomio interpolado, etc.

Advertencia: El ejemplo que sigue a continuación incluye código en Erlang. Si bien el código no es difícil de entender, puede resultar un poco tedioso para algunos lectores y no es indispensable para entender el trabajo realizado. A pesar de lo anterior, se recomienda encarecidamente observar el código del ejemplo. Erlang es un lenguaje que puede llegar a ser muy elegante, y se pueden conseguir funcionalidades distribuidas complejas en unas pocas líneas (como lo demuestra el ejemplo).
En el caso de no entender el funcionamiento del lenguaje, se recomienda continuar leyendo desde la Sección 4.3.

En el siguiente ejemplo, un conjunto de nodos ocupará el patrón mencionado para obtener un resultado. Cada nodo eligirá un número al azar, lo compartirá con el resto de los nodos, recibirá los números generados por los otros nodos y finalmente calculará la suma de todos los números.

a)
```
example(NNodes, MyId)->
  NodesIds = lists:seq(1, NNodes),
  ExampleRecv = spawn(?MODULE, example_recv, [self(), NodesIds, 0]),
  register(example, ExampleRecv),
  spawn(?MODULE, example_send, [NodesIds, MyId]),
  receive
    {exampleResult, ExampleRecv, Result}->
      io:format("The result is:~w~n", [Result])
  end.
```

Esta es la función que cada nodo va a ejecutar, `example`. `MyId` es el identificador de este nodo en cuestión y `NNodes` es la cantidad total de nodos. Primero, la función genera la lista completa de los identificadores de los nodos (que van desde 1 hasta `NNodes`). Esta lista se pasa como parámetro en la siguiente línea. Luego, la función lanza un proceso que se encargará de recibir los mensajes provenientes de los otros nodos. Ese proceso encargado de recibir se registra con el nombre `example`. A continuación, la función lanza otro proceso encargado de generar un número al azar y mandárselo al resto de los nodos. Finalmente, la función queda a la espera de recibir un mensaje de resultado final proveniente del proceso que recibió los mensajes. Al recibir el resultado final, lo imprime en pantalla.

b)
```
example_send(NodeIds, MyId) ->
  RandNum = crypto:rand_uniform(1, 100),
  [comm:send_to_node(MyId, I, {example, {MyId, RandNum}}) || I <- NodeIds].
```

Esta es la función que se lanza en el segundo proceso de la función anterior. Primero, genera un número al azar entre 1 y 100. Luego, ocupa el módulo de comunicación para enviarle un mensaje a todos los nodos (incluyendo él mismo). Dentro del mensaje enviado, incluye el nombre del proceso destinatario, su propio id y el número generado. Al incluir esta información en el mensaje, el módulo de comunicación del nodo receptor sabe a qué proceso redirigir el mensaje y el proceso encargado de recibir el mensaje puede llevar la cuenta de cuáles nodos que le han enviado su número y cuáles no.

c)
```
example_recv(ParentPid, [], Acc)->
  ParentPid ! {exampleResult, self(), Acc};
example_recv(ParentPid, RemList, Acc) ->
  receive
    {example, {NodeId, Num}} ->
      NewRemList = lists:delete(NodeId, RemList),
      NewAcc = Num + Acc,
      example_recv(ParentPid, NewRemList, NewAcc)
  end.
```

Esta es la función que se ejecuta al lanzar el primer proceso de la función `example`. `ParentPid` es el identificador del proceso padre, `RemList` es la lista de ids de los nodos de los cuales no se ha recibido mensaje y `Acc` es el el acumulador parcial.

Hay dos casos en la función. El primero es que `RemList` sea una lista vacía. Si eso ocurre, no quedan mensajes por recibir y `Acc` es el resultado final. Dado eso, simplemente se reporta el resultado final al proceso padre. El otro caso, es que aún queden mensajes por recibir. Si ese es el caso, la función queda esperando a recibir un mensaje con el formato definido en la función `example_send`. Cuando lo recibe, actualiza la lista de ids `RemList` y el acumulador `Acc` y se llama recursivamente a sí misma.

d)
```
main()->
  Nodes = ?NODES,
  _CoordPid = comm:start_coord(Nodes),
  NNodes = length(Nodes),
  NodeIds = lists:seq(1, NNodes),
  [spawn(lists:nth(I, Nodes), ?MODULE, example, [NNodes, I]) || I <- NodeIds].
```

Esta es la función que se ejecuta para poder correr el ejemplo completo en todos los nodos. Se debe ejecutar desde el nodo que será el cliente. Esta función inicia la comunicación y luego lanza un proceso con la función `example` en cada uno de los nodos que participarán.

4.3. Módulo del Algoritmo Completo

El módulo del algoritmo completo (`full_alg.erl`) se encarga de componer las funciones principales de los módulos de los pasos (explicados anteriormente en la Sección 4.2.1) para obtener el flujo representado en la Figura 3.2.

Para lograr lo anterior, tiene dos funciones: una función auxiliar que se encarga de generar un `N` candidato y una función principal recursiva que implementa el *loop* de la Figura 3.2 y el comportamiento que sigue al *loop*.

- `gen_n`: Esta es una función auxiliar que se encarga de generar un candidato a módulo RSA: `N`. Para lograr este objetivo sigue los siguientes pasos:
  1. Primero, lanza un proceso que ejecuta la función principal del módulo `step1.erl` y se queda esperando el resultado de ese proceso. Recibe como resultado los candidatos a fragmentos de su nodo: `Pi` y `Qi`.
  2. Luego, al recibir el resultado obtenido del proceso anterior, lanza otro proceso que ejecuta la función principal de `step2.erl` y espera por su resultado. Recibe como resultado el módulo `N` calculado con el resultado anterior de todos los nodos.
  3. Finalmente, le manda los resultados obtenidos de ambos procesos a su proceso padre: `Pi`, `Qi` y `N`.
- `main`: Esta es la función que ejecuta el algoritmo completo. Es recursiva para poder simular el *loop* que hay en el algoritmo entre los tres primeros pasos (ver Figura 3.2).

  Esta función recibe como input valores análogos a los que recibe el algoritmo (Sección 3.2.1), el tamaño en bits del módulo RSA a generar, la cantidad de nodos participantes, el exponente público dado y el parámetro umbral que tendrá el sistema.

  Estos son los pasos que sigue la función:
  1. Lanza un proceso con la función auxiliar `gen_n` y se queda a la espera de un resultado. Recibe como resultado los valores generados de `Pi`, `Qi` y `N`.
  2. Teniendo el resultado anterior, lanza un proceso con la función principal de `step3.erl`, el test de biprimalidad, y se queda esperando su resultado.
  3. Si el proceso anterior retorna `false`, se llama recursivamente a sí misma para poder reiniciar todo el algoritmo.
  4. Si el proceso del test de biprimalidad retorna `true`, lanza un proceso con la función principal de `step4.erl` y se queda a la espera de un resultado. Recibe como resultado `Di`, una de las claves del conjunto aditivo generado.
  5. Finalmente, se lanza un proceso para ejecutar la función principal del paso del último módulo, `step5.erl`. Se recibe como resultado `Dii`, una de las claves del conjunto umbral generado.
  6. Se reporta el resultado final al cliente, el módulo generado `N` y el exponente público recibido originalmente, `E`.

  Además de hacer todo lo anterior, esta función cuenta la cantidad de llamados recursivos que se hace a sí misma. Con esto, se puede saber cuántos `N`s fue necesario generar antes de encontrar uno que fuera biprimo.

4.4. Módulos Locales

Los módulos locales contienen funciones auxiliares que no implican ningún tipo de distribución entre varios nodos. Dicho de otra forma, funciones que se computan localmente. Estos módulos se encuentran en la carpeta `src/local`. Se implementaron dos módulos de este tipo: uno de polinomios (`poly.erl`) y otro de funciones matemáticas (`math_helpers.erl`).

A continuación se describen ambos módulos:

4.4.1. Módulo de Polinomios

Como se pudo ver en la Sección 3.2, muchas de las operaciones necesarias para completar el algoritmo implican trabajar con polinomios. Dada esa necesidad, este módulo implementa todas las funcionalidades que se requieren relativas a polinomios.

Este módulo contiene funciones para realizar las siguientes tareas:

- Generar un polinomio al azar.
- Evaluar un polinomio en un punto.
- Interpolar un polinomio en un punto dado utilizando el método de Lagrange [1].

4.4.2. Módulo de Funciones Matemáticas

El algoritmo requiere muchas operaciones modulares (que no vienen implementadas en Erlang) y otras funciones matemáticas más complejas (como el cálculo del símbolo de Jacobi o de mínimo común múltiplo). Todas estas funciones de cálculos matemáticos fueron implementadas en este módulo.

A continuación, se presenta la lista completa de las funciones implementadas:

- Calcular inverso modular.
- Calcular potencia modular de manera eficiente (en tiempo logaritmo respecto al exponente).
- Calcular un *sharing aditivo* de un número. Recordar que un *sharing aditivo* de un número a es un conjunto $\{a_i\}_{i=1}^{n}$ que cumple que: $\sum_{i=1}^{n} a_i = a$. La función recibe el tamaño del conjunto a generar.
- Calcular el máximo común divisor de dos números (en tiempo logaritmo respecto al exponente).
- Calcular símbolo de Jacobi.
- Calcular potencia entera de manera eficiente (en tiempo logaritmo respecto al exponente).

4.5. Estructura del Código

La estructura del código se puede ver en la Figura 4.2.

El directorio `/src` contiene todos los módulos mencionados en las secciones anteriores de este capítulo. Dentro de `/src`, el subdirectorio `/local` contiene los módulos locales mencionados en la Sección 4.4 y el subdirectorio `/steps` los mencionados en la Sección 4.2.1.

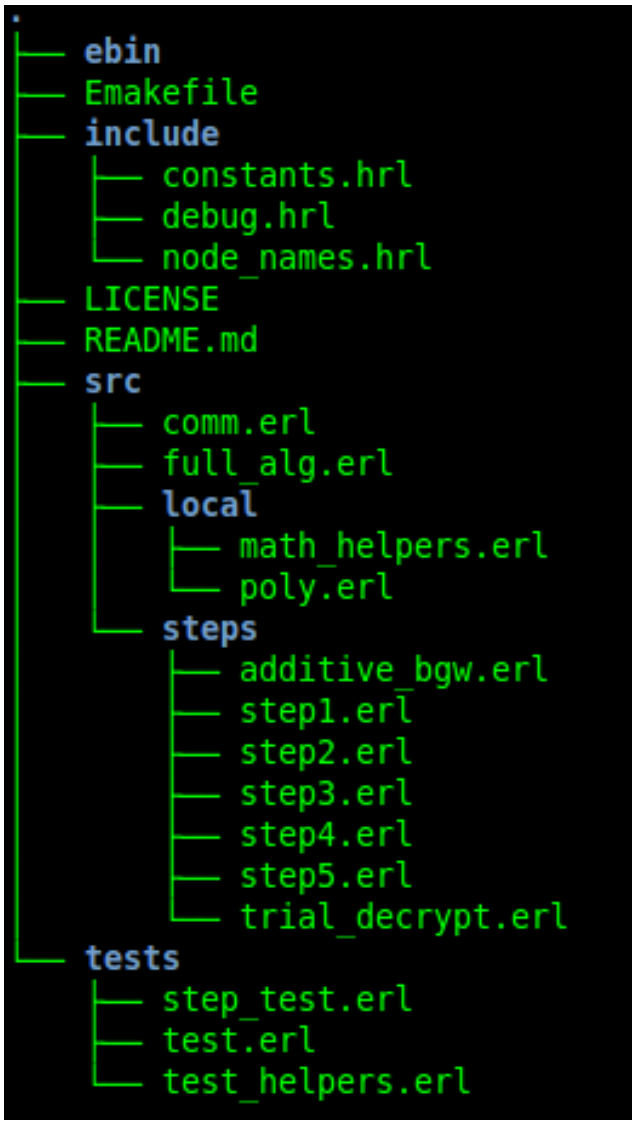


Figura 4.2: Estructura del Código

El directorio `/ebin` es el que contiene los archivos compilados `.beam`. Se mueven automáticamente al compilarse.

El directorio `/include` contiene archivos de *headers* de Erlang. Uno de ellos contiene una macro para activar el modo *debug* del código, otro guarda algunas de las constantes que se ocupan en el algoritmo y el otro es para poder definir los nombres de los nodos con los que

correrán los tests.

El directorio `/tests` contiene los test que se desarrollaron para el sistema, tanto para probar el algoritmo completo como para probar los pasos por separado.

4.6. Metodología

4.6.1. Control de Versiones

Se ocupó Git para mantener el control de veriones. Para esto, se creó un repositorio privado propiedad de Niclabs en GitHub para guardar el código. Dado que sólo había una persona trabajando en el código, se ocupó una sola rama, `master`. Todos los cambios se hicieron directamente a esa rama.

4.6.2. Entorno de Trabajo

La mayoría del código de desarrolló en un computador personal con Ubuntu 14.04 de 64 bits. Se usó la versión 19 de Erlang.

Como IDE de desarrollo se ocupó Jetbrains IntelliJ IDEA con el plugin de Erlang instalado.

4.6.3. Compilación

Dado que el proyecto creció bastante, se decidió automatizar la compilación. Para esto último, se escribió un *Emakefile* (muy similar a un archivo *make* para C).

Capítulo 5

Pruebas y Resultados

Con el objetivo de comprobar el correcto funcionamiento del algoritmo, se realizaron varias pruebas en distintos contextos. Este capítulo describe las pruebas realizadas y los resultados que se obtuvieron a partir de esas pruebas.

Además de comprobar que el algoritmo funcionaba correctamente, el otro objetivo de estas pruebas era obtener una estimación del tiempo que el algoritmo implementado demora en ejecutarse para distintos contextos y parámetros.

5.1. Descripción de las Pruebas

5.1.1. Pruebas Locales

Como se dijo en la Sección 3.5, se pueden tener varias instancias de la máquina virtual *BEAM* corriendo en la misma máquina física. En este tipo de prueba, cada participante del algoritmo (cliente y nodos), vive en una instancia distinta de la máquina virtual. De esta manera, se simula distribución usando paralelismo.

Este tipo de prueba es más simple del explicado en 5.1.2. Esto se debe a que son mucho más fáciles de configurar y se requiere sólo una máquina para poder correrlas. Dado lo anterior, este fue el tipo de prueba que se ocupó mientras se desarrollaba.

5.1.2. Pruebas Distribuidas

En este tipo de prueba también se tiene una instancia de la máquina virtual de Erlang por participante. Sin embargo, dichas instancias ya no viven en la misma máquina física, sino que cada una está en una máquina física distinta. Las máquinas físicas involucradas en la prueba están conectadas a través de una LAN (*Local Area Network*).

5.1.3. Verificaciones de Correctitud

Para poder verificar que algoritmo estaba implementado de manera correcta, se hicieron dos comprobaciones de consistencia y correctitud. Estas comprobaciones se realizaron para los dos tipos de pruebas descritos anteriormente en esta misma Sección (Pruebas Locales y Distribuidas).

1. **Consistencia en el valor de** `N`:

   Cada nodo participante retorna el valor de `N` que obtuvo. Se comprueba que el valor sea consistente en todos los nodos. La obtención del mismo valor por parte de todos los nodos comprueba que el algoritmo está bien implementado hasta al menos el segundo paso: Cálculo Distribuido del Módulo N de la Sección 3.2.3.

   Además, durante la fase de desarrollo, se hicieron otras pruebas con valores pequeños de `B` (ver Sección 5.2) en la que se factorizaba el valor obtenido de `N` para comprobar que fuera producto de dos primos. Los resultados de estas pruebas fueron exitosos y permitieron verificar que el test de biprimalidad estaba implementado correctamente.

   Lamentablemente, para las pruebas reportadas en esta sección, no se puede verificar que el `N` obtenido sea producto de dos primos. Esto se debe a que los valores de `N` son de aproximadamente 1024 bits y por ende factorizarlos se vuelve mucho más costoso (esta es, justamente, la gracia del algoritmo).

2. **Consistencia en el valor de** `D`: Para correr las pruebas, se ocupó una versión ligeramente distinta de la función `main` (Sección 4.3). Además de retornar los valores de `N` y `E`, dicha función modificada también retorna el valor de `Dii` obtenido por el nodo en el que se está ejecutando. De esta manera, se pueden obtener los valores `Dii` de todos los nodos.

   Como se explicó en la Sección 3.2.1 y en la subsección Generación del Conjunto Umbral de d (el exponente privado) de la Sección 3.2.3, el conjunto de valores `Dii` debe permitir realizar criptografía umbral. Dicho de manera simple, esto significa que con cualquier subconjunto de tamaño `T` (el parámetro umbral del sistema) del conjunto de claves `Dii` se debe poder obtener el valor `D`; donde `D` sería el valor del exponente privado en un sistema RSA no umbral con el mismo exponente público `E` y el mismo módulo `N` que este sistema umbral.

   Dado todo lo anterior, esta verificación consiste en calcular todos los conjuntos de tamaño `T` del conjunto de claves `Dii` y comprobar que se obtiene el mismo valor de `D` a partir de todos esos subconjuntos.

   El hecho de que todas las posibles combinaciones de valores `Dii` guarden el mismo valor de `D` demuestra que el último paso del algoritmo (Generación del Conjunto Umbral de d de la Sección 3.2.3) está implementado correctamente.

3. **Correctitud en el valor de** `D`: Una vez hecha la verificación anterior, se puede comprobar que el `D` obtenido por los nodos es efectivamente el inverso de `E` módulo `N`.

El hecho de que esa igualdad se cumpla comprueba que el penúltimo paso del algoritmo (Generación del Conjunto Aditivo de d de la Sección 3.2.3) está correctamente implementado.

5.2. Parámetros Utilizados

Como se explicó en la Sección 3.2, el algoritmo debe recibir varios parámetros de entrada para ejecutarse. La implementación tiene parámetros análogos a los que recibe el algoritmo:

- `E`: el exponente público que se ocupará.
- `B`: el tamaño en bits esperado del módulo RSA.
- `NNodes`: el número de nodos participantes en el sistema.
- `T`: el parámetro umbral del sistema.

Estos fueron los valores que se ocuparon en los dos tipos de prueba mencionados en la sección anterior:

- `E` = 65537
- `B` = 1024
- Valores de `NNodes` y `T` de acuerdo a la Tabla 5.1

NNodes	T
3	2
5	3

Tabla 5.1: Valores de `NNodes` y `T` utilizados al correr las prueba

5.3. *Hardware* Utilizado

5.3.1. Pruebas Locales

Las pruebas locales reportadas en la Sección 5.4 fueron ejecutadas en un computador de escritorio marca Dell con las siguientes características:

- Procesador: Intel® Core™ i5-2400 CPU @ 3.10GHz x 4
- Memoria: 16 GiB

5.3.2. Pruebas Distribuidas

Las pruebas distribuidas (para las que se necesitaron hasta seis computadores), fueron realizadas en computadores con el mismo procesador del computador mencionado en la sección

anterior (5.3.1) y con 8 o 16 GiB de memoria RAM.

5.4. Resultados Obtenidos

Antes que nada, cabe mencionar que todas las verificaciones explicadas en la Sección 5.1.3 resultaron correctas para todos las pruebas mencionadas a continuación. Dado lo anterior, se puede decir que el algoritmo fue correctamente implementado y que cumple con las especificaciones de la mencionadas en la Sección 3.2.1.

Como se explicó en la Sección 4.3, la función `main` no sólo ejecuta el algoritmo completo, si no que también cuenta la cantidad de "intentos" que el algoritmo debe realizar antes de obtener un resultado exitoso (donde un intento es la generación de un módulo `N`).

Ahora bien, como se explicó en la Sección 3.3, no es posible predecir con certeza cuánto tiempo tomará el algoritmo en ejecutarse. Dado lo anterior, las pruebas presentadas a continuación pretenden establecer una relación entre la cantidad de intentos y el tiempo de ejecución.

En ningún caso se puede esperar que los resultados presentados se repitan y reflejen de manera determinística el tiempo que toma el algoritmo. La cantidad de intentos siempre es aleatoria.

Pruebas Locales

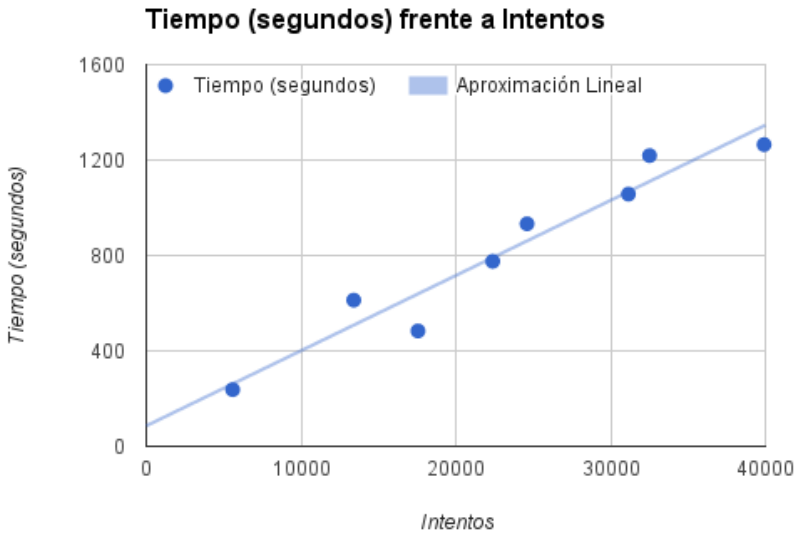


Figura 5.1: Resultados obtenidos para tres nodos y un cliente en modo local

Figura 5.2: Resultados obtenidos para cinco nodos y un cliente en modo local

Pruebas Distribuidas

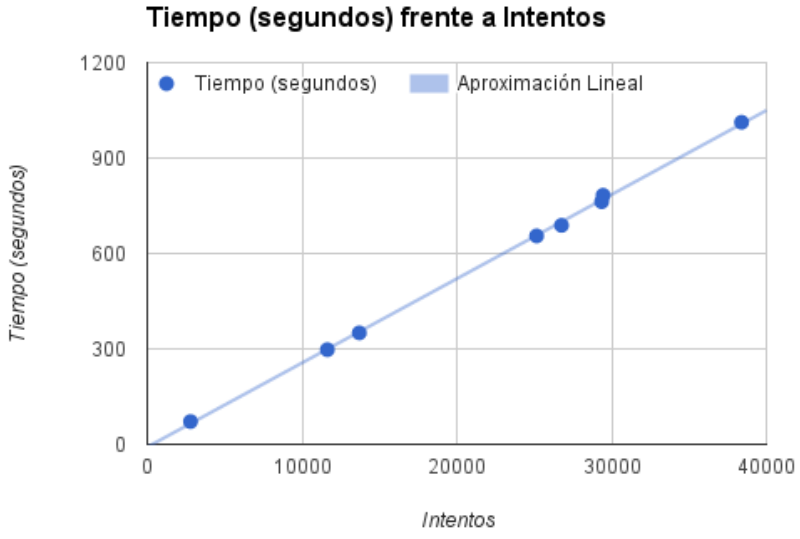


Figura 5.3: Resultados obtenidos para tres nodos y un cliente en modo distribuido

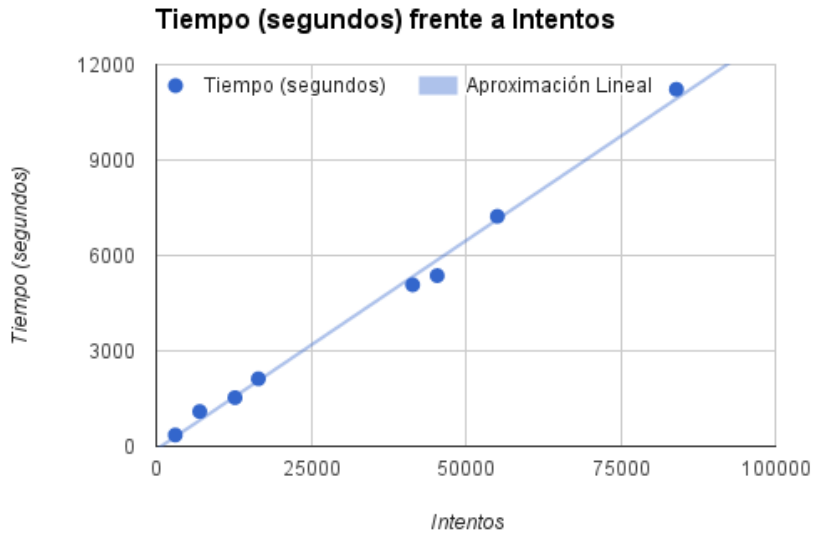


Figura 5.4: Resultados obtenidos para cinco nodos y un cliente en modo distribuido

Estos gráficos fueron generados con los datos que están en el Apéndice A.

Además, el algoritmo fue probado una vez con 3 nodos en modo local para generar un módulo de 2048 bits. Para generar el módulo, el algoritmo debió realizar aproximadamente 150000 intentos en 80 minutos

5.4.1. Relaciones entre Intentos y Tiempos

Se observará de los resultados que existe una relación lineal entre los intentos y el tiempo de ejecución. Dado lo anterior, se muestra también la recta que mejor aproxima los puntos de cada gráfico. En la Tabla 5.2, se pueden ver los valores de las pendientes para cada caso.

	Local		Distribuido	
	3 nodos	5 nodos	3 nodos	5 nodos
Pendiente	0.032 s	0.144 s	0.026 s	0.131 s

Tabla 5.2: Valores de las Pendientes de las Aproximaciones Lineales

El valor de la pendiente indica el costo en tiempo que toma realizar cada intento en cada prueba.

Se puede observar que tanto en modo local como distribuido, el costo de un intento aumenta considerablemente al pasar de 3 nodos a 5 nodos. Esto se debe a que la comunicación se complica al aumentar el número de nodos. Cuando el algoritmo indica que hay que mandarle mensajes a todos los nodos y recibir mensajes de todos los nodos, los nodos del sistema de

cinco nodos deben mandar y esperar dos mensajes más. Esto último, el esperar mensajes, es lo que ralentiza la ejecución.

El otro comportamiento que se puede observar es que el costo de un intento disminuye ligeramente al pasar de pruebas locales (una máquina física) a pruebas distribuidas (tantas máquinas físicas como participantes). La razón de esto es que en modo local los nodos deben compartir los recursos propios de la máquina física. Dentro de los recursos que se deben compartir están la entropía de la máquina física y la capacidad de procesamiento. Al pasar a modo distribuido, cada nodo dispone de sus propios recursos.

5.5. Comparación entre Esperanza Teórica de Intentos y Promedio Obtenido

Como se dijo en la sección 3.3, el tiempo del algoritmo es aleatorizado, por lo que no se puede predecir. A pesar de lo anterior, el teorema de los números primos ofrece una aproximación de la esperanza de intentos que se deben realizar antes de encontrar un módulo biprimo del tamaño requerido y continuar con el resto del algoritmo.

Como se explicó, se deben hacer en promedio $(b \ln 2)^2$ intentos para encontrar un número primo de b bits. La Tabla 5.3 muestra cuántos intentos se deben realizar en promedio para distintos tamaños de módulos.

Tamaño del Módulo	Tamaño de los Primos	Intentos Esperados
32	16	123
64	32	492
128	64	1968
256	128	7872
512	256	31487
1024	512	125948

Tabla 5.3: Intentos Esperados para Distintos Tamaños de Módulos

De acuerdo a la tabla anterior, se deberían hacer 126000 intentos *en promedio* para un módulo de 1024 bits. Sin embargo, en las figuras 5.1 - 5.4 se puede ver que no hubo una sola muestra que superara este supuesto promedio.

Dado lo anterior, se decidió medir la cantidad de intentos que el algoritmo demora en obtener un módulo biprimo para distintos tamaños de módulo. El promedio de intentos realizados para cada tamaño se comparó con el promedio esperado obtenido con el teorema de los números primos. El algoritmo fue ejecutado cinco veces para cada tamaño de módulo. El siguiente gráfico muestra el resultado:

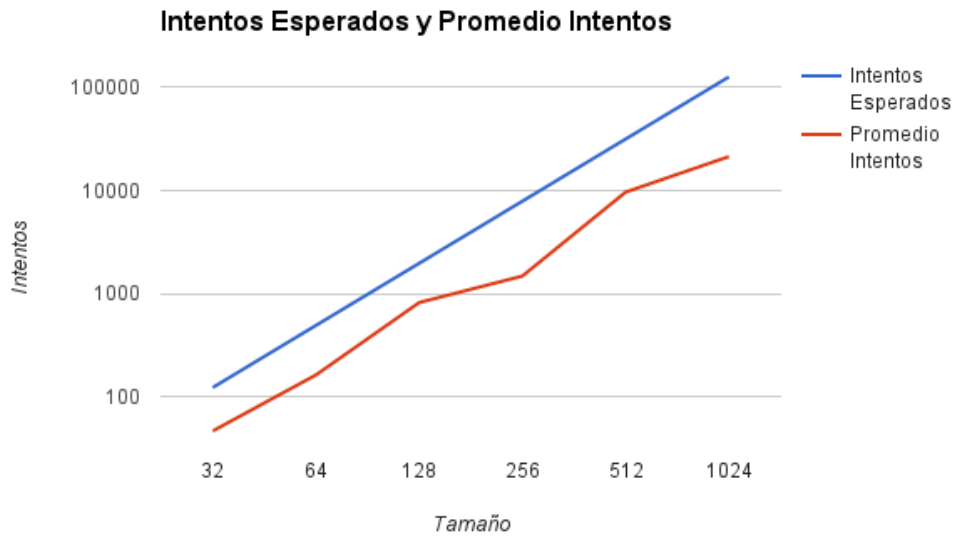


Figura 5.5: Comparación entre Promedio Obtenido y Promedio Esperado

Se puede observar que el promedio obtenido empíricamente es mucho menor al promedio esperado, para todos los tamaños de módulos probados (los datos utilizados para generar el gráfico están en el Apéndice A).

Capítulo 6

Conclusiones

Como se dijo al comienzo de este documento, se han propuesto sistemas de criptografía umbral RSA. Sin embargo, muchos de esos sistemas deben disponer de un actor confiable que sea capaz de generar y distribuir las claves. Esto representa una vulnerabilidad, ya que dicho actor podría ser atacado, revelando información que vulneraría el sistema. Así, los sistemas antes mencionados son frágiles en las etapas iniciales de generación y distribución de claves.

El algoritmo distribuido propuesto por D. Boneh y M. Franklin permite generar las claves RSA de manera distribuida, sin requerir un generador y distribuidor confiable. En este algoritmo, los nodos que se encargan distribuidamente de la operación privada son los mismos encargados de generar las claves de manera conjunta. Además, el algoritmo funciona de manera tal que ningún nodo puede obtener información concerniente a las claves de los otros nodos o a la factorización del módulo RSA.

La implementación del algoritmo mencionado fue el objeto del presente trabajo. Esta tarea fue lograda de manera exitosa, ya que se comprobó que funciona de la manera esperada.

Se explicó (ver sección 3.3) que no se puede predecir con certeza cuál será el tiempo de ejecución del algoritmo, ya que el tiempo de ejecución depende directamente de la cantidad de intentos que se deben realizar para encontrar un módulo biprimo. A pesar de lo anterior, el promedio de intentos fue mucho menor a la esperanza teórica de intentos en las mediciones realizadas.

Dado que la implementación puede mejorar y crecer mucho, (ver 6.1) la modularidad del código permitirá modificar, quitar y agregar elementos de manera sencilla.

6.1. Trabajo Futuro

Entre las mejoras que se le podrían implementar al sistema en el futuro están:

- **Capa de Seguridad:**

  Para que el sistema se pueda correr en Internet, se debe implementar primero una capa de seguridad que cifre y autentifique los mensajes. Además, el cliente es capaz de ver los mensajes que redirige entre los nodos. Lo ideal sería que todos los nodos ocuparan encriptación simétrica (compartiendo una clave) para ocultar sus mensajes del cliente.
- **Optimizaciones:**

  En [6], se propusieron algunas optimizaciones que permiten aumentar la probabilidad de elegir módulos biprimos y detectar que un módulo no es biprimo de manera temprana. Implementar estas optimizaciones reduciría el tiempo promedio de ejecución.
- **Esquemas de Comunicación:**

  La cantidad de mensajes enviados se podría reducir a la mitad si se implementara un sistema de comunicación completo (ver sección 3.4). Esta mejora podría representar un ahorro importante en el tiempo de ejecución.

Capítulo 7

Bibliografía

[1] Polinomios de Lagrange. `https://en.wikipedia.org/wiki/Lagrange_polynomial`. Accessed: 2016-10-3.

[2] Página oficial de Erlang. `http://www.erlang.org`. Accessed: 2016-08-09.

[3] *Optimal Asymmetric Encryption Padding (OAEP)*. `https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding`. Accessed: 2016-10-3.

[4] E. Bach and J. Shallit. In *Algorithmic Number Theory, Vol. 1: Efficient Algorithms.*, pages 443–444. The MIT Press, 1996.

[5] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault tolerant distributed computation. In *Proc. STOC*, pages 1–10, 1988.

[6] Dan Boneh and Matthew Franklin. Efficient generation of shared RSA keys. *Journal of the ACM (JACM)*, 48(4):702–722, July 2001.

[7] R. Crandall and C. Pomerance. *Prime Numbers: A Computational Perspective, 2nd ed.* Springer-Verlag, 2005.

[8] Jonathan Katz and Yehuda Lindell. Digital signature schemes. In *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*, pages 439–486. Chapman & Hall/CRC, 2007.

[9] Jonathan Katz and Yehuda Lindell. Public-key encryption. In *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*, pages 375–438. Chapman & Hall/CRC, 2007.

[10] Jonathan Katz and Yehuda Lindell. Secret sharing and threshold encryption. In *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*, pages 501–506. Chapman & Hall/CRC, 2007.

[11] Michael Malkin, Thomas Wu, and Dan Boneh. Experimenting with shared generation

of RSA keys. In *Proc. NDSS*, pages 43–56, 1999.

[12] T. Rabin. A simplified approach to threshold and proactive RSA. In *Advances in Cryptology — CRYPTO*, pages 89–104, 1998.

[13] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[14] Victor Shoup. Practical threshold signatures. In *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'00, pages 207–220, Berlin, Heidelberg, 2000. Springer-Verlag.

[15] William Stallings. Public-key cryptography and RSA. In *Cryptography and Network Security: Principles and Practice*, pages 253–285. Pearson Education, 3rd edition, 2002.

Apéndice A

Tablas de Resultados Completos

La medición se realizó en microsegundos. Los tiempos en segundos fueron calculados para ocuparlos en los gráficos.

Tabla A.1: Resultados obtenidos para tres nodos y un cliente en modo local

Intentos	Tiempo (s)	Tiempo (μs)
5584	238	237627376
13404	613	612732654
17556	484	483623449
22385	776	775554969
24602	933	932806523
31152	1057	1057269147
32517	1219	1218856530
39916	1265	1264886523

Tabla A.2: Resultados obtenidos para cinco nodos y un cliente en modo local

Intentos	Tiempo (s)	Tiempo (μs)
1932	228	228196957
2427	280	279530526
2868	333	332832752
3745	420	419552840
4936	676	676107116
19462	2417	2417454706
36612	4105	4105160998
45355	5237	5237022013

Tabla A.3: Resultados obtenidos para tres nodos y un cliente en modo distribuido

Intentos	Tiempo (s)	Tiempo (μs)
2795	72	71642472
11635	298	297637771
13700	350	350417992
25155	655	655299951
26765	689	688517609
29359	763	762839291
29451	783	783023482
38397	1013	1012579383

Tabla A.4: Resultados obtenidos para cinco nodos y un cliente en modo distribuido

Intentos	Tiempo (s)	Tiempo (μs)
3073	355	355291619
7022	1095	1095059504
12721	1531	1531067195
16495	2124	2124142846
41414	5077	5077027775
45377	5367	5366761465
55094	7231	7230584128
84006	11226	11225538456

Tabla A.5: Comparación entre promedio de intentos esperado y promedio de intentos obtenidos. El promedio obtenido fue calculado con 5 ejecuciones

Tamaño	Intentos Esperados	Promedio Intentos	Desviación	Máximo
32	123	47	29	87
64	492	163	123	320
128	1968	819	777	2053
256	7872	1479	1316	3213
512	31487	9632	7119	20747
1024	125948	21285	12604	37132