Red social distribuida ‘Eagal’

Maximiliano Noseda[1], Julian Martinez[1], Diego García[1], Lucia Corleto[1], Rodrigo Calvo[1]

[1]Universidad Nacional de La Matanza,
Departamento de Ingeniería e Investigaciones Tecnológicas,
Florencio Varela 1903 - San Justo, Argentina
m@noseda.xyz, julymartinez94@gmail.com, diegoarielgarcia@gmail.com, lucorleto@gmail.com,
rodrigocalvo95@gmail.com

Resumen
El objetivo de la investigación es el desarrollo de la red social de Eagal donde cada persona usuaria de un Eagal tendrá un perfil; en ese perfil se cargarán sus datos de uso, preferencias e información de debug para el soporte de Eagal.
Lo curioso de esta red social es que no tendremos un servidor central que almacene, procese y distribuya la información sino que el procesamiento y almacenamiento estará distribuido en cada dispositivo celular asociado a un Eagal.

Palabras claves: MPI, Android, Social Network.

Introducción

Nuestra investigación está centrada en la posibilidad de lanzar de forma productiva una forma de solucionar la infraestructura de una red social, en este caso usando el ejemplo de nuestro dispositivo Eagal, distribuida; esto quiere decir que no es una infraestructura centralizada en un servidor o en algunos servidores; sino que cada cliente es a la vez servidor de otros clientes.
Lo que se observa hoy, en las redes sociales más usadas, es una arquitectura “descentralizada” en varios servidores. Esto quiere decir que hay una diferencia bien marcada entre cliente y servidor. Estos varios servidores que comentamos suelen tener una parte de la información total de la red social organizada por posibilidad de consulta; es decir, los datos de usuarios provenientes de Argentina estarán en los servidores de Argentina ya que, estadísticamente serán más consultados que en Rusia. Pero, si un usuario de Rusia necesita los datos de un perfil de Argentina la consulta tardará un poco más ya que el servidor de Rusia no tendrá esos datos y deberá consultar con sus pares quien la tiene almacenada para devolverla al usuario. De esta forma vemos un modelo donde se diferencian claramente clientes de servidores y los servidores tienen parcialidad de la información pero con la posibilidad de comunicarse entre ellos para resolver, a pesar de la penalidad en el tiempo, la solicitud de cualquier usuario. Lo que venimos a presentar, en esta oportunidad es una nueva arquitectura donde cada cliente es servidor “distribuido” al mismo tiempo; cada servidor almacena y procesa los datos de su cliente y de los clientes más frecuentemente consultados desde él mismo.

1 Desarrollo

En esta investigación donde se busca constituir una arquitectura completamente descentralizada sin servidor, se trabaja bajo el concepto de MPI.

MPI es un estándar de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores. Su principal característica es que no precisa de memoria compartida, por lo que es muy importante en la programación de sistemas distribuidos. Más información aquí.

El concepto de MPI no se respeta conceptualmente al 100% ya que este indica que "Con MPI el número de procesos requeridos se asigna antes de la ejecución del programa, y no se crean procesos adicionales mientras la aplicación se ejecuta" pero, en nuestro proyecto cada vez que un Eagal se sincroniza con un nuevo dispositivo Android y se crea un nuevo perfil de la plataforma se suma un nuevo nodo (para el caso proceso) a la red social.

Lo que se busca con esta investigación es una arquitectura donde el desarrollador de la red social no tenga que invertir en comprar o alquilar servidores e infraestructura para mantener su sistema funcionando; sino que, la misma aplicación que se distribuye para el uso de la red social o lo que se puede llamar 'cliente' cumpla sumando todo el conjunto de usuarios con el 'servidor'.

Para entrar más en detalle, cada aplicación conectada con Eagal recopilará datos del perfil del usuario. Se incluyen datos como "Horas a las que se programan las alarmas", "Canciones o sonidos elegidos para sonar", "De qué ciudades le interesa al usuario conocer el clima", "La temperatura y humedad local del usuario", "Niveles de batería de su dispositivo", "Nivel de brillo del mismo", "Volumen", datos técnicos de funcionamiento, etc. Estos datos son empaquetados en el "Perfil" del usuario de Eagal y son "publicados" en la red social para que otros perfiles puedan consultarlos.

El proceso de publicación es donde está el cambio de paradigma; en una red social de servidores distribuidos la información se conforma en el formato deseado y se la envía al servidor con el que estamos trabajando para su almacenamiento y su distribución. Pero, en el modelo "Eagal" esto cambia ya que no hay uno o varios servidores centrales que almacenen la información, sino que cada dispositivo procesa y guarda en su base de datos local la información que generó pero, además, le envía la actualización a todos los nodos que son del "vecindario" de este; quiere decir con los que intercambia información ya que el usuario decide consultar esos perfiles.

En el caso de que tengamos un nuevo perfil este hará una suerte de "multicast" informando a un buen % de otros perfiles de su existencia. Cuando el perfil comience a relacionarse con otros el % con el que el perfil origen se comunicaba se va reduciendo hasta que se llega a un óptimo teórico donde los perfiles o nodos comunicados intercambian información solo con los que sí deben hacerlo y no inundan a los otros nodos con información irrelevante para ellos.

Ya se empieza a notar que se forma un concepto de "vecindad". La vecindad es un grupo de uno o más nodos que intercambian información entre ellos de forma regular. Si lo vemos desde una óptica que hoy existe mis nodos vecinos serían mis amigos en Facebook. Claro está que un nodo estará en varias vecindades, ya que cada nodo tendrá su vecindad que son aquellos nodos con los que se relaciona y aquellos con los que se relaciona a su vez intercambiarán información con otros y esos otros con otros formando una suerte de "grafo" de perfiles de Eagal.

Es en esa vecindad que se basa este modelo ya que en el caso de que un nodo necesite un dato y no lo tenga en su base local lo consultará con sus nodos vecinos (haciendo un ranking de más afines a menos afines) si aquellos nodos vecinos no tienen el dato tampoco se podrá consultar con los vecinos de ellos y así hasta lograr la información.

Puede considerarse el hecho de que sí el dato está muy lejos en "distancia de nodos" puede llegar a ser muy alto el tiempo de espera. Pero esta situación podría darse en un % muy chico del número total de consultas ya que se considera que la mayor parte de ellas se realizará en la vecindad del nodo.

En el caso de que se produzca una actualización en el perfil como puede ser una nueva canción añadida a Eagal el nodo guardará el cambio en su base local y lo enviará a toda su vecindad para que esta ya pueda visualizarlo. Esto quiere decir que la vecindad se mantiene actualizada; cada cambio en un nodo disparará la actualización en sus vecinos por lo que cada nodo tendrá la última versión de los perfiles de sus nodos vecinos.
Con ésto último se puede apreciar algo: las consultas entre nodos no se harán para pedir "la ultima versión" del perfil de X nodo ya que ante cualquier cambio los nodos vecinos recibirán el "update". Por lo que si un nodo debe mostrar información de un perfil de un nodo vecino ya tendrá la versión más actualizada del mismo en su base local. Una búsqueda en nodos se produciría cuando el perfil solicita una búsqueda de un perfil que el nodo no conoce y deba salir a buscarlo en sus vecinos y, en todo caso, en los vecinos de sus vecinos.
De todas maneras es preciso mantener una vecindad acotada; es decir, solo con los nodos que realmente tenemos contacto ya que si la vecindad es muy grande al momento de enviar un update se produciría una cantidad de mensajes innecesarios ya que a muchos de los nodos que le llega el dato realmente no lo necesitan. Esto ocasiona dos problemas: 1- uso innecesario de red por mensajes irrelevantes. 2- base local de nodos con tamaño excesivo al necesario.
Es por esto que se plantea una política de tamaño de vecindad; se deben tener definidas una serie de reglas que indiquen cuando un nodo deja de ser vecino. El caso contrarío es más fácil; para entrar a una vecindad se necesita, solamente, que un nodo consulte sobre un nodo desconocido y "descargue" su información; en ese momento pasan a ser vecinos y a intercambiar actualizaciones. Pero, lo complicado es cuando dejan de serlo; se plantea que si dos perfiles no se han "interesado" entre sí por más de diez días dejan de ser vecinos hasta que alguno consulte por el otro y se realice una búsqueda entre nodos. Cuando hablamos de nodos interesados queremos decir que un perfil no ha visualizado otro en cierta cantidad de días; ese control es local ya que al ser vecinos siempre tienen la información actualizada localmente; pero si el nodo nota que localmente no se ha consultado cierto perfil en X cantidad de días se produce el borrado de la vecindad del nodo del perfil que no ha sido consultado.
Cada nodo se comporta como un "proceso" autónomo dentro de la red de Eagal comunicándose los unos con los otros a través de mensajes como plantea el estándar MPI.
Una ventaja natural de este modelo es que no es tan dependiente de internet como la arquitectura actual de servidores distribuidos. Internet si es necesario para descargar siempre las últimas versiones de los perfiles de nodos vecinos o avisarles de un cambio en nuestro perfil. Pero, si internet estuvo activo hasta cierta hora tenemos la última versión de cada perfil hasta esa hora por lo que si internet dejara de funcionar nuestra aplicación nos seguiría mostrarnos información de los perfiles que nos interesan. De forma análoga se podría generar más contenido a nuestro perfil quedando almacenado en la base local. Al volver internet se produciría la sincronización periódica que mantiene las bases locales de los nodos vecinos actualizadas pero con un poco más de info a subir o a bajar por el tiempo que no hubo conexión.
Partimos de la base de que la aplicación Eagal que implica el dispositivo y la aplicación para Android trabajará con una red social que es donde se implementará esta solución
La integración de esta solución en la aplicación Eagal cumplirá el rol de "base de datos" y "medio de acceso a los datos". Esto quiere decir que la red social de Eagal tomará su información de la base local del dispositivo y también de sus nodos vecinos por lo que se convierte en el proveedor de datos del sistema y a la vez que es el medio de almacenamiento.

2 Explicación del algoritmo.

```
Class Nodo {
    //Ejemplo de búsqueda de un perfil
    Function BuscarPerfil(IdentificadorDePerfil) {
        If(BaseDeDatosLocal.Existe(IdentificadorDePerfil))
            Return BaseDeDatosLocal.BuscarYDevolver(IdentificadorDePerfil);

        //Si no esta localmente consulto a todos mis vecinos.
        //En el caso que esta funcion se ejecute porque soy un nodo vecino de un nodo
        //que busca un perfil y no tengo el dato en mi base local busco el perfil
        //en mis vecinos. Se forma una "red" de búsqueda.
        If(Vecindad.TieneVecinos())
            Return Vecindad.BuscarEnVecinos(IdentificadorDePerfil);
    }

    //Ejemplo envío una actualización de mi perfil a mis vecinos
    Function EnviarActualizacionAMisVecinos(Perfil) {
        If(Vecindad.TieneVecinos())
            Vecindad.EnviarActualizacionAVecinos(Perfil);
        Return OK;
    }

    //Ejemplo me envían una actualización de perfil
    Function GuardarActu(Perfil) {
        BaseDeDatosLocal.Guardar(Perfil);
        Return OK;
    }
}
```

```
Class Vecindad {
    Lista<Nodos> Vecinos;

    Function TieneVecinos(){
        Return Vecinos.Count() > 0;
    }

    Function BuscarEnVecinos(IdentificadorDePerfil){
        Declare  TaskId = 1; //Búsqueda de perfil en el nodo.
        return MPI.Do(Vecinos, TaskId, IdentificadorDePerfil);
    }
    Function EnviarActualizacionAVecinos(Perfil){
        Declare  TaskId = 2; //Enviar mi actualización de perfil al nodo.
        return MPI.Do(Vecinos, TaskId, Perfil);
    }
}
```

```
Class MPI {
    //Esta función enviará un mensaje broadcast a todos mis nodos vecinos
    //Con 2 datos: El id de tarea que tienen que hacer y los argumentos de esa tarea.
    //Este broadcast es recibido por un thread paralelo que corre en cada thread y
    //espera que le lleguen esas solicitudes. Si el rank es 0 soy el nodo que envio
    //el pedido y lo ignoro.
    //Una vez enviado hago un "Reduction" para aguardar la respuesta de los nodos.
    Function Do(Vecinos, TaskId, Args){
        Declare respuesta;

        Declare rank;
        Declare data = { taskId: TaskId, args: Args};
        Declare root = 0;
        Declare result;

        MPI_Init(Vecinos);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        //Envi el mensaje a todos
        MPI_Bcast(&data, 1, MPI_INT, root, MPI_COMM_WORLD);

        //Espero respuesta
        MPI_Reduce(&respuesta, &result, 1, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);

        if (rank != root) {
            respuesta = vacio;
        }

        MPI_Finalize();

        Return respuesta;
    }
}
```

```
//Desde el punto de vista del nodo que recibe estos mensajes
Class MPI_Thread extends Thread {
    //La idea es que este proceso esté corriendo de forma constante en background y,
    //reciba los broadcast enviados por un nodo que realiza una solicitud.
    //Si el nodo que envío soy yo no hago nada.
    //Sino realizo la operación pedida y devuelvo el resultado.
    Function Run() {
        Declare TaskId;
        Declare Args;
        Declare respuesta;
        do{
            Declare rank;
            Declare data;
            Declare root = 0;
            Declare result;

            MPI_Init();
            MPI_Comm_rank(MPI_COMM_WORLD, &rank);
            MPI_Bcast(&data, 1, MPI_INT, root, MPI_COMM_WORLD);

            if (rank != root) {
                TaskId = data.taskId;
                Args = data.args;

                swich(TaskId){
                    case 1: //Búsqueda de perfil en el nodo.
                        respuesta = Nodo.BuscarPerfil(Args);
                        break;
                    case 2: //Enviar mi actualización de perfil al nodo.
                        respuesta = Nodo.GuardarActu(Args);
                        break;
                }
            }

            //Envio la respuesta.
            MPI_Reduce(&respuesta, &result, 1, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
            MPI_Finalize();
        } while(true);
    }
}
```

3 Pruebas que pueden realizarse

La propuesta puede realizarse creando la funcionalidad en la aplicación y poniendo en funcionamiento algunos nodos Eagal observando cómo se van creando las vecindades y se va dando la comunicación entre ellos.
La implementación de la solución se realizaría utilizando las mismas herramientas que para el desarrollo de la aplicación: Visual Studio Code con: React Native, NodeJS, SDK de Android y la consola ADB para debug.
Para implementar MPI deberá utilizarse NDK ya que no existe una solución nativa para Android. Con el uso de NDK se podrá desarrollar en C++ utilizando las funciones de MPI.

4 Conclusiones

Podemos concluir que más allá de que el planteo realizado pueda llevarse a la realidad hay cosas por seguir pensando y por mejorar. Son situaciones como:

- ¿Qué sucede con los primeros nodos que se unen al a red? La pregunta parte de que los primeros dos nodos que existan deben ser vinculados ya que no tienen como encontrarse como consecuencia de la falta de un servidor central en el cual apoyarse.
- ¿Qué sucede cuando ninguno de mis nodos vecinos está online para resolver mi solicitud?
- ¿Cómo se manejan las cuentas que son eliminadas o cuando la aplicación es desinstalada? Ya que deja de ser un nodo de la red de Eagal y no hay que tenerlo en cuenta para consultarlo.

A pesar de estas cuestiones que quedan pendientes de resolver vemos como una posibilidad a tener en cuenta para aplicaciones que no justifiquen el uso de un servidor centralizado. Esto reduciría costos de puesta en marcha y mantenimiento de la app. Serían los mismos clientes quienes se den servicio y mantengan la red.

5 Referencias

1. Kurt B.Ferreira, Scott Levy, Kevin Pedretti, Ryan E.Grant:Parallel Computing Volume 77, Pages 57-83 (2018).
2. Leila Bahria, Barbara Carminati, Elena Ferrari: Online Social Networks and Media Volume 6, June 2018, Pages 18-25 (2018).
3. Bo Yuana , Lu Liu, Nick Antonopoulos: Future Generation Computer Systems Volume 86, September 2018, Pages 775-791: Efficient service discovery in decentralized online social networks (2018).