Centro de Investigación y de Estudios
Avanzados del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Biblioteca para mantener la coherencia de datos compartidos en dispositivos móviles

Tesis que presenta

Genaro Saucedo Tejada

para obtener el Grado de

Maestro en Ciencias de la Computación

Directora de la Tesis

Dr. Sonia Guadalupe Mendoza Chapa

México, Distrito Federal **Diciembre 2011**

Resumen

En la presente tesis se trata el problema de mantener la coherencia de datos compartidos, que suelen ser utilizados por los sistemas que dan soporte a la colaboración entre personas. Concretamente, nuestro estudio se centra en aplicaciones de edición colaborativa que pueden ser ejecutadas en dispositivos móviles y que están diseñadas para dar soporte a una interacción de tipo cara a cara entre sus usuarios.

El problema resuelto consta de dos partes: 1) las cuestiones encontradas en la modificación concurrente de datos distribuidos en una red local y 2) las dificultades presentes en el desarrollo de sistemas colaborativos para dispositivos móviles. La primera parte se abordó por medio de un análisis de las técnicas para mantener la coherencia de datos compartidos en sistemas colaborativos. De entre dichas técnicas se seleccionó la de Transformación Operacional, ya que brinda buena velocidad de respuesta a las acciones de los usuarios. Dicha técnica fue analizada, incluyendo sus distintas variantes y su evolución histórica. Finalmente, se seleccionó un algoritmo en particular de entre los diversos algoritmos de Transformación Operacional publicados en la literatura científica. Para abordar la segunda parte del problema, se consideraron las dimensiones de la movilidad, las cuales permiten caracterizar los problemas enfrentados durante el desarrollo de sistemas para dispositivos móviles. De entre estas dimensiones de la movilidad, se identificó cuáles son las más significativas para los sistemas colaborativos y se propusieron soluciones a nivel diseño e implementación.

A partir de estas soluciones, se elaboró una biblioteca denominada CDCDM (Coherencia de Datos Compartidos en Dispositivos Móviles), la cual soporta el desarrollo de sistemas colaborativos que se ejecutan en dispositivos móviles y que enfrentan el problema de mantener la coherencia de datos accedidos de manera simultanea por varios usuarios. Los sistemas que se benefician de esta biblioteca son principalmente editores colaborativos.

Para probar dicha biblioteca fue desarrollada una aplicación, la cual es un editor de texto colaborativo que permitió identificar algunas ventajas y limitaciones de la plataforma de desarrollo utilizada, así como algunos inconvenientes encontrados en la construcción de sistemas colaborativos ejecutables en dispositivos móviles.

Abstract

This thesis deals with the problem of maintaining the consistency of shared data, which are usually employed in systems supporting the collaboration among people. Specifically, our study focuses on collaborative editing applications, which are able to be executed in mobile devices and are designed to provide support for a face-to-face interaction among its users.

The solved problem consists of two parts: 1) the issues resulting from the concurrent modification of shared data distributed in a local area network, and 2) the difficulties presented during the development of groupware systems for mobile devices. The former part is faced with an analysis of the proposed techniques to ensure data consistency in groupware systems. From these techniques, we selected the Operational Transformation technique, as it provides good response time to the users' actions. Such a technique has been analyzed, including its variants and historical evolution. Then, we chose a particular algorithm from the different Operational Transformation algorithms proposed in the scientific literature. To cope with the latter part of the problem, we considered the mobility dimensions, which allow to characterize the problems faced during the development of systems intended for mobile devices. From these mobility dimensions, we identified which are the most significant for groupware systems and then we proposed solutions at the design and implementation levels.

From these proposed solutions, we created a library called CDCDM (Coherencia de Datos Compartidos en Dispositivos Móviles), which supports the development of groupware systems that are able to be executed in mobile devices and that confront the matter of maintaining the consistency of data acceded in a simultaneous way by several users. The groupware systems being of benefit to this library are mainly editors.

In order to test such a library, we developed an application, which is a groupware text editor that allowed to identify some advantages and limitations of the used development platform as well as some disadvantages found in the construction of groupware system executable on mobile devices.

Agradecimientos

Agradezco al CONACyT por haberme brindado la beca que me permitió concluir mi maestría y a mi asesora Dr. Sonia Guadalupe Mendoza Chapa por el esfuerzo que realizó.

También agradezco a mis compañeros por los ratos divertidos que me brindaron y a Sofía Reza por su cordialidad.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En la Sección 1.1 se explica, de una forma general, los principales conceptos del contexto de investigación en el cual se desarrolla la presente tesis, así como la terminología más importante que será utilizada en el resto del documento. En la Sección 1.2 se describe el problema que se intenta solucionar o más concisamente, las nuevas oportunidades que se desea explorar. En la Sección 1.4 se da un breve resumen de los resultados obtenidos en la presente tesis. Finalmente, en la Sección 1.5, se describe la organización del documento.

1.1 Antecedentes

Las capacidades de un individuo son bastante limitadas, por esta razón las personas (y en general los seres vivos) forman grupos, logrando adquirir capacidades superiores a las del individuo. Cuando se forma un grupo de individuos con una meta u objetivo común, se dice que sus integrantes **colaboran** para lograr dicho objetivo. Sin las ventajas brindadas por la colaboración, la sociedad actual sería inconcebible.

Siendo una actividad tan importante y frecuente, la colaboración se vale de la tecnología y más recientemente de la tecnología de computadoras. El Trabajo Colaborativo Asistido por Computadora (TCAC) es un campo de investigación que estudia cómo las actividades colaborativas y su coordinación pueden ser apoyadas por medio de sistemas computacionales [Carstensen and Schmidt, 1999].

El término TCAC fue acuñado originalmente por Irene Greif y Paul M. Cashman en 1984, en un taller al que asistieron personas interesadas en utilizar la tecnología para ayudar a la gente en su trabajo [Grudin, 1994].

Algunas preguntas que intenta responder el TCAC son: ¿Qué caracteriza al trabajo colaborativo? ¿Cómo se puede modelar el trabajo colaborativo? ¿Qué prestaciones deben ser provistas por las computadoras?

El TCAC estudia los efectos psicológicos, sociales y organizacionales del uso de sistemas computacionales para la colaboración, así como las técnicas y herramientas necesarias para desarrollar dichos sistemas. Por esta razón, el cómputo colaborativo es un campo que integra psicólogos, sociólogos, antropólogos, además de científicos

de la computación.

Los sistemas computacionales, que son producto del TCAC, se denominan sistemas colaborativos y son objeto de algunas clasificaciones revisadas por Ellis et al. [Ellis et al., 1991]. Estás clasificaciones serán útiles en el contexto de la presente tesis.

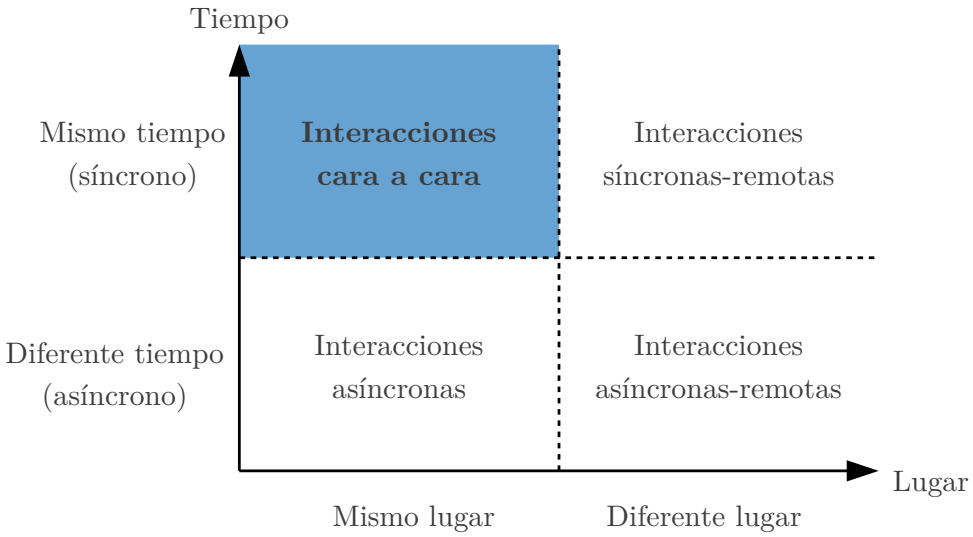


Figura 1.1: Topologías de los sistemas colaborativos

La primera clasificación considera dos dimensiones, lugar y tiempo, como se muestra en la figura 1.1.

La dimensión de lugar corresponde a la separación de los participantes de la sesión de trabajo, quienes pueden estar en el mismo lugar (colocalizados) o en diferentes lugares. Cuando los participantes están en el mismo lugar se pueden ver entre sí y hablar directamente, en consecuencia la comunicación es normalmente oral. En contraste con la situación anterior, cuando los participantes están en diferentes lugares, es necesario un soporte de comunicación, puesto que los participantes pueden estar inclusive en diferentes países gracias a Internet.

En la dimensión de tiempo, los sistemas pueden ser síncronos o asíncronos. Un sistema es síncrono si soporta la colaboración simultánea de los participantes, mientras que un sistema asíncrono permite a los participantes trabajar en diferentes momentos, i.e., no se requiere de otros participantes para trabajar.

Todo sistema colaborativo cae en al menos uno de los cuadrantes formados en la matriz de espacio-tiempo. Sin embargo, algunos pueden abarcar más de un cuadrante, e.g., Google Docs permite tanto la interacción síncrona-remota como la asíncrona-remota, gracias a la infraestructura provista por Google.

La siguiente clasificación se basa en la funcionalidad y algunas clases se intersectan:

- **Sistemas de mensajes**. Es el ejemplo más común, pues incluye al correo electrónico y a los sistemas de tabloides (*Bulletin Board Systems*).

- **Edición colaborativa**. Estos sistemas permiten a los miembros de un grupo componer y editar un documento o diagrama conjuntamente. Pueden ser síncronos, asíncronos o una combinación de ambos. Distintos sistemas permiten diferentes grados de concurrencia.

- **Apoyo para decisiones grupales y cuartos de juntas electrónicas**. Los sistemas de apoyo para decisiones grupales proveen prestaciones, basadas en computadora, para la exploración de problemas no estructurados en un ambiente grupal. Buscan mejorar la productividad acelerando el proceso de toma de decisiones o mejorando la calidad de dichas decisiones. Muchos de estos sistemas han sido implementados como cuartos de juntas electrónicos, que contienen múltiples terminales y despliegues públicos y que están conectados entre sí. Ejemplos de aplicaciones son la lluvia de ideas, la identificación y el análisis de protagonistas y el análisis de problemas.

- **Conferencias por computadora**. Se utiliza la computadora como medio de comunicación en diferentes formas. La primera es permitir a los participantes interactuar en la conferencia por medio del uso de alguna aplicación. La segunda, normalmente conocida como teleconferencia, consiste en brindar ayuda mediante telecomunicaciones. Finalmente, la tercera consiste en una combinación de las dos anteriores.

- **Sistemas de coordinación**. El problema de la coordinación es "la integración y el ajuste armonioso de los esfuerzos individuales, orientándolos hacia el logro de una meta final" [Singh, 1989]. Los sistemas colaborativos abordan este problema mostrando a los participantes acciones propias y acciones relevantes de otros participantes, dentro del contexto de la meta global. Además, estos sistemas pueden disparar acciones de los participantes al notificarles cambios de estado o alguna condición esperada.

Como se puede ver, las clases en las que se puede incluir un sistema colaborativo son variadas y dentro de la misma clase existen diversos dominios de aplicación.

1.2 Nuevas oportunidades

En años recientes se ha presentado un auge significativo en el desarrollo de dispositivos móviles. Este desarrollo se caracteriza por el aumento tanto de su capacidad como de su cantidad, lo cual nos brinda grandes oportunidades para desarrollar aplicaciones más demandantes y que puedan ser usadas por más personas.

Los dispositivos móviles actuales son relativamente potentes, e.g., un *smartphone* actual tiene mayor potencia de procesamiento que una computadora de escritorio de hace 10 años. También son accesibles cada vez a más personas, gracias a la disminución de costo.

A pesar del progreso alcanzado, los dispositivos móviles presentan ciertas características que complican el desarrollo de aplicaciones para estos. Las principales características que los distinguen son las siguientes: conectividad limitada o intermitente, capacidades (relativamente) acotadas, fuente de poder restringida, variedad de interfaces de usuario, proliferación de plataformas, etc. Estas características son llamadas las dimensiones de la movilidad de un sistema [B'far, 2004].

Como se puede ver, no son pocas las características a tener en cuenta para el desarrollo de una aplicación móvil exitosa. Posiblemente, por esta razón, las aplicaciones para dispositivos móviles no han crecido tanto como lo han hecho los dispositivos mismos.

La oportunidad que se busca explotar en la presente tesis es el desarrollo de aplicaciones móviles del área de TCAC, i.e., sistemas colaborativos móviles. Concretamente el problema a resolver es la falta o poca variedad de sistemas colaborativos síncronos móviles. La solución propuesta es desarrollar una biblioteca que facilite la creación de sistemas colaborativos síncronos móviles por medio de la técnica de control de concurrencia denominada Transformación Operacional, la cual es especialmente adecuada para sistemas colaborativos síncronos.

Solución propuesta

Para abordar los problemas que surgen de las dimensiones de la movilidad y lograr el desarrollo de sistemas colaborativos móviles, se plantea el desarrollo de una biblioteca que enfrente la mayor cantidad posible de dimensiones de la movilidad.

La forma en la que dicha biblioteca enfrenta a las distintas dimensiones se describe a continuación:

- **Conectividad limitada o intermitente**. Siendo la intermitencia de la conexión un problema común en las redes inalámbricas (propias de los dispositivos móviles), la biblioteca soporta "sesiones dinámicas", lo cual se refiere a que una sesión puede recibir nuevos participantes en todo momento y también prescindir de participantes que se desconecten abruptamente. Lo anterior debe ser manejado de forma transparente al usuario para que no perturbe o interrumpa su trabajo. Respecto a la limitación de ancho de banda, se consideran redes *ad-hoc* con tecnología Bluetooth, las cuales proveen un ancho de banda constante y sin costo monetario, a diferencia de la red GSM.

- **Capacidades limitadas**. Las capacidades limitadas de los dispositivos se mitigan mediante el uso de Java ME (Micro Edition), la cual es una plataforma enfocada a esta clase de dispositivos. Además, la biblioteca interfiere lo menos posible con el diseño global de las aplicaciones que la utilizan, gracias a su diseño de *toolkit*, facilitando el desarrollo de aplicaciones simples cuando el dispositivo objetivo así lo requiera.

- **Fuente de poder limitada**. Esta dimensión no es atacada fuertemente por los programas, sin embargo la propiedad de "sesiones dinámicas" permite a

los participantes desconectarse fácilmente de una sesión para ahorrar energía. Por otro lado, el hardware seleccionado para la conexión de red (Bluetooth) está diseñado para tener un consumo de energía bajo, como lo requieren los dispositivos móviles.

- **Variedad de interfaces de usuario**. La biblioteca por sí sola no contiene funcionalidad de interfaz de usuario, debido a que constituye un tema lo suficientemente amplio como para quedar fuera del alcance de esta tesis. Sin embargo, aunque esta dimensión no es enfrentada directamente, si se tomó en cuenta, pues la biblioteca se definió cómo genérica y no presenta ninguna limitación con respecto a cómo se desarrollan las aplicaciones que la utilizan.

- **Proliferación de plataformas**. Este es un problema importante en los dispositivos móviles, sin embargo la plataforma Java ME le hace frente de una manera muy eficaz, mediante el uso de una máquina virtual que logra abarcar una gran cantidad de dispositivos.

- **Conocimiento de la ubicación**. Esta dimensión se relaciona con ciertos dominios de aplicación y no se presenta como una característica de todas las aplicación móviles, por lo que no es abordada por la presente tesis.

- **Transacciones activas**. Las transacciones activas son aquellas iniciadas por el sistema, lo cual contrasta con la mayoría de las aplicaciones de escritorio que son pasivas y están en espera de acciones del usuario. Este tipo de transacciones son útiles en sistemas móviles pues, a diferencia de los sistemas de escritorio, el usuario no está atento al dispositivo móvil durante todo el tiempo que la aplicación se ejecuta. Este esquema de programación es fácilmente soportado por la plataforma Java ME, ya que ésta provee multi-hilos.

Aunque los dispositivos móviles definen fuertemente la estructura que debe adoptar la biblioteca, no hay que dejar de lado el objetivo principal de la biblioteca, que es el desarrollo de sistemas colaborativos. Como se explicó en la Sección 1.1. los sistemas colaborativos son susceptibles a ciertas clasificaciones, lo cual es importante en el contexto de la presente tesis, pues nos permite delimitar las funcionalidades esperadas de la biblioteca y de las aplicaciones que la utilizan.

En la matriz de lugar-tiempo (ver figura 1.1), la biblioteca propuesta se limita a dar soporte a la interacción síncrona en la dimensión de tiempo. Esta limitación se debe a que el soporte tanto de la interacción síncrona, como de la asíncrona, podría representar un alcance mayor al que se puede cubrir en una tesis de maestría, por tanto se optó por una sola de las dos.

En cuanto a la dimensión de lugar, la biblioteca está enfocada al trabajo colocalizado, debido a que se utiliza la tecnología Bluetooth. Está tecnología fue seleccionada porque presenta un bajo consumo de energía y porque permite hacer ciertas suposiciones respecto a la calidad de la red. El enfoque hacia el trabajo colocalizado también es reforzado por el hecho de que la biblioteca no ofrece prestaciones de comunicación

(audio, video, chat, etc.). Estas prestaciones pueden ser provistas de forma externa, por lo que el desarrollador de aplicaciones no las puede dar por hecho. Sin embargo, la biblioteca permite cambiar a TCP/IP como tecnología de red para soportar interacción síncrona-remota. Este tipo de interacción no es estudiado por la presente tesis, pues no se considera que ofrezca beneficios a una cantidad suficiente de usuarios.

Estando enfocados a la interacción local y limitados a la interacción síncrona, existen varios tipos de funcionalidades que la biblioteca podría soportar, los cuales son: edición colaborativa (síncronos), apoyo para decisiones grupales, conferencias por computadora (solo interacción, no telecomunicación) y sistemas de coordinación (síncronos). Sin embargo, su principal dominio de aplicación será los sistemas de edición colaborativa.

Las herramientas provistas al programador tienen la propiedad de ser generales, por lo tanto la biblioteca es útil en varios dominios de aplicación.

1.3 Transformación operacional

La técnica Transformación Operacional (OT), que permite mantener la coherencia de datos compartidos juega, un papel importante en la presente tesis. Esta técnica normalmente no es tomada en cuenta por los desarrolladores cuando se espera: 1) que haya poca concurrencia de usuarios y 2) que las redes de comunicación no presenten fallos. Sistemas con estas características pueden hacer uso de técnicas más sencillas para mantener la coherencia de datos. Sin embargo, en sistemas que soportan alto grado de concurrencia y que consideran la posibilidad de fallos en la red, la técnica de OT resulta muy útil, ya que es tolerante a retrasos en la red e inclusive a desconexiones y no restringe las acciones concurrentes de los usuarios.

Para explicar el funcionamiento de esta técnica hay que tener en cuenta tres conceptos fundamentales: operación, contexto y transformación.

Las **operaciones** constituyen la única forma de editar los datos compartidos, por lo tanto toda modificación de dichos datos se deberá expresar por medio de una o más operaciones, e.g., borrar la tercera letra y luego insertar una letra ‘e’.

El siguiente concepto es el de **contexto**. Cuando se generan operaciones, estas son pensadas por el usuario para ejecutarse sobre ciertos datos específicos, los cuales conforman el contexto de dichas operaciones. De hecho, las operaciones están ligadas al contexto en el que se originaron, por lo que si se ejecutan en un contexto diferente pueden causar efectos indeseados.

A manera de ilustración considere las operaciones del ejemplo anterior, i.e., borrar la tercera letra y luego insertar una letra ‘e’. Estas operaciones suponen que existen al menos tres letras, por eso, en dichas operaciones se menciona a la “tercera” letra. Sin embargo, si se intentara ejecutar estas operaciones en una cadena de una sola letra, se produciría un error al borrar la “tercera” letra, pues no existe. Este error ocurre porque el contexto de la operación es diferente a la cadena sobre la cual se intenta ejecutar la operación.

En un sistema colaborativo distribuido los contextos pueden divergir a raíz de

modificaciones concurrentes en presencia de un retardo de red. Un caso simple se da cuando un usuario inserta letras mientras que otro borra letras diferentes, antes de que sus computadoras terminen de comunicarse entre sí. Entonces, cuando las operaciones llegan desde la red, se encuentran con un contexto diferente a aquél en el que fueron generadas. Una operación encontrará una cadena con menos letras y la otra una cadena con más letras. Ante estos cambios de contexto, se necesita aplicar una **transformación**, la cual se define como el cambio que se necesita hacer a las operaciones para que al ejecutarse tengan el efecto deseado en los datos. Qué transformaciones hacer, en qué orden, cuándo si y cuándo no, son decisiones tomadas por un algoritmo de OT.

En el Capítulo 3 se explican algunos de los algoritmos de OT y en la Sección 5.4 se explican las transformaciones sobre el tipo de dato de cadenas de caracteres.

1.4 Resultados

Durante el desarrollo de esta tesis se obtuvieron algunos resultados experimentales, se analizaron las ventajas teóricas de algunos algoritmos y se hizo una implementación reutilizable. A continuación se describen brevemente los resultados principales:

- **Resultados de experimentación**: tras haber diseñado, implementado y probado la biblioteca objetivo de esta tesis se detectaron algunas ventajas y limitaciones inherentes a la tecnología actual de los dispositivos móviles, e.g., la limitación de los componentes gráficos estándar y la ventaja de las redes *ad-hoc*.
- **Análisis de algoritmos de OT**: se analizaron los distintos algoritmos de OT propuestos y se seleccionó el más adecuado para esta implementación.
- **Software de la biblioteca**: el producto final de la investigación y la implementación es la biblioteca para el soporte de la Coherencia de Datos Compartidos en Dispositivos Móviles (CDCDM). Esta biblioteca facilita el desarrollo de sistemas colaborativos móviles, principalmente de editores colaborativos.

1.5 Estructura de la tesis

En la figura 1.2 se listan los capítulos que integran la presente tesis. El Capítulo 2 describe el estado del arte actual en lo referente a dispositivos móviles, coherencia de datos y sistemas colaborativos móviles. El Capítulo 3 conforma al marco teórico, el cual está dedicado a explicar los fundamentos teóricos de la Transformación Operacional. Los capítulos 4 a 7 conforman la aportación de esta tesis. Particularmente, el Capítulo 4 presenta la especificación de la biblioteca, i.e., los requerimientos que se tomaron en cuenta para su desarrollo y plantea la arquitectura seguida para cumplir dichos requerimientos. Los capítulos 5 y 6 detallan el diseño que se utilizó para construir la biblioteca, incluyendo las partes internas. En el Capítulo 7 se describe una

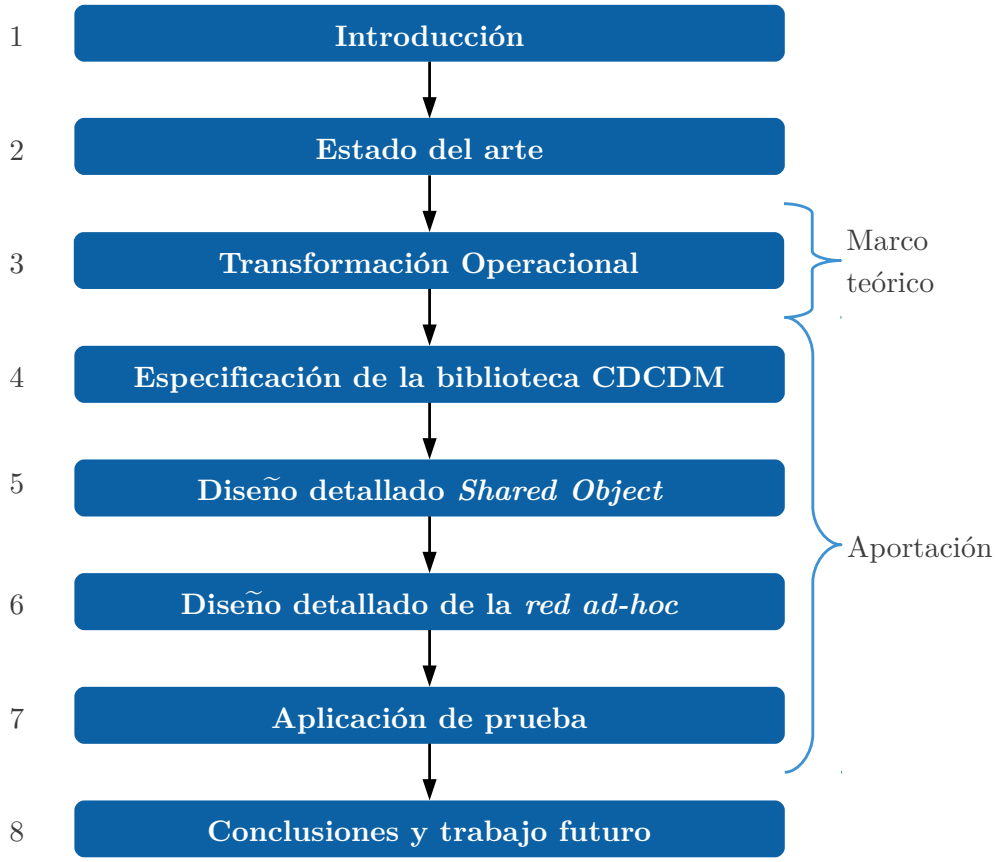


Figura 1.2: Estructura del presente documento

aplicación de prueba y cómo la biblioteca ayudó a su desarrollo. Finalmente, en el Capítulo 8 se presentan las conclusiones y el trabajo futuro.

Capítulo 2

Estado del arte

En este capítulo se describe el estado del arte respecto a las técnicas para mantener la coherencia de datos compartidos en sistemas colaborativos y respecto a los sistemas móviles.

En la Sección 2.1, se describe las técnicas para mantener la coherencia de datos compartidos generalmente utilizadas en sistemas colaborativos. En la Sección 2.2 se describe las ventajas de la técnica Transformación Operacional (OT) y los distintos algoritmos de OT existentes. En la Sección 2.3 se menciona varias aplicaciones que usan OT mientras que en la Sección 2.4 se describe las características de los sistemas móviles. Finalmente, en la Sección 2.5, se comenta esfuerzos previos relevantes para desarrollar sistemas colaborativos en dispositivos móviles y que mejoras con respecto a estos esfuerzos busca lograr la presente tesis.

2.1 Técnicas para mantener la coherencia de datos

Existen varias técnicas para mantener la coherencia de los datos compartidos en sistemas colaborativos. Greenberg y Marwood describen varias de estas técnicas, comenzando con las más simples de implementar y finalizando con las más complejas [Greenberg and Marwood, 1994]. De igual forma, los autores hacen énfasis en los problemas que estas técnicas pueden llegar a causar. A continuación, se describen dichas técnicas:

1. **Ningún control de concurrencia**: se refiere a no tomar ninguna medida preventiva para evitar incongruencias. La justificación de este enfoque es que lo producido en el espacio compartido no es realmente un fin sino un medio, por lo tanto no se requiere que las diferentes copias sean estrictamente iguales. Además se menciona que, gracias a los protocolos sociales, normalmente no se tendrán problemas tales como interferencias entre colaboradores. El problema de esta solución es obvio, ya que en la mayoría de los casos si es importante la coherencia del espacio compartido.

2. **Control por candados**: esta solución es normalmente adoptada en sistemas distribuidos. Consiste simplemente en bloquear los recursos para impedir que otros colaboradores los alteren concurrentemente. Sin embargo, el bloqueo de un recurso requiere el intercambio de una serie de mensajes que puede volver lento al sistema, especialmente en redes de área amplia como es el caso de Internet. Las consecuencias de esta situación son que el trabajo colaborativo no es suficientemente fluido y que el usuario puede perder la capacidad de trabajar concurrentemente. Munson y Dewan describen un *framework* para el control de concurrencia en sistemas colaborativos [Munson and Dewan, 1996]. En dicho trabajo se toman en cuenta candados para distintos tipos de estructuras de datos y para varios niveles de concurrencia (e.g., el candado de lectura es diferente del candado de escritura).

3. **Serialización**: esta es otra técnica derivada de los sistemas distribuidos, la cual consiste en imponer un orden total a las acciones realizadas por los colaboradores. De esta manera, al garantizar que las acciones ocurran en todos los sitios exactamente en el mismo orden, se tiene un documento compartido idéntico. Los inconvenientes de utilizar esta técnica conciernen principalmente factores humanos. Cuando dos colaboradores efectúan concurrentemente acciones conflictivas, la serialización de dichas acciones puede causar que no se cumplan las intenciones de uno de los colaboradores o de ambos, e.g., cuando dos colaboradores mueven una figura a diferentes posiciones, la figura primero se moverá a la posición de la acción que se determinó como la primera y luego a la que se determinó como la segunda, causando que se pierda la intención del colaborador que hizo el movimiento primero; además, es posible que el segundo colaborador no hubiera querido hacer el movimiento de haber sabido que otro colaborador realizaba un movimiento sobre el mismo objeto. Li et al. analizan esta situación y proponen algunas mejoras a la técnica de serialización [Li et al., 2000].

4. **Reversión de ejecución**: es nombrada de esa manera por Ellis y Gibbs [Ellis and Gibbs, 1989]. Sirve para resolver conflictos y se utiliza en conjunto con candados optimistas. Específicamente consiste en guardar la información necesaria para revertir el espacio compartido al estado válido inmediato anterior, en caso de que sea negado un candado. Esta técnica logra rapidez en la ejecución pues un colaborador puede realizar una acción, ver el resultado y continuar trabajando sin esperar a que el sistema tenga asegurado el candado, sin embargo, puede causar molestias al colaborador cuando sus cambios son revertidos.

5. **Transformación operacional**: esta técnica permite que se efectúen operaciones localmente sin la necesidad de informar a priori a otros sitios. Las operaciones realizadas son enviadas a los demás sitios donde son transformadas para que el resultado sea congruente con todos los sitios. Esta técnica será la que se aborde en esta tesis, por lo que a continuación se presenta un análisis histórico de su evolución.

2.2 Evolución histórica de Transformación Operacional

Esta técnica fue propuesta originalmente por Ellis y Gibbs, como un algoritmo que no requiere de candados sino que, para mantener la coherencia, realiza correcciones en el espacio compartido cuando recibe operaciones de los sitios remotos. Este algoritmo es denominado *distributed Operational Transformation* (dOPT). Las operaciones se dividen en dos partes: generación y ejecución. La generación corresponde al momento en que el colaborador ejecuta la acción correspondiente por medio de la interfaz de usuario, mientras que la ejecución se refiere a concretar la operación sobre el espacio compartido. La generación ocurre en un solo sitio, mientras que la ejecución tiene lugar en todos los sitios. De ser necesario, dichas ejecuciones son transformadas, e.g., cuando se reciben después de tiempo. Para determinar si una operación fue recibida tardíamente se usa un mecanismo de estampas de tiempo, que en conjunto con la dirección de red, forman un orden total.

En los más de 20 años que tiene la técnica OT se han propuesto varios algoritmos que aportan mejoras, simplificaciones y correcciones. A continuación se resume la historia de estos algoritmos.

En 1989, Ellis y Gibbs propusieron el algoritmo dOPT, dando inicio a la investigación de OT.

Gordon probó en 1995 que el algoritmo dOPT era incorrecto [Gordon, 1995] y propuso una corrección; el error en el algoritmo dOPT es nombrado *dOPT Puzzle* en algunos artículos y será explicado en detalle en la Sección 3.3 del Capítulo 3.

En 1995, se desarrolló el sistema *Jupiter* [Nichols et al., 1995] en Xerox PARC; este sistema contaba con un servidor central al cual se conectaban terminales; si el servidor lo requería, transformaba las operaciones recibidas de una terminal y las reenviaba a las demás terminales, las cuales podrían necesitar volver a transformar las operaciones. Este esquema no presenta el error *dOPT Puzzle*.

En 1996, Ressel et al. propusieron el algoritmo adOPTed, esto fue independientemente de Gordon y constituyó otra corrección, pero sin hacer una prueba formal [Ressel et al., 1996]. En 2002, Bradley realizó una prueba formal del algoritmo adOPTed en su tesis de maestría [Bradley, 2002]. La vertiente del algoritmo adOPTed no continuó desarrollándose, sin embargo este algoritmo influyó en la otra vertiente que se explica a continuación.

Hendrie demostró en 1998 que el algoritmo de Gordon era incorrecto [Hendrie, 1998].

Simultáneamente, Sun et al. propusieron el algoritmo *Generic Operational Transformation* (GOT) [Sun et al., 1998]. Este algoritmo asegura la convergencia utilizando dos tipos de transformaciones: 1) la transformación de inclusión (IT) que incluye en una operación el efecto de otra operación y 2) la transformación de exclusión (ET) que excluye de una operación el efecto de otra operación. El algoritmo GOT introduce el concepto de contexto de operación, el contexto de una operación es la secuencia de operaciones que se habían ejecutado cuando la operación se generó. El algoritmo GOT se organiza en tres casos, de los cuales el tercero es el que hace fallar al algorit-

mo dOPT; para este caso se realiza una serie de ITs y ETs según sea necesario para modificar el contexto de cada operación, solucionado así el error *dOPT Puzzle*.

Ese mismo año, Ellis y Sun propusieron una optimización del algoritmo GOT por medio del algoritmo GOT *Optimized* (GOTO) [Sun and Ellis, 1998], siguiendo el concepto de contexto de operación, pero integrando algunos conceptos del algoritmo adOPTed. La optimización logra reducir el número de transformaciones, aprovechando las restricciones impuestas por el algoritmo adOPTed sobre las funciones de transformación y haciendo algunos cambios al algoritmo original GOT.

Una propuesta más se realizó en 1998, la cual consiste en una serie de algoritmos aparentemente independiente de GOT y adOPTed [Suleiman et al., 1998]. Aunque está propuesta no siguió ninguna de las dos vertientes anteriores, cabe resaltar que ya considera la existencia de sitios móviles e incorpora procedimientos para soportar la desconexión y reconexión de dichos sitios; además soporta el trabajo en los sitios móviles aun cuando estos se encuentran desconectados.

En 2006, fue propuesto el algoritmo *Context-based* OT (COT) [Sun and Sun, 2009], que representa la evolución final del contexto de operación. El algoritmo COT es más simple que el GOTO, pues trata todos los casos de forma general en vez de usar estructuras *case* como lo hacen los algoritmos GOT y GOTO. Además, elimina completamente la necesitad de utilizar ET y generaliza el concepto de contexto pues considera un conjunto en vez de una secuencia (no se requiere un orden). Junto con el algoritmo COT, se publicó también la prueba formal de éste. Las simplificaciones anteriores hacen que el algoritmo COT sea actualmente el algoritmo de OT más avanzado. Este algoritmo se seleccionó para la presente tesis por sus simplificaciones y optimizaciones, a pesar de que no considera originalmente sitios móviles. Una explicación detallada de este algoritmo se presentará en la Sección 3.6 del Capítulo 3.

La figura 2.1 muestra como influyeron entre sí los diferentes algoritmos revisados y los años en los que fueron publicados. Como se puede ver, cuatro algoritmos se desarrollaron a partir del algoritmo original dOPT, pero solo una de las vertientes continuó desarrollándose hasta la década pasada, mientras que el resto detuvo su desarrollo en la década de los 90s. Aunque el algoritmo COT se deriva del algoritmo GOT, el algoritmo adOPTed influyó en la creación del algoritmo GOTO y este a su vez en el algoritmo COT. Una observación interesante es que el trabajo de Suleiman et al. no consideró los progresos que ocurrieron entre 1995 y 1998.

2.3 Aplicaciones de Transformación Operacional

Actualmente existen muchas aplicaciones de escritorio y aplicaciones Web que utilizan la técnica OT; ejemplos destacado son los siguientes:

- CoVim es un editor de texto basado en el editor para terminal vim (vi improved)[1].

[1] http://cooffice.ntu.edu.sg/covim/

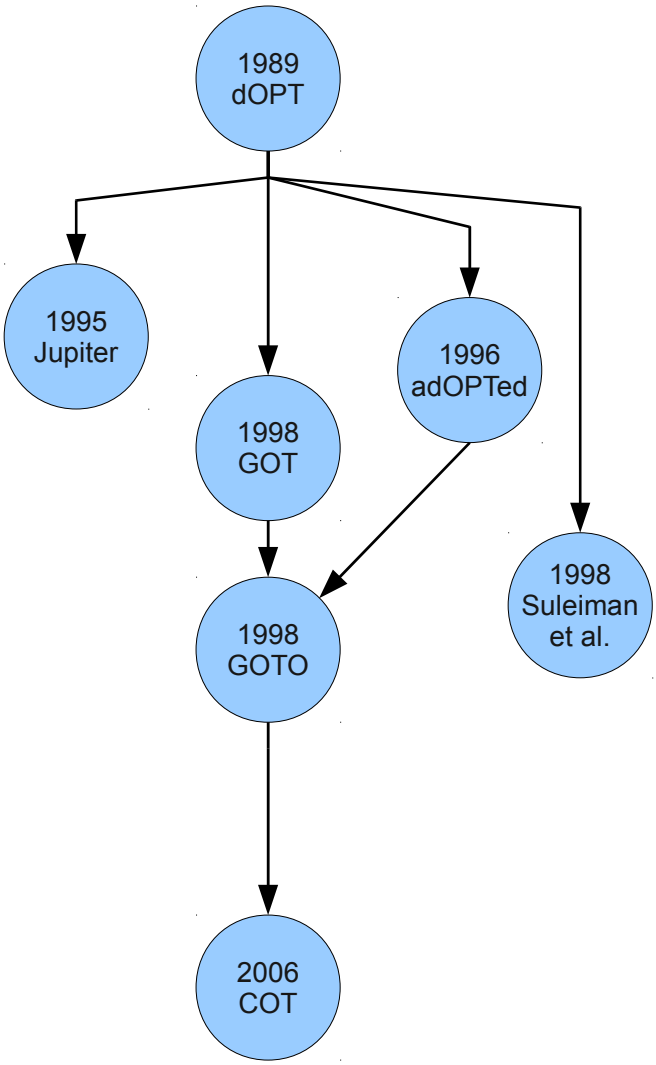


Figura 2.1: Evolución histórica de los algoritmos OT

- CoCKEditor es una aplicación Web de edición de texto[2].
- CodoxWord es la versión con soporte colaborativo del procesador de palabras MS Word[3].
- CoFlash es un editor de gráficos vectoriales con el formato Adobe Flash[4].
- CoPowerPoint es la versión con soporte colaborativo del editor de presentaciones MS PowerPoint[5].

[2] http://cooffice.ntu.edu.sg/cockeditor/
[3] http://www.codoxware.com/codoxword
[4] http://cooffice.ntu.edu.sg/coflash/
[5] http://cooffice.ntu.edu.sg/copowerpoint/

- CoCalc es un editor de hojas de cálculo basado en OpenOffice.org Calc[6].
- CoMaya es una herramienta de diseño 3D basada en Autodedk Maya[7].
- Google Docs es un servicio de aplicaciones Web que incluyen funcionalidades de procesador de palabras, hoja de cálculo y presentaciones.[8]

Con lo antes mencionado, se puede ver la amplia gama de aplicaciones que puede ser soportada por OT, sin embargo la mayoría de estas aplicaciones son para equipos fijos. Algunas aplicaciones están basadas en la Web, pero la cantidad de dispositivos móviles que las pueden ejecutar correctamente es reducido (en comparación con la cantidad de dispositivos que pueden ejecutar Java ME), aunado al inconveniente de que el usuario debe estar conectado a la Web.

2.4 Dimensiones de la movilidad

Los dispositivos móviles son sistemas de cómputo que pueden fácilmente ser movidos de un lugar físico a otro y cuyas capacidades de cómputo pueden ser utilizadas mientras están en movimiento. Algunos ejemplos son los asistentes personales digitales (PDA) y los teléfonos móviles. Existen muchas tareas que un dispositivo móvil puede hacer y que un sistema estacionario no; es por estas funcionalidades adicionales que se debe tratar de forma diferente a los dispositivos móviles.

Algunos aspectos que distinguen a los dispositivos móviles son: el uso de redes inalámbricas, su tamaño reducido, sus fuentes de poder limitadas, etc. Estas características causan que las aplicaciones móviles sean inherentemente distintas a las aplicaciones convencionales.

Las dimensiones de la movilidad [B'far, 2004] permiten caracterizar el problema de construir una aplicación móvil en un dispositivo móvil. A continuación se describen:

1. **Conciencia de la ubicación**. Puesto que las aplicaciones son móviles, los dispositivos no siempre se encontrarán en la misma ubicación; aunque esta característica puede ser un problema también brinda la oportunidad de mejorar la aplicación, e.g., un usuario puede trabajar en dos lugares diferentes llevando el dispositivo móvil a estos lugares, eliminando de esta forma la necesidad de tener un dispositivo en cada lugar.

2. **Calidad del servicio de la red**. El cambio de ubicación física implica desconexiones de la red casi seguras; además, el uso de tecnología inalámbrica introduce factores que afectan la calidad del enlace, e.g., el clima u objetos sólidos entre el receptor y el emisor. Como consecuencia, la aplicación móvil debe tener la capacidad de manejar errores de desconexión.

[6] http://www3.ntu.edu.sg/home/czsun/projects/cocalc/
[7] http://cooffice.ntu.edu.sg/comaya/
[8] https://docs.google.com/

3. **Capacidades limitadas del dispositivo**. Las limitaciones de tamaño físico en los dispositivos móviles afectan la capacidad de su CPU y memoria. El desarrollo de aplicaciones es afectado por dichas limitaciones, por tanto se debe cuidar el uso de estos recursos; otra forma en la que afecta el desarrollo de aplicaciones es que no siempre se puede portar las herramientas y bibliotecas que se acostumbra utilizar en dispositivos estacionarios.

4. **Fuente de poder limitada**. Debido a la movilidad se debe usar baterías en los dispositivos móviles y debido a la limitación de tamaño físico las baterías no pueden ser muy grandes. Como consecuencia, en algunos casos la aplicación se debe preocupar de ahorrar energía, apagando la luz de la pantalla, desconectándose de la red o disminuyendo la velocidad o el uso del procesador.

5. **Interfaces de usuario variadas**. En una aplicación estacionaria se puede dar por hecho el ratón, el teclado y el monitor, sin embargo en una aplicación móvil nada es seguro, ya que la resolución, los colores de la pantalla y el método de entrada y salida pueden variar.

6. **Proliferación de plataformas**. Debido a que los dispositivos móviles son menos costosos de fabricar que los dispositivos estacionarios, hay una gran cantidad de fabricantes y de modelos. Por lo tanto, las aplicaciones deben buscar ser independientes de la plataforma, puesto que un requerimiento común de las aplicaciones móviles es que sean soportadas por múltiples plataformas.

7. **Transacciones activas**. En un sistema estacionario, el usuario inicia la interacción y el sistema responde. Generalmente esta es la única forma en que el sistema interactúa con el usuario estacionario, el cual siempre está sentado frente a la computadora y enfocado en esta. Sin embargo, en un sistema móvil el usuario no está enfocado en el sistema móvil sino en otras actividades que implican movilidad, por esta razón el sistema algunas veces necesita iniciar la interacción por sí mismo. A este comportamiento distinto del sistema se le llama transacción activa.

2.5 Sistemas colaborativos en dispositivos móviles

En lo que respecta a dispositivos móviles, se tienen pocos esfuerzos para producir bibliotecas de sistemas colaborativos. Particularmente, Roth detalla el desarrollo del *framework* Pocket DreamTeam [Roth, 2005], que realmente es una extensión del *framework* (DreamTeam) para sistemas fijos [Roth, 2000]. Una característica importante de Pocket DreamTeam es que se limita a dispositivos Palm; además requiere una infraestructura, la cual consiste de un nodo *proxy* que es una computadora fija y accesible desde el dispositivo móvil. Sin embargo, resulta un problema el hecho de que se necesite un nodo *proxy*, pues no siempre será posible tener acceso a Internet o a la red local donde se encuentre el nodo *proxy*, e.g., cuando los usuarios estén sin

señal de Wi-Fi o de la red celular. Este problema será abordado en la presente tesis por medio de redes *ad-hoc*.

Otro punto importante del trabajo de Roth es que no utiliza la tecnología Java ME, debido a lo que denomina como "limitaciones técnicas". Aunque Roth no detalla estas limitaciones, la Kilobyte Virtual Machine (KVM), utilizada por Java ME, solo implementa conexiones TCP/IP de manera opcional. El único tipo de conexión del que se puede depender en la KVM son las peticiones HTTP, las cuales no son apropiadas para sistemas colaborativos síncronos.

Actualmente existe una especificación de la tecnología Bluetooth 1.1 para Java ME, JSR 82. Dicha especificación también es opcional, sin embargo gran parte de los dispositivos *smartphone* actuales tiene capacidades Bluetooth incorporadas.

Otros investigadores han propuesto técnicas para modelar escenarios de trabajo colaborativo móvil y de esta forma ayudar a comprender los requerimientos del sistema [Herskovic et al., 2009]. También se han propuesto diseños abstractos que pueden ser utilizados como guía para el diseño concreto de nuevos sistemas móviles colaborativos [Rodríguez-Covili et al., 2011] o patrones arquitecturales para aplicaciones móviles colaborativas [Neyem et al., 2009]. Sin embargo, estos esfuerzos mencionados no consideran la implementación ni el reuso de software y de código, sino que solo abarcan el nivel de modelado y diseño.

2.6 Análisis comparativo

En lo referente a técnicas para mantener coherencia de datos compartidos, la Transformación Operacional presenta muchas ventajas con respecto a las otras técnicas. En sistemas colaborativos síncronos, la Transformación Operacional es la mejor opción gracias a su velocidad de respuesta, la cual no es afectada por la latencia de la red ni por el tiempo de respuesta de otros sitios. También es mejor que la técnica de Reversión de Ejecución puesto que OT transforma los cambios de los usuarios en vez de eliminarlos.

En el área de dispositivos móviles, la biblioteca propuesta en la presente tesis tiene la ventaja, con respecto a Pocket DreamTeam, de no requerir infraestructura adicional. En cuestiones de implementación, la presente tesis tiene la ventaja de que en la actualidad, la plataforma Java ME está más difundida que la plataforma Palm. Hay que tener en cuenta que la plataforma Java ME está basada en una máquina virtual por lo que no está enfocada a ningún tipo de hardware. La plataforma Palm contrasta porque requiere de software y además de un hardware específico.

Con respecto a las propuestas de Herskovic y Rodríguez, el trabajo de la presente tesis está orientado a resolver problemas de implementación mas no al modelado y al diseño abstracto. El trabajo de Neyem tampoco presenta detalles de implementación y se enfoca en el espacio de comunicación y coordinación, haciendo poco énfasis en el espacio de producción. Otra diferencia de la presente tesis con respecto a los trabajos de Herskovic, Rodríguez y Neyem, es que en esta se explora principalmente el trabajo colaborativo cara a cara.

Capítulo 3

Transformación Operacional

La técnica Transformación Operacional (OT de inglés *Operational Transformation*) tiene como finalidad mantener la consistencia de datos compartidos en el contexto de editores colaborativos. Esta técnica tiene la ventaja de ofrecer una gran velocidad de respuesta a las acciones de los usuarios aun en condiciones de red poco favorables.

Este capítulo sirve de introducción a la técnica OT pues describe los antecedentes y conceptos básicos. En la Sección 3.1 se desarrolla la idea básica de OT, la cual es ilustrada mediante un ejemplo. En la Sección 3.2 se explica en detalle el algoritmo *distributed Operational Transformation* (dOPT) [Ellis and Gibbs, 1989], que representa el primer esfuerzo reportado de la técnica OT, mientras que en la Sección 3.3 se resaltan algunas limitaciones de dicho algoritmo. En la Sección 3.4 se define los conceptos básicos del concepto Contexto de operación, en tanto que en la Sección 3.5 se explica las condiciones de contexto. Finalmente, en la Sección 3.6 se describe y ejemplifica el algoritmo *Context-based Operational Transformation* (COT).

3.1 Conceptos básicos

Desde un punto de vista abstracto y simplificado, un sistema de edición colaborativa en tiempo real se puede ver como: un objeto compartido que representa al documento en proceso de edición y una serie de operaciones realizadas sobre este objeto.

Para un determinado objeto existen operaciones específicas mediante las cuales solo es posible editarlo. Tomando un editor de texto como ejemplo, el objeto compartido sería un *buffer* de texto simple, el cual inicialmente es una cadena vacía que puede ser modificada por dos operaciones: "insertar un carácter" y "borrar un carácter".

Las operaciones reciben parámetros; en el caso del editor de texto, la operación "insertar un carácter" requiere los parámetros `posición` y `carácter`, para saber en qué posición será colocado el carácter y qué carácter será insertado. La operación "borrar un carácter" solo recibe la `posición` del carácter que será eliminado.

Este ejemplo no presenta gran complicación cuando se tiene un escenario con un

solo usuario que realiza las operaciones localmente. Sin embargo al tener más de un usuario y cada usuario en un sitio remoto, la situación se complica por las siguientes razones:

- **Inconsistencias por retardos en la red**. Cuando un usuario efectúa una operación, ésta es ejecutada localmente en un tiempo imperceptible para el usuario pero, en sitios remotos, dicha operación puede ser ejecutada con un retardo considerable. Debido las limitaciones actuales de Internet, este retraso puede tardar segundos. Durante este tiempo de retardo, es posible que el documento se vuelva inconsistente entre dos sitios y que se realicen cambios sobre información atrasada.
- **Operaciones ejecutadas en diferente orden**. Este problema ocurre aun cuando se dé por hecho que los enlaces tienen la propiedad FIFO, debido a que los sitios pueden estar conectados por distintos enlaces y que incluso un mismo enlace puede comportarse de manera diferente en distintos momentos del tiempo. Debido a que cada sitio recibirá las operaciones en distinto orden, los resultados podrían ser diferentes si las operaciones se ejecutan en el orden que se reciben.

A continuación, se presenta un caso que sirve para ejemplificar los problemas de consistencia que surgen cuando no es aplicada la técnica OT. Inicialmente, el objeto compartido está en el siguiente estado: la cadena "bar" (figura 3.1); en este caso se tienen dos usuarios, A y B, cada uno de los cuales trabaja en un sitio diferente. Para este caso se supone que los eventos ocurren en el orden que son descritos, aun cuando este orden es solo uno de muchos posibles. Los eventos se muestran en la figura 3.2 y se describen a continuación:

Índice	0	1	2
Carácter	b	a	r

Figura 3.1: Estado inicial del objeto compartido

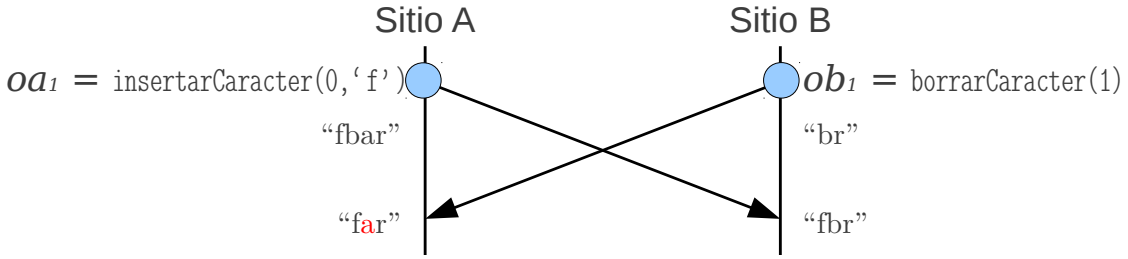


Figura 3.2: Ejemplo de operaciones sin transformación

1. El usuario del sitio A inserta la letra 'f' al inicio de la cadena mediante la operación oa_1 = `insertarCaracter(0,'f')`; para fines prácticos, este evento

repercute inmediatamente en el estado local del objeto compartido, causando que el nuevo estado sea la cadena "fbar"; en seguida la operación es transmitida al sitio B, pero no es recibida inmediatamente.

2. El usuario del sitio B borra la letra 'a' mediante la operación ob_1 = `borrarCaracter(1)`; el nuevo estado local del objeto compartido es la cadena "br" y la operación es transmitida al sitio A.

3. El sitio A recibe la operación ob_1 y la ejecuta; el resultado de ejecutar `borrarCaracter(1)` en la cadena "fbar" es la cadena "far".

4. El sitio B recibe la operación oa_1 y la ejecuta; el resultado de ejecutar `insertarCaracter(0,'f')` en la cadena "br" es la cadena "fbr".

Como se puede ver, los dos estados finales locales del objeto compartido son inconsistentes; el estado del sitio B se puede considerar correcto, pues satisface la intención de ambos usuarios es satisfecha. La intención de A era insertar la letra 'f' al inicio de la cadena (posición 0) en tanto que la de B era borrar la letra 'a' (posición 1), por tanto la cadena "fbr" satisface ambas intenciones. Sin embargo, en el sitio A, la cadena "far" solo satisface la intención de A, puesto que B no quería borrar la letra 'b' aun cuando esta está en la posición 1.

Este problema se podría solucionar si ob_1 fuera transformada para preservar la intención de B; esta transformación tiene que ser acorde con los cambios locales que B desconocía al momento de producir la operación ob_1; dichos cambios son el conjunto de operaciones $\{oa_1\}$ aun no recibidas en el sitio B al momento de generar ob_1.

La transformación que deben recibir las operaciones que desconocen a oa_1 sería la siguiente: como oa_1 recorrió un espacio a la derecha a todos los caracteres posteriores a la posición 0, todos los índices posteriores a 0 son incorrectos. Para volverlos correctos se les debe sumar 1, que es la cantidad de espacios recorridos. Aplicando esta transformación a ob_1 se obtiene la siguiente operación transformada ob'_1 = `borrarCaracter(2)`. Al ejecutar ob'_1 sobre la cadena "fbar", en vez de ob_1 (paso 3) en el sitio de A, el resultado es la cadena "fbr". Esta cadena es consistente con la del sitio de B y mantiene tanto la intención de A como la de B.

En la siguiente sección se explica cuándo y cómo se deben hacer las transformaciones por medio del algoritmo dOPT.

3.2 Algoritmo dOPT

En esta sección se explica el algoritmo dOPT [Ellis and Gibbs, 1989], el cual es importante por ser la primer propuesta de transformación operacional. Para poder entender el algoritmo se definen algunos conceptos que también serán de utilidad en capítulos posteriores.

3.2.1 Definiciones

Sistema Colaborativo - Está definido por un conjunto S de sitios y un conjunto $\mathbb{O}$ de operadores; cada operador $Op \in \mathbb{O}$ tiene un vector $par[n]$ de parámetros de entrada.

$$CS = \langle S, \mathbb{O} \rangle$$

Operación - Una operación $op \in O$, donde O es el conjunto de todas las posibles operaciones, está definida por un operador $Op \in \mathbb{O}$ y los valores de sus parámetros:

$$op = \langle Op, val[n] \rangle$$

El vector val contiene un valor por cada uno de los parámetros de Op, i.e., $|op.val| = |Op.par|$.

La generación y ejecución de las operaciones se consideran eventos separados. Una operación puede ser tanto local como remota.

Transformación de inclusión - Transforma la operación op_a contra la operación op_b y da como resultado op'_a de una forma tal que el impacto de op_b está efectivamente incluido en op'_a. Se denota como:

$$op'_a = IT(op_a, op_b)$$

3.2.2 Descripción de funcionamiento

Suponga que el algoritmo es ejecutado en el sitio $i \in S$ y que hay un solo proceso en cada sitio de S. Todas las variables con subíndice i se consideran únicamente locales, pues el algoritmo se está ejecutando en el proceso i. Los subíndices j se aplican a variables tanto locales como remotas, i.e., i puede ser igual a j pero también puede ser diferente.

Q_i es una cola local en el proceso i que contendrá tripletas de la forma: ⟨identificador del sitio, reloj vector, operación⟩.

L_i es el *log* o bitácora local de las operaciones ya ejecutadas en el sitio i.

s_i es un reloj vector [Garg, 2002] que registra las operaciones ejecutadas en el sitio i; los relojes vector denotados por s_j pueden ser locales o remotos y no necesariamente tienen información actual porque se reciben de la red con atraso o posiblemente también son desencolados con retraso.

Algoritmo 3.1 Inicialización

1: $Q_i \leftarrow \emptyset$
2: $L_i \leftarrow \emptyset$
3: $s_i \leftarrow < 0, 0, ..., 0 >$

Algoritmo 3.2 Generar operación

1: operación op_i es recibida desde la interfaz de usuario local
2: Q_i.enque($< i, s_i, op_i >$)
3: transmitir $< i, s_i, op_i >$ a todos los demás sitios

Algoritmo 3.3 Recibir operación

1: $< j, s_j, op_j >$ es recibido desde la red
2: Q_i.enque($< j, s_j, op_j >$)

Algoritmo 3.4 Ejecutar operaciones

1: \\buscar en Q_i operaciones causalmente menores o iguales que s_i
2: **for all** $< j, s_j, op_j > \in Q_i$ donde $s_j \leq s_i$ **do**
3: Q_i.deque($< j, s_j, op_j >$)
4: \\si es causalmente menor hay que transformar op_j
5: **if** $s_j < s_i$ **then**
6: $< k, s_k, op_k >\leftarrow$ entrada más reciente en L_i tal que $s_k \leq s_j$
7: \\repetir para todas las operaciones desconocidas por o_j
8: **while** $< k, s_k, op_k > \neq \emptyset$ and $op_j \neq \emptyset$ **do**
9: **if** $s_j[k] \leq s_k[k]$ **then**
10: $op_j \leftarrow IT(op_j, op_k)$
11: **end if**
12: $< k, s_k, op_k >\leftarrow$ entrada siguiente en L_i
13: **end while**
14: **end if**
15: \\una vez transformada op_j ejecutarla
16: ejecutar la operación op_j
17: \\una vez ejecutada op_j agregarla a L_i
18: L_i.append($< j, s_i, op_j >$)
19: $s_i[j]$++
20: **end for**

Por simplicidad se omite el pseudocódigo de la "prioridad" considerado en el algoritmo original. Dicha prioridad solamente es utilizada para decidir un orden total de las operaciones concurrentes, el cual se considera igual a la dirección de red.

En el algoritmo 3.4, el ciclo **for all** (línea 2) tiene como objetivo evitar la ejecución de operaciones más recientes que la última operación ejecutada localmente. El primer condicional **if** (línea 5) identifica si una operación puede ser ejecutada tal como fue generada o si tiene que ser transformada. En caso de que una operación requiera transformación, el ciclo **while** (línea 8) se encarga de realizar todas las transformaciones necesarias de acuerdo al *log* local L_i. En este ciclo se efectúan sobre la operación las transformaciones de inclusión necesarias. Finalmente, la operación es ejecutada y agregada al *log* (líneas 16-18); también se actualiza el reloj vector local s_i (línea 19).

La correctitud del algoritmo 3.4 no será discutida puesto que se conoce un contraejemplo (explicado en la Sección 3.3). En la Sección 3.6 se explica un algoritmo que ya ha sido demostrado como correcto.

3.3 dOPT puzzle

El *dOPT puzzle* es un error detectado en el algoritmo original de Ellis y Gibbs [Sun and Sun, 2009]. Aunque este error puede ser resuelto de muchas maneras, Sun y Sun han propuesto una solución basada en el concepto de contexto de operación. Esta solución tiene ciertas ventajas con respecto a las soluciones anteriores para el error *dOPT puzzle*; dichas ventajas fueron discutidas en la Sección 2.2. Esta sección se enfoca únicamente en señalar el problema existente en el algoritmo dOPT.

A continuación, se muestra un ejemplo (obtenido de [Sun and Sun, 2009]) en el cual el algoritmo dOPT falla en mantener la coherencia de datos. Este ejemplo es similar al mostrado en la figura 3.2, con la diferencia de que en este caso las operaciones son definidas sobre cadenas de caracteres. La primera operación es `insertarCadena(idx, str)` con los parámetros índice (`idx`) y cadena (`str`) y la segunda operación es `borrarCadena(idx, num)` con los parámetros índice (`idx`) y número de caracteres a borrar (`num`). La primera operación inserta la cadena `str` en la posición `idx`, en tanto que la segunda borra `num` caracteres consecutivos a partir de la posición `idx`.

Índice	0	1	2
carácter	a	b	c

Figura 3.3: Estado inicial del objeto compartido 2

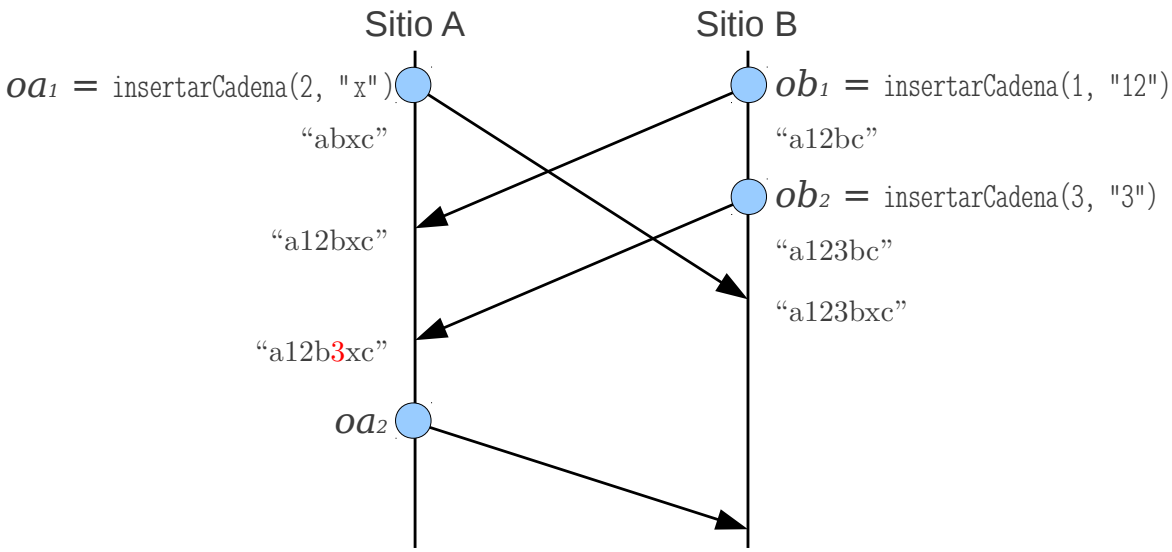


Figura 3.4: Contra-ejemplo del algoritmo dOPT

En la figura 3.4 se muestra el diagrama de la ejecución distribuida en dos sitios. Las operaciones se describen a continuación:

1. El estado inicial del objeto compartido es “abc”.

2. En el sitio B se producen las operaciones ob_1 = `insertarCadena(1, "12")` y ob_2 = `insertarCadena(3, "3")`; el resultado local de la ejecución de dichas operaciones es “a123bc”.

3. En el sitio A se genera la operación oa_1 = `insertarCadena(2, "x")`, cuyo resultado local es “abxc”; a continuación, la operación oa_1 es enviada al sitio B.

4. En el sitio B se recibe la operación remota oa_1. De acuerdo al algoritmo dOPT, esta operación es transformada contra ob_1 y ob_2; por lo tanto la operación resultante de haber aplicado estas dos transformaciones es oa_1'' = `insertarCadena(5, "x")`, la cual una vez ejecutada en el sitio B sobre el objeto compartido produce el estado “a123bxc”. Hasta este punto el algoritmo se comporta correctamente.

5. En el sitio A se recibe la operación remota ob_1, la cual es transformada contra oa_1 y la operación resultante es ob_1' = `insertarCadena(1, "12")`; como se puede apreciar, la operación resultante no sufre cambios con respecto a la original, puesto que oa_1 no le afectó dado que el índice de oa_1 es 2. La operación oa_1'' es ejecutada en el sitio A cambiando el estado del objeto compartido a “a12bxc”.

6. El sitio A recibe la operación remota ob_2, la cual es desencolada debido a que cumple la condición de causalidad $s_j \leq s_i$ de la línea 2 del algoritmo 3.4. Posteriormente, se entra al condicional **if** de la línea 5 (cuando se generó ob_2 no se había ejecutado la operación oa_1). Dentro del condicional **if** la operación ob_2

es transformada contra oa_1. Al hacer la transformación resulta que la operación oa_1 afecta a la operación ob_2, puesto que el índice de oa_1 (2) es menor que el índice de ob_2 (3); entonces los caracteres tendrían que ser recorridos un espacio; la operación transformada resultante es ob'_2 = `insertarCadena(4, "3")`, la cual produce el estado inconsistente "a12b3xc" del objeto compartido al ser ejecutada.

¿Porqué se produce este error? De acuerdo a Sun y Sun, la operación ob_2 está definida en el contexto "a12bc" mientras que la operación oa_1 en el contexto "abc". Por lo tanto sus parámetros no son comparables.

Aunque este problema puede ser resuelto de otras formas [Sun and Ellis, 1998], estas realmente son parches en vez de cambios en los fundamentos teóricos del algoritmo dOPT, como los propuestos por [Sun and Sun, 2009].

3.4 Contexto de operación

Las definiciones que se darán en esta sección se basan en las de la Sección 3.2.1 por lo que hay que tenerlas presentes. Dichas definiciones son independientes del algoritmo dOPT, así que son válidas a pesar del error *dOPT puzzle*.

Operación original - Una operación generada por el usuario y que no ha sido transformada; pudo haber sido generada localmente o recibida desde la red.

Operación transformada - Una operación que es el resultado de una o más transformaciones.

Toda operación transformada debe provenir de una operación original; la operación original de op se denota por $org(op)$; cabe mencionar que la original de op puede ser op misma, cuando op no ha sido transformada. La original de una operación siempre será la misma sin importar cuantas veces haya sido transformada, como se muestra en el siguiente ejemplo:

op es una operación original
$op' \leftarrow IT(op, op_1)$
$op'' \leftarrow IT(op', op_2)$
$op''' \leftarrow IT(op'', op_3)$
$op = org(op)$
$op = org(op')$
$op = org(op'')$
$op = org(op''')$

En el ejemplo anterior se definen tres operaciones transformadas op', op'' y op'''. Primero la operación original op es transformada contra la operación op_1 por transformación de inclusión (definida en la Sección 3.2.1) y se obtiene op'. En este caso es irrelevante si las operaciones op_1, op_2 y op_3 son originales o transformadas porque para este ejemplo solo interesan op', op'' y op''', que fueron definidas como transformadas. La operación op' es transformada contra op_2 y se obtiene op''; esta última es

transformada contra op_3 y se obtiene op'''. A partir de op se obtienen tres operaciones transformadas. Después de las tres asignaciones, se muestran cuatro igualdades, en las cuales se observa lo mencionado antes del ejemplo, i.e., una operación original es su propio original ($op = org(op)$) y el original de una operación será el mismo sin importar cuantas veces sea transformada ($op = org(op') = org(op'') = org(op''')$).
Estado del documento - Un estado del documento DS se refiere al estado del objeto compartido en determinado momento y se define como:

- Inicialmente $DS = \emptyset$, cuando no se han producido operaciones.
- Después de ejecutar la operación op sobre DS, este cambia su valor para incluir todo lo que incluía antes más la original de op, $DS \leftarrow DS \cup org(op)$.

Nótese que el estado del documento siempre es un conjunto de operaciones originales, el cual representa de forma única el estado de un objeto compartido. Si dos conjuntos contienen los mismos elementos entonces se tienen conjuntos equivalentes. Esta equivalencia aplica para el estado del documento que también es un conjunto.
Contexto de operación - El contexto de una operación op se denota por $C(op)$ y se define como:

- Para una operación original op, $C(op) = DS$ donde DS es el estado del documento sobre el que se generó op.
- Para una operación transformada op', $C(op') = C(op) \cup org(op_x)$ donde $op' \leftarrow IT(op, op_x)$ y $op = org(op')$.

El contexto de una operación transformada incluye el contexto de la operación antes de que fuera transformada y a la original de la operación contra la que fue transformada. La transformación de inclusión (definida en la Sección 3.2.1) es una forma de modificar el contexto de una operación, agregandole operaciones originales.

La Tabla 3.1 resume la notación definida en este capítulo.

3.5 Condiciones de contexto

Las condiciones de contexto aseguran que las transformaciones sean realizadas correctamente y sirven como base para el diseño de algoritmos de la técnica OT. Uno de estos algoritmos es el algoritmo COT, el cual hace cumplir la mayoría de estas condiciones.

Las condiciones de contexto se describen a continuación:

- **Condición de contexto 1** (CC1) - Dada una operación original op y un estado del documento DS, donde $op \notin DS$, op puede ser transformada$C(op) \subseteq DS$ para ejecutarse en DS solo si $C(op) \subseteq DS$.

  La condición CC1 restringe el orden de ejecución en un sistema de OT y cubre la restricción que imponía dOPT, i.e., las operaciones deben ser ejecutadas en su orden causal [Ellis and Gibbs, 1989].

Notacion	**Nombre**	**Comentario**
op, oa_1, op_a	operación, operación a uno, operación a	puede ser original o transformada
op', oa'_1, op'_a	operación prima, operación a uno prima, operación a prima	siempre es transformada
DS, CD	conjuntos de operaciones	conjuntos cuyos elementos son siempre operaciones
$org(op)$	original de una operación	el resultado será siempre una operación original
$C(op)$	contexto de una operación	el resultado será siempre un conjunto de operaciones originales
$IT(op_1, op_2)$	transformación por inclusión	el resultado será siempre una operación transformada

Tabla 3.1: Resumen de la notación

- **Condición de contexto 2** (CC2) - Dada una operación original op y un estado del documento DS, donde $op \notin DS$ y $C(op) \subseteq DS$, el conjunto de operaciones contra las que op debe ser transformada antes de ser ejecutada en DS es $DS \setminus C(op)$.

- **Condición de contexto 3** (CC3) - Dada una operación de cualquier tipo op y un estado del documento DS, op puede ser ejecutada en DS solo si $C(op) = DS$.

  La condición CC3 se podría considerar un corolario de CC2 puesto que $DS \setminus C(op) = \emptyset$, por lo que op no requiere ser transformada.

- **Condición de contexto 4** (CC4) - Dada una operación original op_x y una operación op de cualquier tipo, donde $op_x \notin C(op)$, op_x puede ser transformada al contexto de op solo si $C(op_x) \subseteq C(op)$.

  La condición CC4 impone un orden en el que deben ser transformadas las operaciones.

- **Condición de contexto 5** (CC5) - Dada una operación original op_x y una operación op de cualquier tipo, donde $op_x \notin C(op)$ y $C(op_x) \subseteq C(op)$, entonces $C(op) \setminus C(op_x)$ es el conjunto de operaciones contra las que op_x debe ser transformada antes de ser transformada contra op.

- **Condición de contexto 6** (CC6) - Dadas dos operaciones op_a y op_b, estas pueden ser transformadas entre sí, ya sea $IT(op_a, op_b)$ o $IT(op_b, op_a)$, solo si $C(op_a) = C(op_b)$.

En resumen, las condiciones CC1 y CC4 determinan los posibles órdenes en que las operaciones pueden ser ejecutadas y transformadas. Por su parte, las condiciones

CC2 y CC5 sirven para encontrar las operaciones contra las que se tiene que transformar una operación. Finalmente, CC3 y CC6 son requeridas para asegurar que la transformación y ejecución de una operación sean correctas.

Las condiciones CC2 a CC6 deben ser aseguradas por el algoritmo de OT que se esté usando, mientras que CC1 puede ser asegurada por medio de esquemas externos al algoritmo, e.g., ordenando las operaciones por medio de un servidor albergado en el nodo central de una red con topología de estrella.

3.6 Algoritmo COT

El algoritmo COT se describe en el Algoritmo 3.5.

Los algoritmos 3.1, 3.2 y 3.3 de la Sección 3.2.2 se continúan utilizando, sin embargo el desencolado en Q_i se realiza de manera externa al Algoritmo 3.5. Este desencolado se debe realizar cumpliendo la condición CC1.

Se debe mantener un estado del documento DS, el cual reemplaza al *log* L_i del Algoritmo 3.4 de la Sección 3.2.2.

Algoritmo 3.5 COT-DO(op, DS)

Entrada: operación op, el estado del documento DS

1: transform($op, DS \setminus C(op)$)
2: execute(op)
3: $DS \leftarrow DS \cup org(op)$

Algoritmo 3.6 transform(op, CD)

Entrada: operación op, un conjunto de operaciones CD

1: **while** $CD \neq \emptyset$ **do**
2: Seleccionar y remover op_x de CD, tal que $C(op_x) \subseteq C(op)$ `//CC4`
3: transform($op_x, C(op) \setminus C(op_x)$)
4: $op \leftarrow IT(op, op_x)$
5: $C(op) \leftarrow C(op) \cup org(op_x)$
6: **end while**

3.6.1 Descripción de funcionamiento

Para que el algoritmo COT funcione correctamente, se debe asegurar las condiciones de contexto CC1, CC3, CC4 y CC6. Antes de ejecutar el algoritmo COT la condición CC1 debe ser asegurada de forma externa a la función COT-DO. Esto se puede hacer mediante la cola Q_i, desencolando únicamente las operaciones que cumplen CC1 para procesarlas con COT-DO.

Las operaciones en la cola Q_i que no cumplen con la condición CC1 permanecerán en Q_i hasta que esta condición se cumpla. La condición CC1 se cumple para una

operación *op* cuando se hayan recibido y ejecutado todas las operaciones en el contexto de *op*.

La ejecución comienza por el algoritmo 3.5, la función `transform` es llamada (línea 1) con el fin de que la operación *op* sea transformada al contexto DS, i.e., $C(op) = DS$, cumpliendo con la condición CC3. Después *op* ya es válida (línea 2) para el estado del documento actual y es ejecutada. El estado del documento debe ser modificado pues ya incluye la operación *op* (línea 3).

La función `transform` es recursiva. En cada llamada recursiva se efectúan transformaciones; la operación contra la que *op* será transformada es elegida de acuerdo a CC4 (línea 2 del Algoritmo 3.6). El caso base de la recursividad es cuando el parámetro de entrada $CD = \emptyset$, entonces no se entra al ciclo **while** y se comienza a salir de las llamadas recursivas. Al regresar de la primera llamada a `transform`, *op* fue transformada contra todas las operaciones existentes en la diferencia de contextos $CD \leftarrow DS \setminus C(op)$.

La condición CC6 ya se cumple (línea 4 del Algoritmo 3.6) y se hace la transformación de inclusión. Esta condición se cumple porque al salir de la función `transform` (línea 3), $CD = \emptyset$ y dado que CD era originalmente la diferencia de los contextos $(DS \setminus C(op))$, los contextos ahora son equivalentes.

3.6.2 Ejemplo

Con el fin de ilustrar como el algoritmo COT enfrenta el error *dOPT puzzle*, se retomará el ejemplo de la Sección 3.3, en el cual falla el algoritmo dOPT.

Se debe tomar en cuenta que una operación puede tener dos nombres: 1) el nombre de la variable local en determinado nivel de la pila de llamadas a funciones y 2) el nombre del diagrama de ejecución. De igual forma los estados del documento DS se pueden ver como una variable o como un conjunto de operaciones. Finalmente, se considera que una variable pasada como parámetro a una función, conserva las modificaciones que reciba dentro de dicha función.

Índice	0	1	2
carácter	a	b	c

Figura 3.5: Estado inicial del objeto compartido

La figura 3.6 muestra el diagrama de la ejecución distribuida. Se tienen dos sitios, A y B, y las operaciones se describen a continuación:

1. El estado inicial del objeto compartido es “abc”.

2. En el sitio B se producen las operaciones ob_1 = `insertarCadena(1, "12")` y ob_2 = `insertarCadena(3, "3")`; el resultado local de la ejecución de dichas operaciones es “a123bc”.

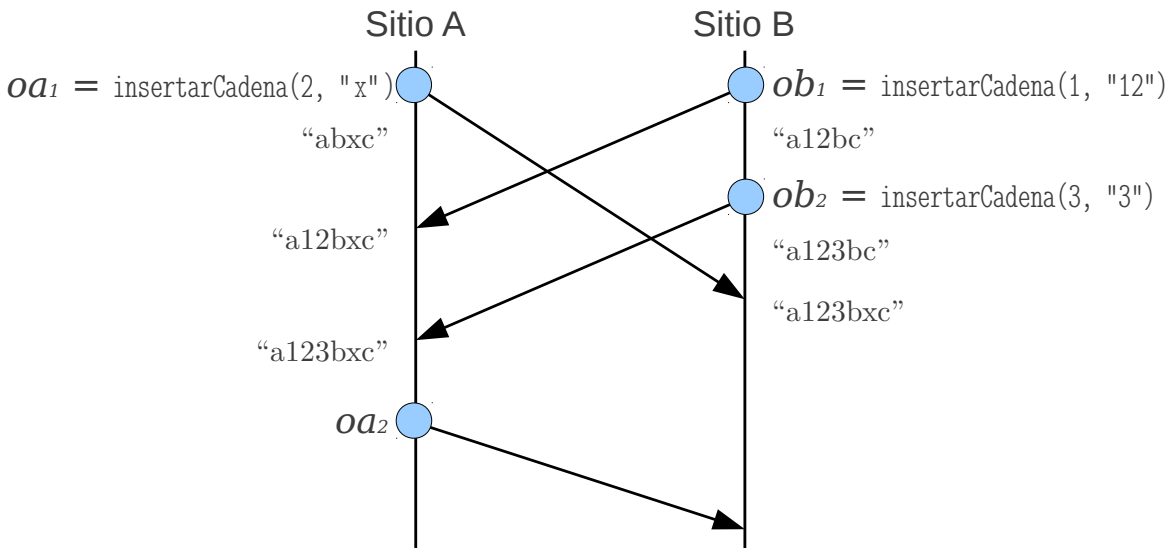


Figura 3.6: Ejecución distribuida del algoritmo COT

3. En el sitio A se produce la operación oa_1 = `insertarCadena(2, "x")`, cuyo resultado local es “abxc”; a continuación, la operación oa_1 es enviada al sitio B.

4. En el sitio B se recibe la operación remota oa_1. El contexto de oa_1 es igual a $\emptyset$, el conjunto de operaciones DS en el sitio B es $\{ob_1, ob_2\}$ pues ya se ejecutaron dos operaciones. La condición CC1 se cumple porque $C(op_1) \subseteq DS$ o bien $\emptyset \subseteq \{ob_1, ob_2\}$. Entonces la función COT-DO es llamada con los parámetros $op = oa_1$ y $DS = \{ob_1, ob_2\}$. Después se llama a la función `transform` (línea 1 del Algoritmo 3.5) con los parámetros $op = oa_1$ y $CD = DS \setminus \emptyset = \{ob_1, ob_2\}$. Dentro de la función `transform` (línea 1 del Algoritmo 3.6) se entra al ciclo **while**, luego se selecciona una operacion op_x (línea 2) tal que $C(op_x) \subseteq C(oa_1)$ (condición CC4). La única operación que cumple esta condición es ob_1 pues su contexto es $\emptyset$, ahora $CD = \{ob_2\}$ y $op_x \leftarrow ob_1$. Se vuelve a llamar a la función `transform` (línea 3) de forma recursiva con el parámetro $op = op_x = ob_1$ y $CD = C(oa_1) \setminus C(ob_1) = \emptyset$; esta llamada regresa inmediatamente porque no se entra al ciclo **while** (línea 1). Una vez que se regresa al primer nivel de la recurrencia (línea 4) se realiza la transformación ya que los contextos son iguales y se tiene que $op \leftarrow oa_1' \leftarrow IT(oa_1, ob_1)$. El resultado de transformar oa_1 contra ob_1 es oa_1' = `insertarCadena(4, "x")`. Finalmente, se actualiza el contexto de la operación op pues ya incluye a ob_1 (línea 5).

   En la segunda iteración del ciclo **while** se remueve la operación ob_2 de CD. De igual forma la función `transform` regresa inmediatamente porque $CD = C(oa_1') \setminus C(ob_2) = \{ob_1\} \setminus \{ob_1\} = \emptyset$. Después de la transformación, $op = oa_1''$ = `insertarCadena(5, "x")` y se sale del ciclo **while** porque $CD = \emptyset$. También se sale de la primera llamada de la función `transform`.

   Regresando al Algoritmo 3.5 en la línea 2 se ejecuta la operación op que ahora

tiene el valor $oa_1'' =$ `insertarCadena(5, "x")`. Finalmente, se agrega $org(op)$ al estado del documento; $org(op)$ tiene el siguiente valor $org(op) = org(oa_1'') = oa_1$.

5. El sitio A recibe la operación ob_1. De forma similar al paso 4, esta operación es transformada contra oa_1, ejecutada y agregada finalmente al estado del documento.

6. Finalmente, el sitio A recibe la operación ob_2. Al entrar a la función COT-DO (Algoritmo 3.5) se tiene que $op = ob_2$ y $DS = \{oa_1, ob_1\}$. Después se llama la función `transform` con los parámetros $op = ob_2$ y $CD = DS \setminus C(ob_2) = \{oa_1, ob_1\} \setminus \{ob_1\} = \{oa_1\}$, dentro de la función `transform` (línea 3 del Algoritmo 3.6) se selecciona a la operación oa_1 de $CD = \{oa_1\}$ ya que cumple la condición CC4 y se vuelve a llamar la función `transform` ahora con los parámetros $op = op_x = oa_1$ y $DS = C(op) \setminus C(op_x) = C(ob_2) \setminus C(oa_1) = \{ob_1\} \setminus \emptyset = \{ob_1\}$. Dentro de la segunda llamada de `transform` se transforma a oa_1 contra ob_1 y se regresa a la primera llamada de `transform`, donde la variable op_x tiene el valor $op_x = oa_1' =$ `insertarCadena(4, "x")`. El índice de oa_1' (línea 3) cambió a 4 porque refleja el efecto de ob_1, i.e., se insertaron dos caracteres a la izquierda de la cadena `"x"`. En la primera llamada de la función `transform` (línea 4 del algoritmo 3.6) será transformada ob_2 contra oa_1' en vez de oa_1, como ocurría en el algoritmo dOPT. El resultado es que oa_1' no afecta al índice de ob_2 puesto que la cadena `"x"` está a la derecha de la cadena `"3"` (el índice 3 de ob_2 es menor que el índice 4 de oa_1'). Finalmente, se modifica el contexto de ob_2 (línea 5), se regresa a la función COT-DO, se modifica el estado del documento DS (línea 3) y se termina.

En este ejemplo, el estado del objeto compartido es coherente, gracias a que oa_1 fue transformada al contexto de ob_2 (obteniendo oa_1') antes de que ob_2 fuera a su vez transformada. La condición CC6 es respetada porque $C(oa_1') = C(ob_2)$.

Capítulo 4

Especificación de la biblioteca CDCDM

En este capítulo se describen los problemas enfrentados en el desarrollo de la biblioteca propuesta para dar soporte de la Coherencia a Datos Compartidos en Dispositivos Móviles (CDCDM) así como la influencia que tuvieron estos problemas en el diseño. También, se enuncian los requerimientos de esta biblioteca, los cuales se dividen en dos tipos: 1) la funcionalidad que proporcionan sus herramientas, la cual se explica en la Sección 4.1 y 2) la capacidad de funcionar correctamente bajo algunas de las dimensiones de la movilidad, las cuales son descritas en la Sección 4.2. Finalmente, en la Sección 4.3, se describe el diseño general de la biblioteca CDCDM.

4.1 Requerimientos de funcionalidad

La funcionalidad de la biblioteca consiste principalmente de tres categorías: 1) una herramienta para mantener la coherencia de los datos compartidos, 2) una herramienta para facilitar la transmisión de mensajes entre los sitios participantes y 3) un soporte de red del cual dependen dichas herramientas. A continuación, se describe la funcionalidad proporcionada en cada una de las categorías.

4.1.1 Coherencia de datos

La coherencia de datos es necesaria, ya que en algunos tipos de aplicaciones, e.g., editores de texto colaborativos, se producen datos concurrentemente como resultado del trabajo en equipo. Para ayudar a que estas aplicaciones sean útiles a los colaboradores, es necesario que los datos compartidos sean correctos y que no se vean afectados por inconsistencias que pueden surgir entre las copias distribuidas en los dispositivos participantes.

Además de la coherencia de datos, se desea permitir un alto grado de concurrencia, debido a que algunas aplicaciones, e.g., editores de diagramas colaborativos, pueden requerir que los usuarios trabajen en datos muy próximos. En otros casos, el alto grado

de concurrencia podría no ser un requerimiento básico, pero beneficiaría al trabajo colaborativo al permitir que un mayor número de usuarios trabajen simultáneamente.

La Transformación Operacional se utiliza para mantener la consistencia de datos facilitando el acceso concurrente. En el Capítulo 3, se explicó esta técnica y en el Capítulo 5 se detallará su implementación.

4.1.2 Transmisión de mensajes

La transmisión de mensajes es necesaria para poder satisfacer otros requerimientos de las aplicaciones colaborativas, e.g., la conciencia de grupo. Si se requiriera que la aplicación proporcione comunicación directa la transmisión de mensajes también se necesita.

Dourish y Bellotti definen el concepto de conciencia de grupo como un entendimiento de las actividades de otros, el cual provee un contexto para las actividades propias. Este contexto se usa para asegurar que las contribuciones individuales son relevantes para la actividad grupal global y para evaluar las acciones individuales con respecto a las metas y al progreso del grupo. Toda esta información permite a los grupos administrar el proceso de trabajo colaborativo [Dourish and Bellotti, 1992].

La conciencia de grupo es una parte fundamental del trabajo colaborativo, pues sin esta información los usuarios estarían trabajando de manera aislada mas no como parte de un grupo. Un usuario que desconoce la actividad de los otros usuarios podría producir trabajo que ya se está produciendo, realizar acciones que obstaculicen el trabajo de sus colegas o comprender erróneamente cuál es su trabajo. Por estas razones, cuando los miembros de un grupo trabajan de manera distribuida, necesitan saber si los otros usuarios con los que colaboran están presentes o no y dónde están trabajando, qué acciones están desempeñando, sus intenciones, sus habilidades, entre otros.

Una aplicación colaborativa tiene la responsabilidad de proporcionar esta información a los usuarios, por lo que es un requerimiento de este tipo de soporte informático. En consecuencia, la biblioteca CDCDM debe proporcionar un soporte en este sentido a las aplicaciones que la utilicen. La información de conciencia de grupo es proporcionada a través de la interfaz de usuario y la forma en la que se expresa varía de acuerdo a la aplicación, e.g., por medio de *widgets*. La biblioteca CDCDM no puede implementar toda la amplia variedad de herramientas para proveer conciencia de grupo. Además, la interfaz de usuario está fuera del alcance de la presente tesis. Por estas razones el soporte para la conciencia de grupo cubre solo la capa de red. Aunque este es solo un apoyo parcial, es útil para todas las aplicaciones pues independientemente de cómo se implemente la conciencia de grupo, en todos los casos será necesario transmitir información por la red.

4.1.3 Red *ad-hoc*

La funcionalidad de red *ad-hoc* es requerida para que sea posible facilitar la interacción síncrona y espontánea entre los usuarios.

Los principales tipos de redes presentes en los dispositivos móviles son: 1) red celular de datos, 2) red Wi-Fi y 3) red Bluetooth.

Para lograr la interacción síncrona, la red utilizada debe permitir el envío y la recepción de datos en cualquier momento y desde cualquiera de los dispositivos conectados. La red celular presenta algunos obstáculos para satisfacer esta necesidad, puesto que, en la mayoría de los casos, el único tipo de conexión proveida es por medio del protocolo *Wireless Application Protocol* (WAP) que es de tipo *request-response*. Además, esta conexión no puede ser establecida directamente con otro dispositivo móvil, sino solo con un servidor Web. Por lo tanto, cuando un dispositivo intente mandar información a otro dispositivo por medio de la red celular, primero tendrá que hacer una operación *request* WAP a algún servidor central, el cual necesitará que el segundo dispositivo esté conectado para enviarla la información en la operación *response* WAP correspondiente. Este esquema presenta varios inconvenientes: 1) la conexión de la red celular tiene un costo monetario, 2) la señal de la red celular no está siempre disponible y 3) la red fallaría si el servidor no está disponible. Adicionalmente, se tendrían que implementar métodos de descubrimiento sin los cuales la interacción espontánea no sería posible.

Las alternativas de la red celular son las redes Wi-Fi y Bluetooth, las cuales permiten la conexión local entre dispositivos sin depender de la red celular y no tienen un costo monetario. También permiten la conexión punto a punto, eliminando la necesidad de un servidor central. Ambas redes son soportadas por la biblioteca CDCDM, sin embargo se da preferencia a la red Bluetooth debido a sus mecanismos de descubrimiento. Por medio de la API JSR-82, es posible hacer uso del protocolo *Service Discovery Protocol* (SDP)[1], el cual permite buscar: 1) dispositivos dentro del área de alcance y 2) servicios en los dispositivos encontrados. De esta forma, se puede iniciar una colaboración espontánea, después de haber encontrado un servicio.

La tecnología Bluetooth ofrece otra ventaja sobre Wi-Fi porque Bluetooth fue diseñada específicamente para dispositivos móviles y para consumir menos energía, lo cual aumenta la motivación de usar esta tecnología.

4.2 Requerimientos de movilidad

Otro requerimiento de la biblioteca propuesta es que funcione correctamente bajo las condiciones impuestas por los dispositivos móviles, e.g., desconexiones inesperadas.

A continuación se explica cómo el diseño de la biblioteca CDCDM enfrenta las dimensiones de la movilidad más significativas para las aplicaciones colaborativas móviles (todas las dimensiones de la movilidad se describieron en la Sección 1.2 del Capítulo 1).

[1] https://www.bluetooth.org/Technical/AssignedNumbers/service_discovery.htm

4.2.1 Calidad del servicio de la red

Esta dimensión de la movilidad se refiere a los problemas ocasionados por la red. Puesto que los dispositivos moviles cambian frecuentemente de ubicación física, no pueden estar conectados a una red cableada.

Las redes inalámbricas se basan en la transmisión de ondas de radio a través del aire, lo cual presenta varias limitaciones, e.g., perdida de datos debido a interferencias, desconexiones por dejar el radio de alcance, menor ancho de banda y mayor latencia que las redes cableadas, etc.

Estas limitaciones afectan a las aplicaciones móviles colaborativas debido principalmente a las desconexiones. Una aplicación que está diseñada para trabajar como parte de una red de dispositivos depende de la conexión de red para funcionar. Así, si en su diseño se ignora la posibilidad de que la red no esté disponible, la aplicación podría no funcionar o bien operar de forma incorrecta y perder datos cuando la red falle. En aplicaciones para dispositivos fijos, la red presenta relativamente pocas fallas sin embargo, en dispositivos móviles, las desconexiones no solo son debidas a las fallas de red, sino también a la naturaleza y a las limitaciones de las redes inalámbricas y de los dispositivos móviles, e.g., algunos teléfonos celulares suspenden las aplicaciones móviles cuando reciben una llamada así que, aunque la red esté disponible, la aplicación dejará de responder.

El diseño de la biblioteca CDCDM considera las desconexiones inesperadas al tener una arquitectura distribuida, ya que no hay ningún nodo central que tenga responsabilidades exclusivas y necesarias para otros nodos. En este sentido, cada nodo puede continuar trabajando de manera independiente, i.e., cada nodo desconocerá el trabajo que se realiza en los nodos desconectados, pero puede seguir produciendo trabajo local. En el evento de la reconexión, este nodo envía su trabajo y recibe el trabajo de los otros nodos. La técnica de Transformación Operacional (OT) garantiza que los datos serán coherentes en todos los nodos, aun cuando se haya recibido trabajo retrasado. Esta capacidad de OT contrasta con la técnica de candados [Greenberg and Marwood, 1994], los cuales presentan problemas en el evento de una desconexión pues, bajo algunos esquemas, un candado puede quedar bloqueado indefinidamente, se puede presentar inconsistencias entre nodos o bien se puede perder el trabajo de algún usuario (ver Sección 2.1).

Gracias a que la consistencia de datos, la administración de la red y la lógica de la aplicación se encuentran en capas separadas, es posible iniciar un procedimiento en la capa de red que intente reconectarse a los demás nodos, mientras que en capas superiores se continúa el trabajo sin interrupciones.

Otra de las ventajas de OT es que resulta tolerante a retrasos de la red ocasionados por alta latencia, la cual también es una característica de las redes inalámbricas con las que trabajan los dispositivos móviles.

4.2.2 Proliferación de plataformas

La proliferación de plataformas se refiere a la heterogeneidad sobresaliente en hardware y software de los dispositivos móviles. La heterogeneidad de hardware presenta diferentes capacidades de memoria y procesador, así como diferentes dispositivos de entrada y de salida. La heterogeneidad de software incluye la variedad de sistemas operativos existentes, las distintas versiones de un mismo sistema operativo y las múltiples APIs de desarrollo.

Esta heterogeneidad se enfrenta de varias formas. La principal es por medio de la máquina virtual de Java ME, la cual oculta el sistema operativo y las funciones nativas de la plataforma y proporciona una API con funciones comunes a todos los dispositivos. A cambio de sacrificar las funciones nativas se obtiene la ventaja de que la biblioteca CDCDM funcionará en un mayor número de dispositivos, i.e., todos los que tengan una implementación de Java ME.

Aun cuando Java ME resuelve parcialmente el problema de la heterogeneidad de software, aun se tiene la heterogeneidad de hardware. A la biblioteca propuesta le afecta directamente el hardware de red en este sentido. Algunos dispositivos cuentan con Bluetooth, Wi-Fi o ambos, por lo tanto los dispositivos que no cuenten con al menos uno de estos tipos de red no podrán usar la biblioteca CDCDM. Para que se puedan conectar entre sí los dispositivos que cuentan con diferentes tipos de red, algún dispositivo que tenga ambos tipos de red puede actuar como puente. A manera de ilustración, considere tres dispositivos móviles, uno con Bluetooth únicamente, otro solo con Wi-Fi y un tercero con ambos; los dos dispositivos con Bluetooth se pueden conectar entre sí; de estos dos el que tiene Wi-Fi puede escuchar un puerto TCP/IP; entonces el dispositivo que tiene únicamente Wi-Fi se puede conectar con este último para formar parte de la red. Así, los tres dispositivos pueden interactuar.

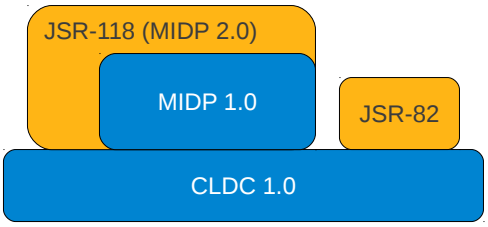


Figura 4.1: Estándares de Java ME

En la figura 4.1, se muestra la jerarquía de los estándares de Java ME. Para usar conexiones de red en Java ME se emplean varios estándares. En particular, para las conexiones TCP/IP por medio de Wi-Fi, se utiliza el estándar JSR-118 que incluye conexiones de tipo *socket*, mientras que para las conexiones por medio de Bluetooth se utiliza el estándar JSR-82 que incluye los protocolos SDP y RFCOMM[2](*Radio Frequency COMMunication*). Este último es un protocolo de transporte orientado a conexiones que garantiza, por medio de acuses de recepción y reenvíos, que los datos se entregan, de forma similar a TCP. Además de estos estándares, la bibliote-

ca CDCDM por si sola requiere únicamente de los estándares básicos de Java ME, que son CLDC 1.0 (*Connected Limited Device Configuration*) y MIDP 1.0 (*Mobile Information Device Profile*). CLDC 1.0 es la base de Java ME pues consiste de la funcionalidad mínima para programar aplicaciones, i.e., todas las implementaciones Java ME lo incluyen. MIDP 1.0 es la especificación básica para utilizar las capacidades de dispositivos móviles.

4.2.3 Fuente de poder limitada

Ya que los dispositivos móviles cambian frecuentemente de ubicación física, no pueden estar conectados todo el tiempo a la red eléctrica, por esta razón deben incluir una batería como fuente de poder. Dado que la batería limita el tiempo de funcionamiento del dispositivo, los usuarios tratan de mantener su dispositivo móvil encendido únicamente cuando lo están utilizando y suelen apagarlo cuando no lo necesitan para prolongar, de esta forma, el tiempo que la batería estará cargada.

Las aplicaciones móviles deben considerar que el usuario apaga frecuentemente su dispositivo y deben guardar su estado en almacenamiento secundario para que el usuario no pierda su trabajo al apagar el dispositivo. Cuando el usuario vuelva a prender su dispositivo, la aplicación debe cargar de nuevo su último estado para que el usuario pueda continuar su trabajo como estaba antes de apagar el dispositivo.

La tecnología Bluetooth fue diseñada para funcionar en dispositivos móviles, por lo que presenta bajo consumo de energía. Al tener en cuenta esta tecnología, también se está buscando reducir el consumo de energía, ya que el uso exclusivo de Wi-Fi representaría un consumo mayor de energía.

4.3 Diseño de alto nivel

Esta sección describe, de manera general, la estructura interna de la biblioteca CDCDM y cómo se comunica con el exterior, i.e., con la aplicación que la utiliza y con la plataforma Java ME.

4.3.1 Arquitectura basada en *toolkit*

La biblioteca CDCDM sigue un diseño de *toolkit* [Gamma et al., 1995], que provee al desarrollador con herramientas que puede incluir en su programa. Esta solución contrasta con el enfoque de *framework*, en el cual se provee una estructura fija sobre la cual el programador puede desarrollar, i.e., el *framework* determina la estructura de la aplicación. El diseño basado en *toolkit* fue preferible para evitar influir en el diseño que el desarrollador quiera usar en su aplicación. Esta libertad es importante en las

[2] https://www.bluetooth.org/Building/HowTechnologyWorks/ProfilesAndProtocols/RFCOMM.htm

aplicaciones para dispositivos móviles, pues en algunos dispositivos de bajas capacidades se querrá tener un diseño simple, mientras que en dispositivos más poderosos se podrá tener un diseño más complejo y vasto.

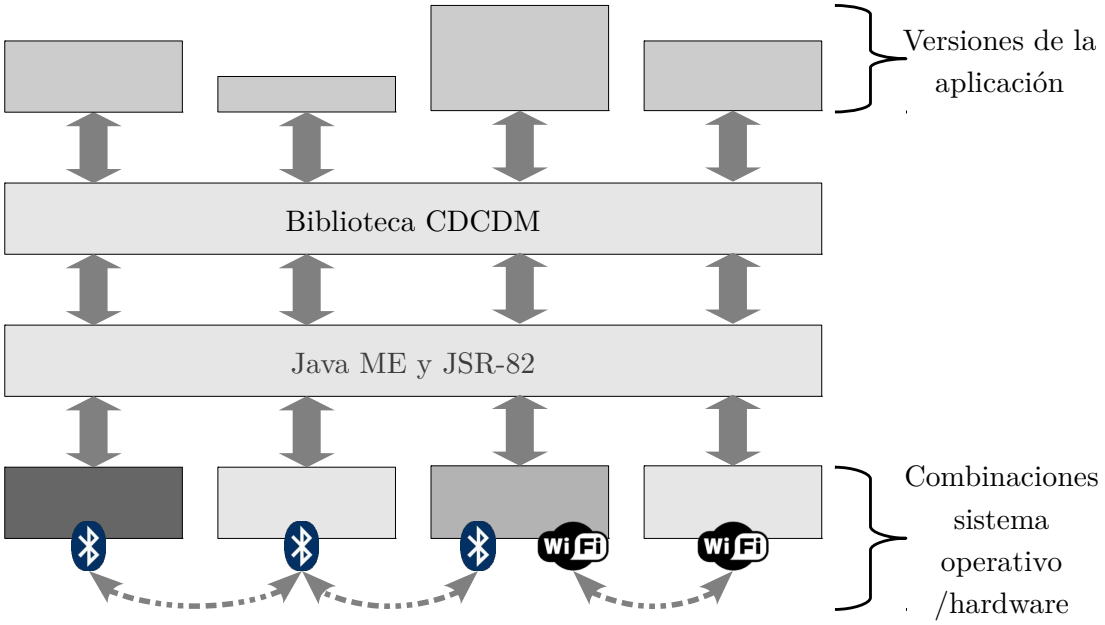


Figura 4.2: Capas de la arquitectura de la biblioteca CDCDM

En la figura 4.2 se muestran la plataforma Java ME y JSR-82, la biblioteca CDCDM y la aplicación que utiliza la biblioteca. La plataforma Java ME y JSR-82 incluye los estándares mencionados en la sección 4.2.2. Debajo de esta capa se encuentra la heterogeneidad dominante en los dispositivos móviles, la cual se representa con diferentes colores para denotar diferentes combinaciones de sistema operativo y hardware. La biblioteca CDCDM no necesita interactuar directamente con estas combinaciones, ni puede hacerlo puesto que Java ME se encuentra en medio. Arriba de la biblioteca CDCDM se muestra otro tipo de heterogeneidad, i.e., diferentes versiones de una misma aplicación. Es conveniente tener varias versiones de una aplicación si se desea aprovechar las ventajas de algunos dispositivos móviles que cuentan, por ejemplo, con mejores métodos de entrada y salida, como teclado QWERTY o pantalla táctil. Por otro lado, se podría desarrollar una versión más simple para usarla en dispositivos móviles que no tengan estas capacidades.

En esta figura se puede ver también que algunos dispositivos están conectados por Bluetooth y otros por Wi-Fi, como se discutió en la Sección 4.2.2.

4.3.2 Arquitectura distribuida

Como se mencionó en la Sección 4.2.1, ningún nodo puede depender de un nodo central ni de una infraestructura de red, por lo que cada nodo tiene todos los componentes necesarios para funcionar independientemente. Además, los nodos se pueden

conectar entre sí para formar una red *ad-hoc* que sigue una topología de árbol, i.e., cada nodo tiene un padre exceptuando al nodo raíz y puede tener uno o más hijos. El nodo padre p de un nodo L es el nodo al que se conectó L y los nodos hijos h de un nodo L son los nodos que se han conectado a L.

Esta topología busca que un nuevo nodo se pueda conectar a cualquier nodo que ya esté en la red, lo cual es conveniente porque algunos nodos podrían no ser visibles por estar fuera del área de alcance o por tener diferente tipo de red. Esta topología también es conveniente en los casos en que un cierto nodo probablemente deje la sesión con pleno conocimiento del usuario. En estos casos el usuario podrá elegir un nodo padre diferente al que probablemente deje la sesión.

La figura 4.3 muestra una secuencia que ejemplifica cómo se forma una red *ad-hoc* cuando un usuario decide crear una nueva sesión colaborativa. En cada paso del diagrama se describen los eventos desde la perspectiva del nodo oscuro, el cual siempre estará denotado por L aun cuando no será el mismo en cada paso de la secuencia. Cada nodo funciona como servidor en el sentido de que acepta nuevas conexiones y funciona como cliente en el sentido de que inicia una nueva conexión con otro nodo. Por esta razón, en la figura 4.3 se muestran flechas que van del nodo que inicia la conexión al nodo que está esperando conexiones. Las letras representan los roles que se mencionaron anteriormente, i.e., nodo padre p, nodo hijo h y nodo actual L.

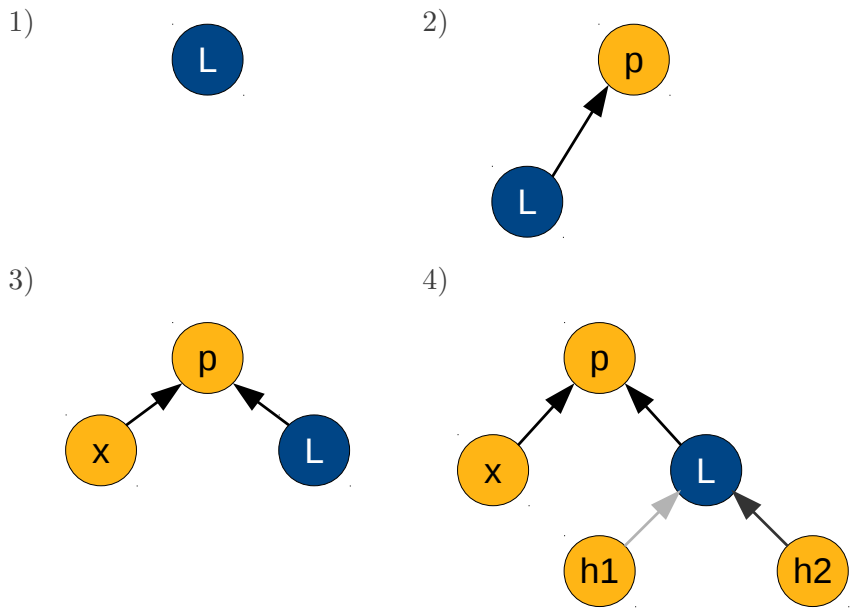


Figura 4.3: Secuencia de la creación de una red *ad-hoc*

A continuación se describen los cuatro pasos de la secuencia mostrada en la figura 4.3:

1. Inicialmente, un nodo L crea una sesión colaborativa.

2. Tiempo después llega un nuevo nodo L (el nodo que era L en el paso 1 ahora se denomina p); L se conecta al nodo p, el cual está esperando nuevas conexiones.

3. A continuación otro nuevo nodo L se conecta al nodo raíz p, el cual será padre del nuevo nodo L (en este paso el nodo nombrado x es el nodo que se llamaba L en el paso 2); el nuevo nodo L escucha tanto en la red Wi-Fi como en la red Bluetooth.

4. En este paso los nombres de los nodos no cambian. El nodo L recibió dos conexiones, de $h1$ y de $h2$, por lo tanto el nodo L tiene ahora tres vecinos, i.e., p, $h1$ y $h2$; la flecha que va de $h1$ a L denota que $h1$ se comunica por medio de Wi-Fi en vez de Bluetooth; $h1$ podría ser un nodo que no tiene Bluetooth o un nodo que está fuera del rango de la red Bluetooth. La flecha que va de $h2$ a L denota una conexión por medio de Bluetooth.

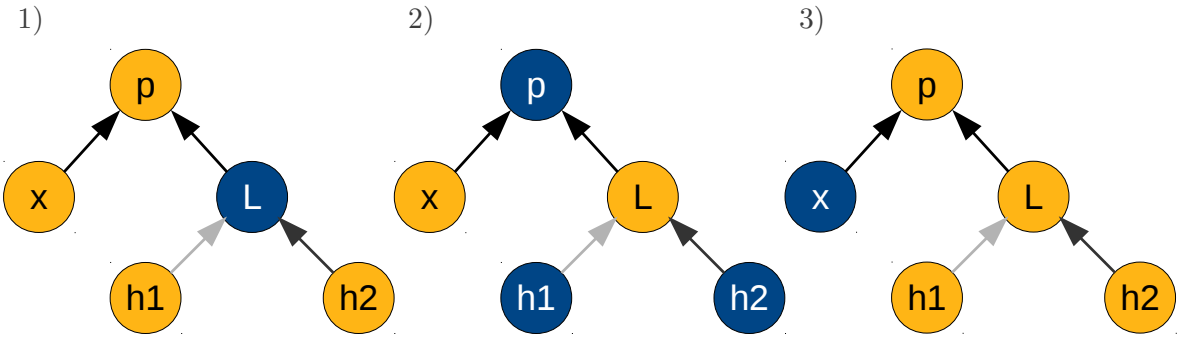


Figura 4.4: Ruteo de mensajes por medio de *blind flooding*

Para ejemplificar cómo se trasmiten los mensajes se presenta una secuencia de ejemplo en la figura 4.4, la cual se explica a continuación:

1. Suponga que se genera una operación en L, la cual se ejecutará localmente y se transmitirá a sus vecinos.

2. Cuando los vecinos $h1$ y $h2$ reciban la operación cada uno la ejecutará, pero no la transmitirán puesto que estos nodos tienen únicamente al nodo L como vecino (y la operación llegó de este nodo). En el nodo p se ejecutará la operación y además será retransmitida, ya que p tiene como vecino a x.

3. Finalmente la operación es recibida y ejecutada en x.

Este tipo de ruteo se llama *blind flooding* [Garg, 2002].

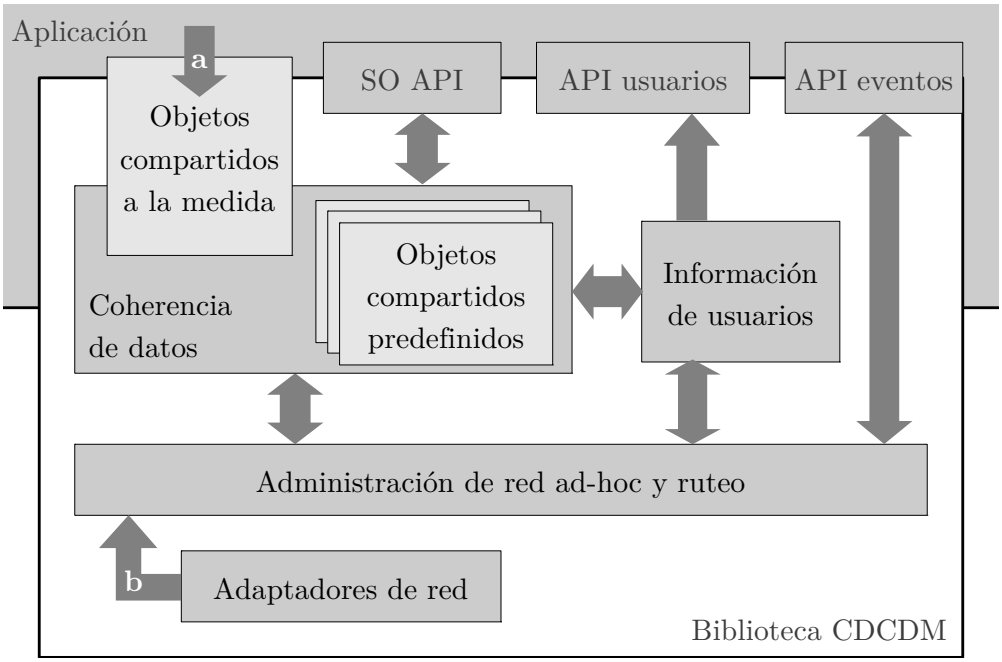


Figura 4.5: Bloques principales de la biblioteca CDCDM

4.3.3 Diagrama de bloques

En esta sección se explica el funcionamiento de cada bloque de la biblioteca CDCDM y cómo se relacionan entre sí. Los bloques principales de la biblioteca se muestran en la figura 4.5.

El bloque Coherencia de datos está encargado de mantener la coherencia de los datos compartidos. En este bloque se mantienen objetos compartidos, los cuales reciben operaciones de la interfaz de usuario y de la red a través del bloque Administración de red ad-hoc y ruteo. Los objetos compartidos pueden ser instancias de clases predefinidas, proporcionadas por la biblioteca CDCDM, o pueden ser hechos a la medida. La biblioteca propuesta permite que se agreguen objetos compartidos externos siempre que estos cumplan con una interfaz determinada (figura 4.5-a). Dentro del bloque Coherencia de datos se ejecuta el algoritmo COT por cada instancia de objeto compartido. Aunque los objetos compartidos sean de diferentes clases, el algoritmo COT funciona igual para todos, gracias a la interfaz antes mencionada. Las aplicaciones deben utilizar los objetos compartidos por medio de la API de objetos compartidos (SO API), a fin de que la biblioteca CDCDM haga internamente las tareas necesarias. En el Capítulo 5, se explica en detalle el diseño interno de este bloque y las clases que lo forman.

El bloque Información de usuarios mantiene una lista de los usuarios que se han conectado a la red *ad-hoc*. Esta información es utilizada por el bloque Coherencia de datos y también por el bloque Administración de red ad-hoc y ruteo para asignar localmente identificadores de usuario únicos. Los datos de los usuarios también son accesibles para las aplicaciones que utilizan la biblioteca CDCDM, por medio de la

API usuarios.

El bloque Administración de red ad-hoc y ruteo se encarga de enviar mensajes locales y recibir mensajes remotos. Además, en algunos casos, los mensajes remotos son reenviados a otros nodos. Los mensajes pueden ser de tres tipos: 1) eventos generados por la aplicación y pasados como parámetros a la biblioteca CDCDM por medio de la API eventos, 2) operaciones sobre objetos compartidos y 3) eventos de conexión y desconexión.

Los mensajes se originan como objetos, pero se tienen que convertir en bytes para poder ser enviados por la red. El bloque Administración de red ad-hoc y ruteo se encarga de esta conversión y de la conversión inversa, i.e., convertir bytes recibidos en objetos. Al llegar los mensajes, los objetos generados se pasan a los bloques que los necesitan, e.g., se reciben bytes que representan una operación, se crea un objeto que representa la operación correspondiente y finalmente, se pasa el objeto al bloque Coherencia de datos que realiza las acciones pertinentes con esta operación remota.

Finalmente, el bloque Administración de red ad-hoc y ruteo está encargado de registrar, en el bloque Información de usuarios, a los nuevos usuarios que se conectan. Este bloque también mantiene el control de cuáles de los nodos conocidos están actualmente conectados y cuáles están desconectados. Cuando un nodo conectado directamente al nodo local se desconecta, ya sea intencionalmente o inesperadamente, este bloque envía mensajes a los demás nodos remotos para notificarlo. De igual forma, cuando se reciben mensajes notificando que algunos nodos remotos se han desconectado, este bloque actualiza la lista de nodos conectados. En el Capítulo 6, se explicará el diseño detallado del bloque Administración de red ad-hoc y ruteo.

El bloque Adaptadores de red engloba los dos tipos de conexiones soportadas, i.e., Wi-Fi y Bluetooth. Estos dos tipos de redes son intercambiables y se pueden utilizar ambos si están disponibles o cualquiera de los dos que esté disponible. Dentro de este bloque, se encapsula el código para iniciar conexiones y esperar nuevas conexiones. Cuando se inicia exitosamente una nueva conexión, este bloque entrega un objeto que representa la conexión abierta y que funciona igual para Wi-Fi y Bluetooth. Esta conexión es pasada al bloque Administración de red ad-hoc y ruteo, el cual no necesita conocer el tipo de adaptador y trata a todas las conexiones de igual forma, i.e., puede funcionar igual con cualquiera de las dos redes o con ambas simultáneamente.

Capítulo 5

Diseño detallado de la capa *Shared Object*

En este capítulo se detalla el diseño de la capa encargada de mantener la coherencia de los datos por medio de Transformación Operacional (OT). En la Sección 5.1 se explica de forma general las principales clases de esta capa y cómo interactúan entre sí para lograr su objetivo. En la Sección 5.2 se detalla el funcionamiento de las clases relacionadas con la administración de usuarios. En la Sección 5.3 se describen los detalles de la codificación de la técnica OT y finalmente en la Sección 5.4 se presenta una implementación concreta de un objeto compartido.

5.1 Vista general de la capa de coherencia de datos

La mayor parte de esta capa se encuentra en el paquete `sharedobject`, sin embargo algunas clases de otros paquetes influyen en el funcionamiento de esta capa.

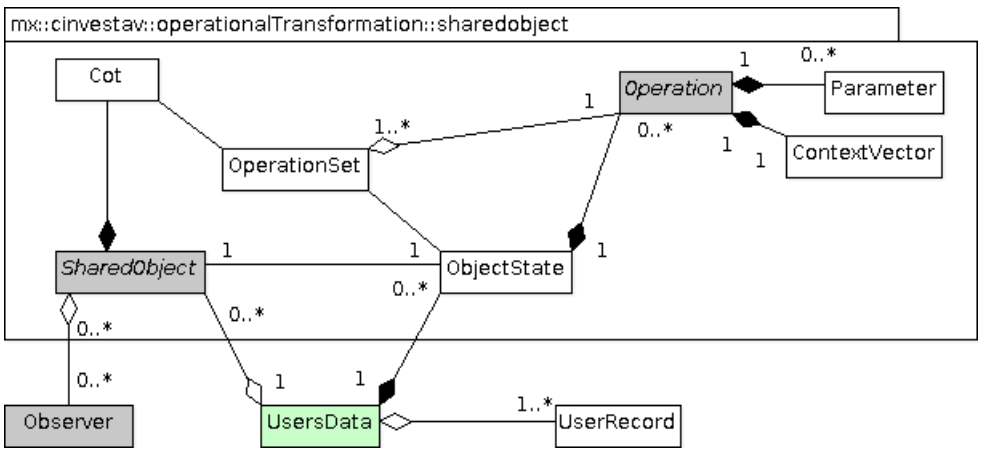


Figura 5.1: Diagrama de clases del paquete `sharedobject`

En la figura 5.1 se muestran las clases encargadas de mantener la coherencia de los

datos. Las clases externas al paquete `sharedobject` se encargan respectivamente de la administración de usuarios (`UsersData` y `UserRecord`) y de observar los cambios que ocurren en el objeto compartido (`Observer`).

La clase `Cot` es responsable de ejecutar el algoritmo COT sobre un objeto compartido dado, representado por la clase abstracta `SharedObject`. Al separar el algoritmo de Transformación Operacional del objeto compartido es posible intercambiar los objetos compartidos sin tener que reescribir el comportamiento de este algoritmo. El algoritmo COT está definido genéricamente, i.e., trabaja únicamente con operaciones, contextos, un objeto compartido y el estado de dicho objeto. Puesto que el algoritmo COT no necesita conocer ni los tipos de operaciones ni el modelo de datos del objeto compartido, éstos fueron definidos como clases abstractas que requieren de una clase concreta para una ejecución real. En la Sección 5.4 se muestra un ejemplo de clases concretas.

La clase `SharedObject` es responsable de transformar las operaciones, ya que el procedimiento de transformación varía con cada tipo de objeto compartido. La clase `Operation` representa una operación y tiene su respectivo vector de contexto, el cual es explicado en el Apéndice A. Durante la ejecución del algoritmo COT se hace uso de conjuntos de operaciones, los cuales son representados por la clase `OperationSet`.

5.2 Administración de usuarios

A la administración de usuarios le concierne las actividades de agregar y retirar usuarios. Esta actividad no es específica de la capa de coherencia de datos pero afecta su funcionamiento porque el algoritmo COT hace uso de los vectores de contexto que suponen un número fijo de usuarios. Sin embargo, en la presente propuesta, es posible aumentar el número de usuarios cualquier momento.

5.2.1 Clase ContextVector

La clase `ContextVector` representa al vector de contexto. Si el número de usuarios fuera fijo se podría usar arreglos de tamaño fíjo, pero debido a que se pueden agregar usuarios, los vectores más nuevos serán más grandes. La clase `ContextVector` provee las operaciones necesarias de los vectores de contexto, i.e., comparación de vectores, lectura de componentes e incremento de componentes. Estas operaciones pueden ser usadas por el algoritmo COT de forma transparente, en tanto que la clase `ContextVector` soluciona internamente los problemas causados por la variación en el número de usuarios, como se describe a continuación.

Comparación de vectores

Problema: Para comparar dos vectores se requiere el mismo número de componentes y no se debe ignorar los componentes que existen en un vector más grande pero no existen en otro más pequeño.

Solución: la clase `ContextVector` realiza esta comparación y considera como cero a los componentes que faltan en el vector de menor tamaño. La falta de componentes

implica que el vector fue creado antes de saber de la existencia de uno o más usuarios, por lo tanto estos usuarios no podían haber realizado ninguna operación.

Creación distribuida de vectores

Problema: Un vector puede crearse cuando el usuario local produce una nueva operación o cuando usuarios remotos producen operaciones. Al comparar dos vectores, el componente i de ambos debe referirse al mismo usuario, sin embargo no se tiene garantía de que en todos los sitios se agreguen los usuarios en el mismo orden, e.g., en el sitio del usuario A pueden agregarse los usuarios B y C de una forma tal que $A < B < C$, mientras que en el sitio del usuario B se tiene que $B < A$ y $B < C$ porque inicialmente B solo conoce de su existencia pero no de la de A y C. Además podría descubrir la existencia de C antes de la existencia de A, quedando el siguiente orden $B < C < A$. Por esta razón, se debe tener un solo orden de los componentes de los vectores de contexto para evitar que las comparaciones seán erróneas. Es importante mencionar que las comparaciones se realizan de forma local por lo que el orden debe ser único en cada sitio, pero puede variar entre diferentes sitios, i.e., los mismos vectores se podrían representar de manera distinta en diferentes sitios.

Solución: la clase `UserData` sirve para registrar a los usuarios tan pronto como se sepa de su existencia. Cada usuario tiene una dirección MAC diferente. Al registrar un nuevo usuario con su dirección MAC en la clase `UserData`, se le asigna un entero que lo identifica localmente de forma única. Este entero será el índice correspondiente a dicho usuario en cada vector de contexto. Puesto que existe el riesgo de que dos hilos intenten registrar concurrentemente a un usuario diferente, la clase `UserData` usa métodos de tipo *synchronized*[1].

Incremento de componentes inexistentes en vectores

Problema: Puesto que se permite el uso transparente de vectores con diferentes tamaños, se podría necesitar incrementar un componente que no está presente en un vector dado. Antes de continuar se debe tener en cuenta que los índices de los vectores de contexto comienzan en 0, la explicación detallada de los vectores de contexto se presenta en el Apéndice A.

Para ilustrar este problema considere que se tienen inicialmente dos usuarios y se genera el siguiente vector: $v = [3, 4]$ al producir una nueva operación. Posteriormente, se conecta un nuevo usuario, por lo tanto los siguientes vectores que se generen serán de tamaño 3. Sin embargo, el vector v requiere ser incrementado en el componente 2 durante un procedimiento de transformación, pero este vector solo tiene los componentes 0 y 1, por lo tanto se produciría un error al intentar incrementar un componente inexistente.

Solución: la clase `ContextVector` abstrae el tamaño real de los vectores, por lo que todos los vectores se pueden considerar del mismo tamaño para comparaciones e incrementos. Internamente, esta clase considera 0 a los componentes faltantes y

[1]Palabra reservada del lenguaje Java que mantiene exclusión mutua por medio de un monitor, el cual solo permite que un hilo entre a los métodos o modifique los campos marcados *synchronized*, hasta que dicho hilo salga de todos los métodos *synchronized* o renuncie al monitor. Se tiene un monitor por cada objeto.

cuando desea incrementar uno de estos componentes, crea un nuevo vector, el cual será una copia del anterior pero colocará en 0 los componentes faltantes. En consecuencia, al incrementar un componente no se producirán errores. Este procedimiento interno de crear nuevos vectores no es percibido de manera externa.

5.2.2 Clase ObjectState

El estado del objeto compartido es el conjunto de las operaciones que se han ejecutado sobre dicho objeto. Este conjunto es mantenido por la clase `ObjectState`. Cuando se genera localmente una nueva operación, su contexto es el estado del objeto compartido que se tiene en ese momento. Esta clase genera vectores de contexto que representan el estado del objeto compartido. Para generar estos vectores se necesita el número de usuarios conocidos hasta el momento. La cantidad de usuarios también afecta la estructura interna de esta clase, pues las operaciones almacenadas están agrupadas por el usuario que las generó, con el fin de obtener una mayor eficiencia cuando se deseé buscar una operación específica.

Debido a que los procedimientos internos de esta clase tienen una gran dependencia con el número de usuarios conocidos, estos procedimientos y los que cambian dicho número se realizan con exclusión mutua.

5.3 Transformación Operacional

Todos los objetos que se encuentran en el paquete `sharedobject` interactúan para llevar a cabo la Transformación Operacional.

La estructura de un sistema de Transformación Operacional se puede dividir en dos partes: el algoritmo de control y las funciones de transformación. El algoritmo de control elegido para la presente tesis es el algoritmo COT, el cual decide qué operaciones requieren transformación, contra qué operaciones se deben transformar y en qué orden. Las funciones de transformación realizan las transformaciones reales sobre los parámetros de una operación dada. Estas funciones de transformación dependen del tipo de dato presentado por el objeto compartido.

El algoritmo COT no requiere toda la información de una operación O, solo el contexto $C(O)$ de la operación O y la operación original de O denotada por $org(O)$; estos conceptos fueron definidos en el Capítulo 3. Los parámetros de la operación pueden ser cualesquiera. Dicho algoritmo tampoco especifica cómo se realizan las transformaciones, simplemente llama al procedimiento $IT(Oa, Ob)$. Por esta razón, el algoritmo COT puede funcionar con operaciones abstractas que tengan la información antes mencionada y teniendo disponible algún procedimiento $IT(x, x)$. Este procedimiento será delegado al objeto compartido, puesto que las transformaciones son específicas del tipo de dato que presentado dicho objeto, el cual también es responsable de ejecutar las operaciones sobre sí mismo.

Un objeto compartido es intercambiable debido a que el algoritmo COT es completamente independiente de las particularidades de dicho objeto. De hecho, el paquete

`sharedobject` no contiene ningún objeto compartido real, solo una clase abstracta `SharedObject` que oculta de la clase `Cot` los detalles del objeto compartido que se usa realmente. La clase `Operation` también es abstracta, así que no representa un tipo particular de operación, sino solo tiene los datos necesarios para el algoritmo COT. Para hacer uso de este algoritmo se requiere una implementación de las clases abstractas, la cual se describe en la Sección 5.4. En el resto de esta sección se describe en detalle las clases del paquete `sharedobject`.

5.3.1 Clase Cot

Método	Entrada	Salida
`enqueuOperation`	Un objeto `Operation` que será agregado a la cola y de ser posible transformado y ejecutado	`void`
`cotDo`	Un objeto `Operation` que representa la operación que será transformada y ejecutada	`void`
`transform`	Un objeto `Operation` que será transformado contra las operaciones de un objeto de `OperationSet`, implementa el Algoritmo 3.6	`void`

Tabla 5.1: Métodos de la clase `Cot`

Esta clase implementa el algoritmo COT y una cola de operaciones pendientes, que es necesaria puesto que este algoritmo no mantiene por sí mismo la condición CC1 definida en la Sección 3.5. En la Tabla 5.1 se muestran los principales métodos de la clase `Cot`. La cola es visible gracias al método `enqueuOperation`, que es el único método público, i.e., es el único punto de entrada de operaciones. Debido a que este método es sincronizado, se garantiza que solo un hilo estará ejecutando el algoritmo COT en un objeto compartido dado. El método sincronizado es importante pues se tienen múltiples hilos que podrían intentar ejecutar y posiblemente transformar operaciones de forma concurrente.

El funcionamiento del método `enqueuOperation` es el siguiente: primero se verifica que la operación recibida sea válida para el procedimiento `cotDo`, i.e., la operación cumple la condición CC1. En caso de que la operación no sea válida será encolada. Si la operación cumple la condición CC1, se llama al procedimiento `cotDo`, el cual está detallado en la Sección 3.6 del Capítulo 3, este procedimiento determina que transformaciones es necesario realizar y utiliza otro procedimiento para realizarlas. Después de que se completa este procedimiento, la operación ya fue transformada y ejecutada. Al tener una operación más en el estado del objeto compartido, existe la posibilidad de que una de las operaciones encoladas cumpla ahora la condición CC1 por lo que se revisa la cola. Si se encuentran operaciones que cumplen CC1, serán pasadas al procedimiento `cotDo` hasta que la cola esté vacía o ninguna operación cumpla dicha condición.

El método `cotDo` y el método `transform` implementan los procedimientos del mismo nombre que están descritos en el Capítulo 3.

5.3.2 Clase abstracta SharedObject

Método	Entrada	Salida
`inclusionTransformation`	Un objeto `Operation` que será transformado y otro objeto `Operation` contra el que se transforma el primero	Un objeto representando la operación transformada
`execute`	Un objeto `Operation` que será pasado a `executeImplementation` y agregado al estado del objeto compartido	`void`
`executeImplementation`	Un objeto `Operation` que será ejecutado	`void`

Tabla 5.2: Métodos de la clase abstracta `SharedObject`

Esta clase funciona como una interfaz entre el algoritmo COT y los objetos compartidos concretos. Puesto que es una clase abstracta no es posible instanciarla, únicamente sus subclases pueden ser instanciadas. Además de la interfaz visible para la clase `Cot`, la clase `SharedObject` tiene funcionalidad común a todos los objetos compartidos. Los métodos de esta clase se muestran en la Tabla 5.2

La interfaz antes mencionada consta de dos métodos, `inclusionTransformation` y `executeImplementation`. Estos métodos corresponden a las funciones utilizadas por los algoritmos 3.5 y 3.6 del Capítulo 3 con los nombres $IT(Oa, Ob)$ y $execute(Op)$. El método `inclusionTransformation` desempeña la transformación. Dado que es abstracto, las subclases concretas de `SharedObject` lo deben implementar, cada una de acuerdo a su tipo de dato particular. El método `execute` no es abstracto pues incluye la funcionalidad de agregar al estado del objeto compartido las operaciones ejecutadas, sin embargo internamente llama al método `executeImplementation`. Este método si es abstracto así que debe ser implementado por cada subclase concreta de `SharedObject`, de acuerdo al tipo de dato representado y a sus operaciones.

Como se puede ver en los métodos, los parámetros son de la clase `Operation`, la cuál también es un clase abstracta, por lo que junto con el objeto compartido concreto, también se debe implementar operaciones concretas, que son subclases de `Operation`. Estas subclases se usan en los objetos compartidos para decidir qué transformaciones hacer y qué efectos tendrá la ejecución de cada operación. La clase abstracta `Operation` también incluye parámetros, con el fin de facilitar la reutilización de estos datos comunes. Aunque se tengan estos parámetros, en ausencia de una subclase no es posible identificar qué operación se representa realmente.

La clase `SharedObject` implementa la interfaz `Observable`, que está relacionada con el uso de objetos `Observer`. Estas clases e interfaz son explicadas en detalle en la Sección 5.5.

5.3.3 Clase ObjectState

Método	Entrada	Salida
`setMinus`	Resta un `ContextVector` del estado actual del objeto compartido	El conjunto resultante representado por un objeto `OperationSet`
`contextDifference`	Resta un `ContextVector` de otro `ContextVector`	El conjunto resultante representado por un objeto `OperationSet`
`addOperation`	Agrega un objeto de `Operation` al estado del documento	`void`

Tabla 5.3: Métodos de la clase `ObjectState`

La clase `Cot` necesita utilizar la información de la clase `ObjectState`, pero no lo hace directamente porque las instancias de `ObjectState` son administradas por la clase `UserData`, con el fin de mantener la integridad de los datos frente a accesos concurrentes locales. La clase `UserData` proporciona *wrappers* para los métodos de la clase `ObjectState` mostrados en la Tabla 5.3. El método `setMinus` recibe un vector de contexto y realiza una resta de conjuntos; específicamente, substrae el contexto del estado actual del objeto, dando como resultado el conjunto $DS/C(Op)$. El método `contextDifference` realiza la resta de dos conjuntos que están representados por vectores de contexto. Esta resta da como resultado el conjunto $C(Oa)/C(Ob)$.

Hay que tener en cuenta el estado del objeto compartido y los contextos son conjuntos de operaciones y conceptualmente se pueden restar, pero sus estructuras internas varían, particularmente los contextos están representados por vectores de contexto, que se explican en el Apéndice A y el estado del objeto compartido está representado por la clase `ObjectState`.

La estructura interna de la clase `ObjectState` es un arreglo de arreglos de operaciones como se muestra en la figura 5.2. En el arreglo externo por cada usuario se tiene un componente, el cual almacena un arreglo interno, cada arreglo interno a su vez almacena las operaciones generadas por dicho usuario. Las operaciones están almacenadas en el orden en que fueron producidas. Esta estructura permite que: 1) la búsqueda de una operación solo requiera dos consultas y 2) las restas de conjuntos sean más eficientes.

La figura 5.2 muestra un ejemplo de resta de conjuntos, aun que conceptualmente se está restando conjuntos, los parámetros de entrada son vectores de contexto, los cuales representan conjuntos de operaciones. Recuérdese que cada componente de un

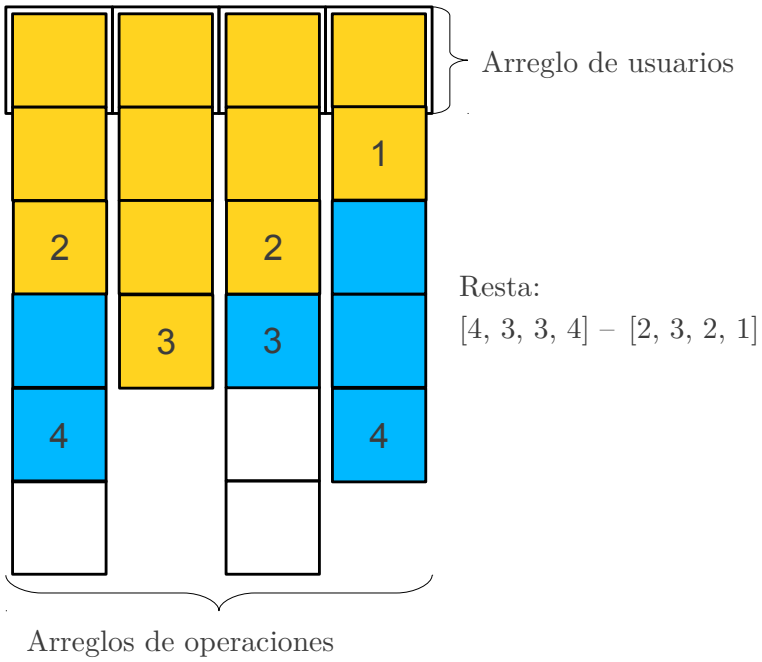


Figura 5.2: Estructura interna de la clase `ObjectState`

vector de contexto corresponde a un usuario. El primer arreglo de operaciones de la figura 5.2 corresponde al primer usuario y el primer componente de los vectores de contexto también corresponde al primer usuario. El primer componente de cada vector de contexto y el arreglo de operaciones del primer usuario se usarán en la primera iteración del procedimiento. En esta iteración el componente del vector de contexto sustraendo es 2 y el componente del vector de contexto minuendo es 4, por lo que se consulta el arreglo de operaciones desde 3 hasta 4 incluso. Se agregará al conjunto resultante todas las operaciones consultadas. Las cuales se muestran en un color obscuro en la figura 5.2. Cada iteración se cambia al siguiente componente de los vectores de contexto y al siguiente arreglo de operaciones. Este procedimiento requiere que ambos vectores de contexto sean subconjuntos del estado del objeto compartido pero este requisito siempre se cumple debido a que la condición CC1 se asegura antes de entrar al algoritmo COT.

El último método mencionado, `addOperation`, solo agrega una operación en su posición correspondiente, i.e., en el arreglo de operaciones correspondiente al usuario que la produjo y en la posición que le corresponde según el orden en el que se generó. `UserData` garantiza que el arreglo de usuarios tenga el tamaño correcto y que no ocurran errores por insertar en posiciones no válidas.

Método	Entrada	Salida
`removeByCC4`	Un objeto `Operation` para la comparación de la condición CC4	Un objeto `Operation` del conjunto que cumple CC4
`addElement`	Un objeto `Operation` que será agregado al conjunto	`void`

Tabla 5.4: Métodos de la clase `OperationSet`

5.3.4 Clase OperationSet

Aunque la mayoría de los conjuntos de operaciones usados en el algoritmo COT son representados con vectores de contexto, existen algunas restas de conjuntos que dan como resultado conjuntos que no se pueden representar por vectores de contexto. Para tratar estos casos se implementó la clase `OperationSet`, en la cual se abstrae un conjunto de operaciones y se proveen los métodos para agregar y remover elementos y se muestran en la Tabla 5.4. Está clase también provee un método requerido por el algoritmo COT, el cual corresponde a la línea 2 del algoritmo 3.6. Este método es `removeByCC4` y su función es remover un elemento de un conjunto pero siempre y cuando se cumpla la condición CC4 (definida en la Sección 3.5). Internamente, se comparan los contextos y se selecciona una operación que tenga un contexto menor o igual que el contexto de la operación pasada como parámetro. Finalmente, la operación seleccionada es removida del conjunto y devuelta.

5.4 Clases concretas SharedTextBuffer

El *buffer* de texto compartido permite editar texto sin formato. En este modelo de datos se tiene una secuencia de caracteres, en la que cada uno ocupa una posición referenciada por un entero empezando desde 0. Para modificar el contenido del *buffer*, se utilizan las siguientes dos operaciones:

- **Insertar(s, p)**: esta operación inserta una cadena de caracteres s en una posición p, recorriendo todos los caracteres del *buffer* que ocupaban una posición mayor o igual que p.
- **Borrar(n, p)**: esta operación borra n caracteres consecutivos empezando en la posición p; los caracteres en posiciones mayores o iguales que $p+n$ son recorridos hacia la izquierda n posiciones.

Se ha probado que editores como vi y Emacs pueden ser implementados usando únicamente estas dos primitivas [Knister and Prakash, 1993], [Ressel et al., 1996] y [Valdes, 1993].

La implementación del *buffer* de texto compartido consta de tres clases concretas: la clase `SharedTextBuffer` que representa al *buffer* de texto e implementa los métodos abstractos de la clase `SharedObject` así como de las clases `OperationInsert` y

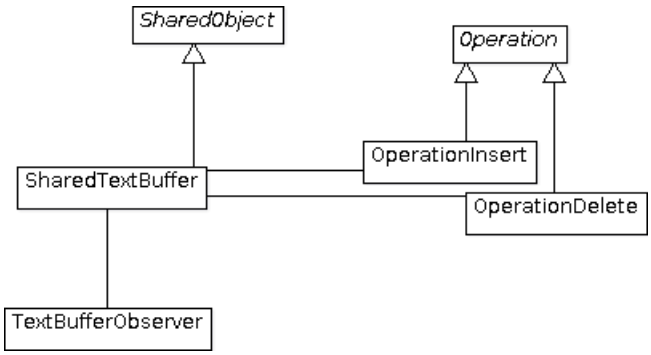


Figura 5.3: Clases concretas de *buffer* de texto compartido

`OperationDelete`, las cuales representan las dos operaciones posibles sobre el *buffer* de texto. Estas clases se muestra en la figura 5.3.

El lenguaje Java provee una clase que implementa un *buffer* de texto simple. Dicha clase llamada `StringBuffer` tiene las funciones de insertar y borrar caracteres, por lo que solo es necesario implementar las transformaciones e identificar las clases concretas de las operaciones recibidas.

Los métodos `inclusionTransformation` y `executeImplementation` reciben operaciones de la clase abstracta `Operation`. En consecuencia, la clase `SharedTextBuffer` debe identificar la clase concreta de estos objetos para después poder efectuar la acción pertinente. Para poder realizar esta identificación, se utiliza el método `isAssignableFrom` y se compara la clase de cada objeto recibido por dichos métodos, desde la clase `Cot`, con las subclases `OperationDelete` y `OperationInsert`. Una vez identificada la clase correcta de las operaciones de entrada, éstas se asignan a una variable de la clase correcta y se continúa con la transformación.

Cada uno de los tipos de operaciones se puede transformar contra todos los tipos de operaciones, incluida la combinación de un tipo contra el mismo tipo. Por esta razón, el número de posibles transformaciones es el cuadrado del número de operaciones. Estas combinaciones se pueden presentar en una matriz cuadrada. En la Tabla 5.5 se muestra la matriz de combinaciones para las operaciones `OperationDelete` (D) y `OperationInsert` (I).

I contra I	D contra I
I contra D	D contra D

Tabla 5.5: Posibles combinaciones de transformaciones

La implementación de estas transformaciones se basa en el pseudo-código propuesto por Sun y Ellis [Sun and Ellis, 1998], quienes sugieren que las operaciones de tipo `OperationDelete` se traslapen, i.e., cuando más de un usuario borra el mismo carácter, solo la primera operación de supresión ejecutada tendrá efecto. Respecto a

las cadenas insertadas, todas deben tomarse en cuenta, aun cuando se inserten en una región que fue borrada concurrentemente.

Para ejemplificar el procedimiento de transformación se utilizará la siguiente notación: la operación op_a será transformada contra la operación op_b en todos los ejemplos, i.e., $op'_a = IT(op_a, op_b)$. Ambas operaciones tienen un parámetro de posición, llamado p_a en op_a y p_b en op_b, el cual existe independientemente del tipo de operación. Con el fin de facilitar la comprensión de los ejemplos se puede suponer op_b ya fue ejecutada localmente y que op_a es recibida desde la red y necesita ser transformada.

Cuando $p_b > p_a$, la transformación no tiene ningún efecto sobre p_a, cuando $p_b < p_a$ la transformación si tendrá efecto sobre p_a, el cual depende del tipo de la operación op_b, pero es independiente del tipo de la operación op_a. Cuando $p_a = p_b$ y cuando el efecto de las operaciones op_a y op_b se traslapa, se presentan casos particulares que se explican posteriormente.

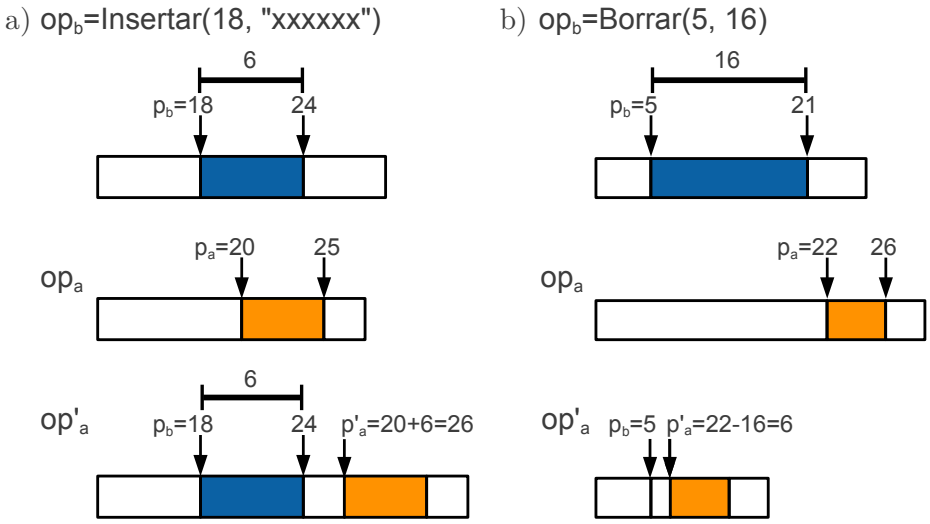


Figura 5.4: Ejemplos de transformaciones

Las operaciones de inserción recorren p_a hacia la derecha, una cantidad de posiciones igual a la longitud de la cadena insertada por op_b. Para ilustrar esta situación véase el ejemplo de la figura 5.4-a, tome en cuenta que el ejemplo funciona igual si op_a es inserción o supresión y por simplicidad se omite esta información. En este ejemplo la operación op_b es de tipo inserción y las posiciones son: $p_b = 18$ y $p_a = 20$. Puesto que $p_b < p_a$ la posición resultante de la transformación (p'_a) si será diferente de p_a, en este caso $p'_a = 26$, ya que p_a fue recorrido a la derecha 6 posiciones, debido a que la longitud de la cadena insertada por op_b es 6.

Las operaciones de supresión recorren p_a hacia la izquierda, una cantidad de posiciones igual a la cantidad de caracteres borrados por op_b, una operación de supresión no puede recorrer p_a hacia la izquierda más allá de p_b. Para ilustrar esta situación véase el ejemplo de la figura 5.4-b, este ejemplo también se comporta igual indepen-

dientemente del tipo de op_a. En este ejemplo la operación op_b es de tipo supresión y las posiciones de p_b y p_a son 5 y 22 respectivamente. La cantidad de caracteres borrados por op_b es 16, por lo que el parámetro p_a deberá ser recorrido a la izquierda 16 posiciones, resultando en $p'_a = 6$.

En el caso particular cuando $p_a = p_b$ en dos operaciones de inserción, se debe decidir cuál de las dos cadenas insertadas debe ir primero. Esta decisión debe tomarse de igual forma en todos los sitios. La clase `UserData` provee un método llamado `breakTie`, que utiliza las direcciones de hardware de los adaptadores de red (MAC) para decidir en qué orden se insertan las cadenas. Cuando la operación op_a es de supresión y op_b es de inserción p_a se recorre igual que en el caso normal, ya que op_a no debe borrar ninguno de los caracteres insertados por op_b y que op_a desconocía. En el caso contrario, cuando op_b es de supresión, p_a no cambia pues la supresión no altera la intensión del usuario que generó op_a, i.e., insertar una cadena en p_a. El caso en el que $p_a = p_b$ y tanto op_a como op_b son de supresión se produce un traslape y se describe a continuación.

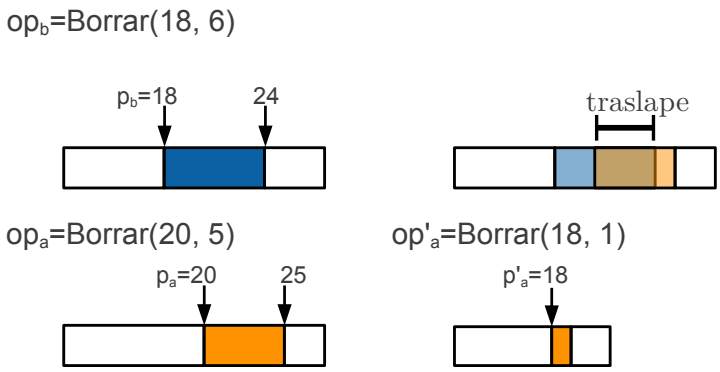


Figura 5.5: Ejemplos de transformaciones con traslape

Cuando dos operaciones de supresión se traslapan, la operación transformada borrará menos caracteres. En la figura 5.5 se muestra un ejemplo de cómo se traslapan dos operaciones de supresión. Las operaciones op_a y op_b afectan a los caracteres que van de la posición 20 a la 24. Puesto que la operación op_b borra primero estos caracteres, la operación op_a no puede volver a borrarlos y no debe borrar caracteres que no sean los que el usuario tenía la intención de borrar. La región traslapada se muestra en la figura 5.5. En la operación op_a solo queda un carácter por borrar, que es el que estaba en la posición 24. Sin embargo, debido al efecto de op_b, este carácter fue recorrido a la posición 18, por lo que p_a es movida a la izquierda, quedando $p'_a = 18$. El número de caracteres a borrar por op'_a ahora será uno. En algunos casos el número de caracteres a borrar se puede volver cero, porque todos los caracteres de la operación op_a podrían haber sido borrados con antelación.

Cuando se tiene una operación de inserción y una de supresión que afectan el mismo rango, se debe mantener el efecto de los caracteres insertados y borrados, i.e, ninguno de los caracteres insertados concurrentemente debe ser borrado, ya que este caso puede llevar a situaciones muy particulares, es el que causa más complicaciones en la implementación. Estás situaciones se describen a continuación.

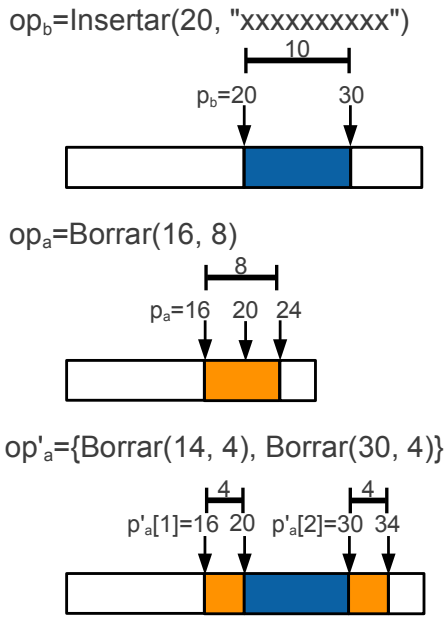


Figura 5.6: Transformación de supresión contra inserción en el mismo rango

En la figura 5.6 se muestra un caso particular en el que se borran caracteres y se inserta una cadena en la misma región. El resultado de la transformación debe cumplir dos objetivos: 1) todos los caracteres que señala la operación op_a deben ser borrados y 2) la cadena que inserta op_b debe permanecer integra. La operación op_a no puede afectar los caracteres insertados por op_b puesto que el usuario no los conocía al momento de emitir la operación de supresión. Se puede ver en la figura 5.6 que el valor de op_b está dentro del rango de caracteres que serán borrados, i.e, $16 < 20 < 24$. Cuando op_a sea transformada no se puede recorrer p_a por que los caracteres de 16 a 19 (incluso) continúan en la misma posición, sin embargo, op_a también intenta borrar los caracteres de 20 a 23 (incluso), no se puede permitir que op'_a borre estos últimos caracteres pues son parte de los caracteres insertados por op_b, los últimos cuatro caracteres que op_a intenta borrar ahora se encuentran en la posición 30, pues fueron recorridos 10 posiciones por op_b. Los primeros cuatro caracteres que op_a pretende borrar aún se encuentran en la misma posición.

En resumen, los caracteres que debe borrar op'_a ahora son discontinuos, debido a esta situación, las operaciones de supresión deben poder indicar más de un rango de caracteres a borrar, como se muestra en op'_a al final de la figura 5.6. Para representar múltiples rangos, la clase `OperationDelete` implementa una lista ligada.

5.5 Observers

Idealmente, el programador que utilice la biblioteca no debe tener acceso a ninguna instancia de `SharedObject` para prevenir que intervenga con el funcionamiento interno de la biblioteca CDCDM. Sin embargo, el programador necesita saber el contenido del objeto compartido para desplegar los cambios en pantalla. Estos cambios se puede conocer por medio del patrón de diseño *Observer* [Gamma et al., 1995]. Mediante este patrón, los objetos son notificados en el momento en que ocurre un cambio, en vez de que necesiten estar revisando constantemente en busca de cambios. La biblioteca CDCDM hace uso de este patrón de la siguiente manera:

1. La biblioteca CDCDM define una clase abstracta `Observer` y una interfaz `Observable`.

2. Las clases que deben hacer estas notificiones implementan la interfaz `Observable`, i.e., definen funcionalidad para los métodos especificados por la interfaz.

3. El programador implementa una clase concreta `Observer`, con conocimiento específico del objeto compartido concreto a observar o utiliza una de las subclases concretas de `Observer` proporcionadas por la biblioteca.

4. El programador registra los objetos `observers` necesarios en la biblioteca CDCDM.

5. En tiempo de ejecución, los objetos que implementan la interfaz `Observable` notifican a los objetos de la clase `Observer` que tienen registrados.

6. El código de la GUI despliega la información pertinente.

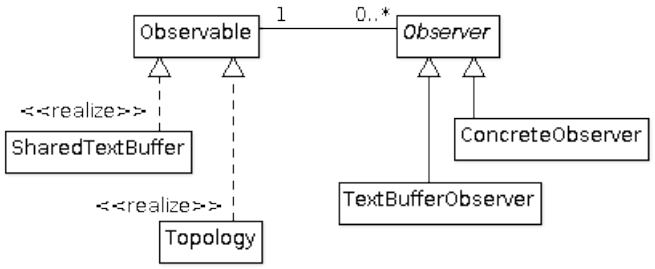


Figura 5.7: Clases relacionadas con el patrón de diseño *Observer*

En la figura 5.7 se muestran las clases antes mencionadas, así como las subclases concretas de la clase abstracta `Observer` y las implementaciones de la interfaz `Observable`.

Método	Entrada	Salida
`registerObserver`	Un objeto `Observer` que será notificado de aquí en adelante	`void`
`unregisterObserver`	Un objeto `Observer` que ya no será notificado de aquí en adelante	`void`
`notifyObservers`	`void`	`void`

Tabla 5.6: Métodos de la interfaz `Observable`

Las implementaciones de la interfaz `Observable` deben proporcionar los siguientes métodos mostrados en la Tabla 5.6. Los primeros dos métodos permiten registrar y eliminar objetos `Observer`. Se pueden tener varios objetos `Observer` en un solo objeto `Observable`. El método `notifyObservers` debe ser llamado por una clase que implementa `Observable` cuando ocurra un cambio interno. Dentro de este método se debe llamar al método `notifyChange` de todos los objetos `Observer` que se tengan registrados.

El método `notifyChange` no proporciona ninguna información a `Observer`, solo informa que ocurrió un cambio. Las clases que implementan `Observer` podría tener conocimiento de la clase concreta `Observable` y extraer la información necesaria para copiarla a otro objeto. Esto es lo que hace la clase `TextBufferObserver`, la cual observa únicamente instancias de la clase `SharedTextBuffer`, utiliza el método `getString` para obtener el contenido del objeto `SharedTextBuffer` y lo copia a un objeto `TextTextBox` que se encuentre en la GUI.

La clase `ConcreteObserver` funciona con cualquier objeto que implemente `Observable`. En su método `notifyChange` únicamente llama al método `notify` de un objeto cualquiera que le fue proporcionado en su constructor. Este objeto debe tener un hilo bloqueado por medio del método `wait`. Dicho hiló será despertado cuando se llame `notify`. El objeto que tiene el hilo bloqueado deberá conocer la clase que implementa `Observable` para extraer directamente la información necesaria. La clase `ConcreteObserver` es más general, pero requiere que el programador tenga conocimiento del objeto `Observable` y decida cómo consultarlo.

La clase `Topology` se explica en el Capítulo 6 pues forma parte de la capa de red *ad-hoc*. Particularmente, esta clase notifica cuando hay cambios de usuarios, i.e., conexiones, desconexiones y reconexiones.

Capítulo 6

Diseño detallado de la capa de red *ad-hoc*

En este capítulo se describe el diseño detallado de la capa de red *ad-hoc* que se encarga de difundir los datos con los que trabajan los usuarios. Se describen todas las clases involucradas en esta capa y cómo interactúan entre sí para lograr su objetivo. En la Sección 6.1 se explica de forma general las partes principales de esta capa. Después en la Sección 6.2 se detalla cómo se tratan las conexiones y desconexiones, mientras que en la Sección 6.3 se explica cómo se envían y reciben los datos. Finalmente, en la Sección 6.4 se describe cómo se procesan los mensajes que son recibidos desde la red.

6.1 Vista general de la capa de red *ad-hoc*

La mayor parte de esta capa se encuentra en el paquete `adHoc` mostrado en la figura 6.1. También se muestran las clases externas que son necesarias para su funcionamiento.

Dos de las clases externas están relacionadas con los adaptadores de red y se encuentran en los paquetes `tcpSocket` para conexiones por TCP/IP en redes Wi-Fi y `bluetooth` para conexiones Bluetooth. También se utiliza la clase `UserData` para registrar datos de usuario necesarios para otras partes de la biblioteca CDCDM, e.g., la dirección MAC de un usuario.

En el centro de la figura 6.1 aparece la clase `Topology`, la cual está relacionada con varías de las clases de este paquete, debido a que almacena la información de la red, que consiste en la lista de los demás nodos de la red y el usuario que le corresponde a cada nodo. Un objeto de la clase `Peer` guarda la información de un nodo, la cual es utilizada para el ruteo y el soporte de la conciencia de grupo, cuando se muestra que un usuario se ha conectado o desconectado y se consulta la lista de usuarios conectados. Un objeto de la clase `Topology` siempre es informado de estos eventos y es responsable de notificar a las instancias de la clase `Observer` sobre estos cambios. Esta clase no tiene otra funcionalidad más que almacenar y organizar los datos mencionados.

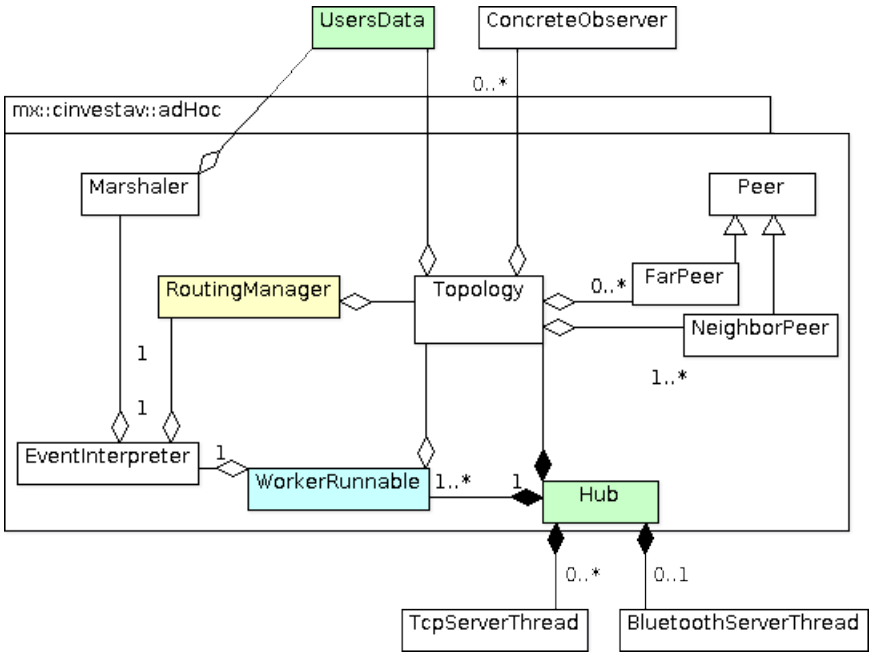


Figura 6.1: Diagrama de clases del paquete adHoc

La clase `Hub` implementa el patrón de diseño *Facade* [Gamma et al., 1995], el cual consiste en usar un solo método para agrupar llamadas a varios métodos en diferentes objetos. Cada vez que se quiere realizar un procedimiento, únicamente se necesita llamar el método *Facade*, el cual internamente hace todas las llamadas necesarias. Por esta razón la clase `Hub` tiene relaciones de composición con varias clases de esta capa, ya que llama métodos de estas clases. La clase `Hub` funciona como el punto de entrada de la capa de red *ad-hoc*.

La clase `WorkerRunnable` puede tener muchas instancias y cada una representa una conexión de entrada. A estas conexiones llegan bytes que son ordenados en mensajes. Una vez que se tiene un mensaje completo, este es pasado a una instancia de la clase `EventInterpreter`, la cual concentra los procedimientos pertinentes para procesar cada mensaje.

La clase `RoutingManager` implementa primitivas para el ruteo y envío de mensajes y únicamente incluye la funcionalidad de estas primitivas, pero no decide cuando se deben usar. Esta decisión es tomada por la clase `EventInterpreter`. Para ejecutar las primitivas se requiere la información mantenida por `Topology`. Las primitivas son ejecutadas en otro hilo dentro de la clase `RoutingManager` para que las llamadas a los métodos de estas primitivas no se bloquen y regresen lo antes posible.

Finalmente, la clase `Marshaller` se encarga de convertir arreglos de bytes, recibidos como mensajes, en objetos que son posteriormente despachados a las capas superiores.

Las funciones de esta capa se explican en detalle en las secciones 6.2 - 6.4 Manejo de mensajes recibidos. Es importante destacar que esta división no separa las clases de manera excluyente, ya que algunas clases toman parte en más de una de las fun-

cionalidades.

6.2 Conexión y desconexión de usuarios

Los eventos de conexión y desconexión de usuarios modifican varios aspectos importantes. En primer lugar, la topología de la red *ad-hoc*, i.e., cuáles nodos son alcanzables, por qué ruta y cuáles no lo son. En segundo lugar, la conexión de un nuevo usuario modifica el tamaño de los vectores de contexto, como se explicó en el capítulo 5. Finalmente, estos eventos afectan a la conciencia de grupo, porque las aplicaciones deben mostrar a cada usuario qué usuarios están conectados en un momento dado, cuándo se conectan usuarios y cuándo se desconectan.

En los procedimientos de conexión y desconexión siempre hay dos partes: 1) un nodo que entra o sale y 2) la red *ad-hoc* a la cual entra el nodo o de la que sale. El nodo es el que inicia individualmente los procedimientos y cada nodo de la red *ad-hoc* responde. Los procedimientos en el nodo individual serán llamados individuales, i.e., conexión individual y desconexión individual. Los procedimientos en la red *ad-hoc* serán llamados colectivos y se ejecutan en uno o más nodos. Estos nodos pueden ser nodos vecinos o nodos no vecinos, pero el procedimiento colectivo varía entre los vecinos y los no vecinos.

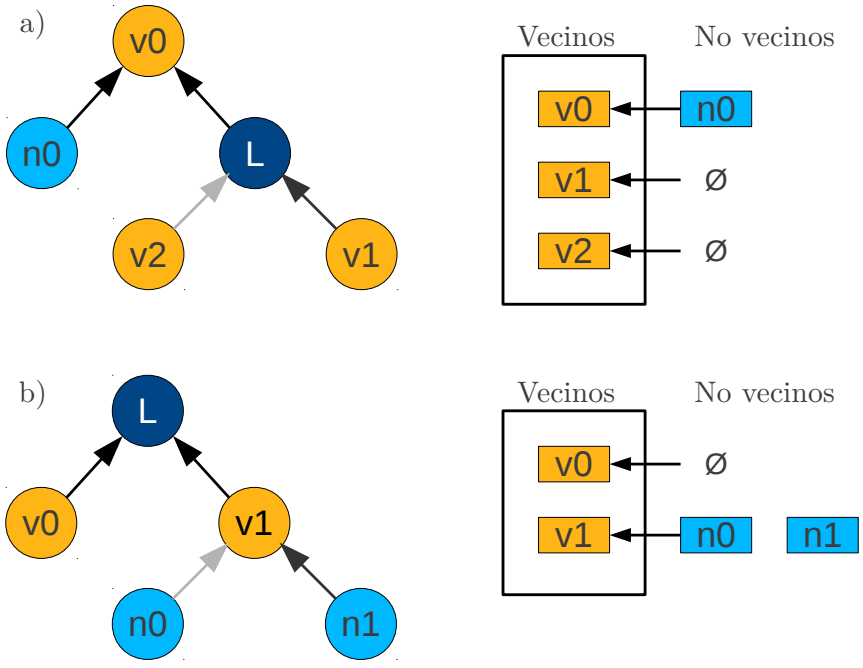


Figura 6.2: Diferencias entre nodos vecinos y no vecinos

En la figura 6.2 se muestra un ejemplo de una red *ad-hoc* y la perspectiva de dos nodos, respecto a que nodos son sus vecinos y cuales no. En cada perspectiva, el nodo

local se nombra L, los nodos vecinos v y los no vecinos n. En el ejemplo a) L tiene tres vecinos y en el ejemplo b) L tiene dos vecinos.

6.2.1 Clase Hub

Método	Entrada	Salida
`disconnect`	`void`	`void`
`registerNeighbor`	Un objeto `NeighborPeer` que representa un vecino recién conectado	`void`

Tabla 6.1: Métodos de la clase `Hub`

Esta clase sigue el patrón de diseño *Facade* [Gamma et al., 1995], ya que reúne las instancias de las demás clases involucradas en el proceso de desconexión individual. Además, mantiene las referencias a las instancias de cada clase y actúa como punto de entrada y como concentrador de los objetos y métodos a llamar.

La clase `Hub` se encarga de inicializar la mayoría de los objetos de la capa *ad-hoc*. Externamente, solo se llama al constructor y se proporciona una instancia de la clase `UserData`.

En los procedimientos de conexión individual y colectiva, esta clase no desempeña actividades, sin embargo, en la desconexión individual coordina la finalización de todos los hilos de esta capa. Para iniciar este procedimiento únicamente se llama al método `disconnect` mostrado en la Tabla 6.1, el cual llama internamente a los métodos de desconexión de los objetos de las clases: `EventInterpreter`, `RoutingManager` y `WorkerRunnable`. Después, el método `disconnect` espera a que todos los hilos en estos objetos terminen antes de regresar. De esta forma, este método evita que la aplicación termine antes de que algunos objetos hayan hecho lo necesario para desconectarse correctamente.

6.2.2 Clase Topology

Esta clase es un registro de nodos y rutas. Todos los cambios en la red se le deben notificar. Esta clase por sí sola no desempeña actividades, únicamente almacena y organiza información.

En los procedimientos colectivos esta clase debe actualizar sus listas. Contiene una lista de nodos vecinos representados por la clase `NeighborPeer` y por cada nodo vecino mantiene una lista de los nodos que están conectados a ese vecino. Estos nodos están representados por la clase `FarPeer`, ya que almacenan menos información. Ambas clases son subclases de `Peer`, la cual contiene la información común, e.g., dirección MAC y nombre amigable. En el ejemplo de la figura 6.2 se muestra estas listas del lado derecho.

En un evento de conexión, la clase `Topology` agrega un objeto de la clase `Peer` que representa al nodo que se conecta y en una desconexión lo mueve a una lista de "inalcanzables" para denotar que este nodo ya no forma parte de la red.

Esta clase implementa la interfaz `Observable` por lo que tiene la responsabilidad de notificar eventos a las instancias de la clase *Observer* que tenga registradas. Estos eventos son las conexiones y desconexiones de nodos. No hay una subclase de `Observer` específica para la clase `Topology` por lo que se utiliza la clase `ConcreteObserver`, la cual fue explicada en la Sección 5.5.

6.2.3 Clase WorkerRunnable

Esta clase sirve para aislar la petición de una conexión de la ejecución de la misma. Esta separación es necesaria pues la ejecución consiste de un ciclo infinito de recepción de datos. Las peticiones en cambio deben tomar poco tiempo porque se pueden presentar en todo momento. El objeto que atiende estas peticiones no debe estar ocupado. Por esta razón, un hilo recibe las peticiones y crea otros hilos de la clase `WorkerRunnable` para ejecutarlas.

Puesto que esta clase es el punto de entrada de los datos recibidos desde otros nodos, participa en todos los procedimientos de conexión y desconexión. En los procedimientos de conexión individual y colectiva de los nodos vecinos, la clase `BluetoothServerThread` crea un hilo `WorkerRunnable` que recibe los datos necesarios para registrar, en la clase `Topology`, al nodo recién conectado. Posteriormente, el hilo `WorkerRunnable` inicia su ciclo principal. En el procedimiento de desconexión colectivo, este hilo recibe el mensaje de desconexión, el cual es procesado por la clase `EventInterpreter` que realiza el resto de las acciones necesarias. Si el nodo que se desconectó era vecino, el hilo `WorkerRunnable` ya no es necesario y termina. En el procedimiento de desconexión individual, este hilo espera un acuse de recepción del nodo al que está conectado antes de terminar, con el fin de no cerrar la conexión antes de que el nodo vecino esté avisado de la desconexión.

La clase `WorkerRunnable` también tiene la función de detectar desconexiones no intencionales, las cuales se identifican cuando ocurre una excepción en los métodos de lectura de la conexión de entrada.

6.2.4 Clase BluetoothServerThread

Esta clase ejecuta un hilo que se bloquea esperando nuevas conexiones en el adaptador Bluetooth. Cuando estas conexiones son recibidas, se toman los datos necesarios para describir al nuevo nodo, i.e., el nombre amigable Bluetooth y la dirección MAC del nuevo nodo. Estos datos son pasados a una nueva instancia de `WorkerRunnable` que se ejecuta en un hilo distinto. Posteriormente, la clase `BluetoothServer` vuelve a escuchar si hay nuevas solicitudes de conexiones.

6.2.5 Clase TcpServerThread

Esta clase es similar a la clase `BluetoothServerThread`, pero escucha un puerto TCP/IP. Java no puede obtener la dirección MAC de adaptadores Wi-Fi, por lo que cada dispositivo debe preguntar la dirección MAC al usuario la primera vez que utilice la conexión Wi-Fi; luego la almacena en memoria secundaria para usarla posteriormente. Al iniciar una conexión TCP/IP por *sockets*, lo primero que hace esta clase es leer seis bytes que corresponden a la dirección MAC del dispositivo que se conectó. Posteriormente, usa la dirección IP en vez del nombre amigable de Bluetooth para completar los datos que debe pasar a la nueva instancia de `WorkerRunnable`.

6.2.6 Clase EventInterpreter

La función de esta clase es coordinar los demás objetos de la capa de red *ad-hoc* para que ejecuten las acciones pertinentes en cada evento iniciado por la red. La conexión y desconexión forman parte de estos eventos. En los procedimientos colectivos de conexión y desconexión, esta clase debe notificar los cambios a la clase `Topology` de los cambios y debe utilizar las primitivas de la clase `RoutingManager` para reenviar el evento al resto de los nodos que aún forman la red. Cuando el evento de conexión o desconexión es de un vecino, se inicia produce una primitiva *Broadcast*. Por el contrario, cuando el evento no proviene de un nodo vecino, este evento llega como parte de una primitiva *Broadcast* iniciado en otro nodo y debe ser reenviado a los demás nodos produciendo una primitiva *Forward*. Estas primitivas se explican en la Sección 6.3.3.

6.2.7 Clase RoutingManager

Método	Entrada	Salida
`terminate`	`void`	`void`
`forward`	Un arreglo de bytes y el identificador del vecino del que proviene el mensaje	`void`
`send`	Un arreglo de bytes y el identificador del vecino destinatario del mensaje	`void`
`broadCast`	Un arreglo de bytes	`void`

Tabla 6.2: Métodos de la clase `RoutingManager`

La clase `RoutingManager` proporciona las primitivas *Broadcast*, *Forward* y *Send*. Está construida para mandar únicamente bytes, ignorando su significado. Por esta razón, no participa directamente en los procedimientos colectivos. Sin embargo, en el procedimiento de desconexión individual, esta clase debe finalizar de forma adecuada mediante el método `terminate`, mostrado en la Tabla 6.2. Este método indica al hilo

de esta clase que debe vaciar la cola de mensajes para después terminar, con el fin de evitar la perdida de información que se ha dado por hecho que se había mandado.

6.3 Transmisión de datos

La principal importancia de la red es la transferencia de datos, lo cual es posible cuando se tienen las rutas actualizadas. El envío y la recepción de datos comprenden tres aspectos: 1) recibir y procesar datos que llegan desde las diferentes conexiones abiertas, 2) enviar por estas conexiones los datos generados localmente y 3) hacer la función de *router* en los nodos que estén funcionando como puente entre varias secciones de la red.

6.3.1 Clase EventInterpreter

Esta clase despacha los mensajes que llegan por la red a las clases que los necesitan. Un solo mensaje puede ser usado por más de una clase, e.g., una operación es usada por `SharedObject` pero también por `RoutingManager` puesto que debe ser retransmitida a los demás nodos. La clase `EventInterpreter` decide qué mensajes deben ser retransmitidos y cuáles no, pero no realiza la retransmisión misma, sino que notifica a `RoutingManager`, que es la clase encargada de efectuar la retransmisión.

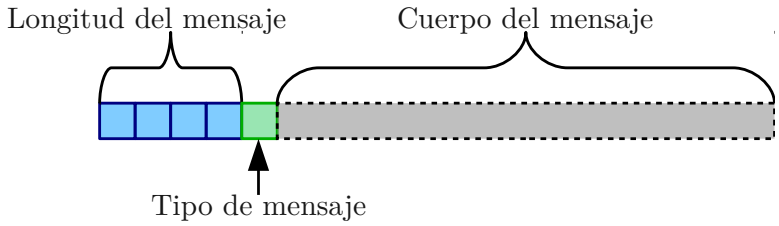


Figura 6.3: Formato de trama

Método	Entrada	Salida
`interpret`	arreglo de bytes a interpretar e identificador del hilo que los recibió	bandera `continuar` que indica al hilo si debe continuar

Tabla 6.3: Métodos de la clase `EventInterpreter`

Los mensajes son recibidos como arreglos de bytes en el método `interpret` mostrado en la tabla 6.3. Este método es llamado por `WorkerRunnable`. Cada hilo tiene un identificador que corresponde al registro de usuario en `UserData` y a la instancia `NeighborPeer` en `Topology`. El identificador del hilo sirve para saber desde qué nodo llegan los datos. Como se muestra en la figura 6.3, el quinto byte del mensaje

indica el tipo de mensaje, mediante el cual la clase `EventInterpreter` determina el procedimiento que se debe ejecutar. Existen los siguientes tipos de mensajes:

- **Eventos de conexión y desconexión**. Estos mensajes son necesarios para el ruteo y deben ser notificados a la clase `Topology`. Cuando se generan localmente se refieren a un nodo vecino, por lo tanto deben ser difundidos por toda la red. Cuando se generan remotamente se refieren a un nodo no vecino, así que se deben reenviar.

- **Eventos de operaciones**. Se refieren a las operaciones de los objetos compartidos. Cuando los eventos de operaciones son recibidos, estos eventos deben ser entregadas al objeto compartido local y reenviados, pues todos los nodos necesitan conocerlos. Cuando se generan localmente, se deben difundir por toda la red.

- **Eventos genéricos**. Son eventos producidos explícitamente por la aplicación. Al llegar desde la red, estos eventos pueden: 1) ser dirigidos al nodo local, 2) usar al nodo local como paso o 3) las dos anteriores.

6.3.2 Clase WorkerRunnable

La función de esta clase es organizar los bytes que recibe en una conexión y formar tramas, las cuales pueden llegar fraccionadas. Esta clase guarda los fragmentos hasta que se tenga la trama completa, entonces la pasa al objeto de la clase `EventInterpreter` para procesar la trama completa. La trama tiene el formato mostrado en la figura 6.3. Los primeros cuatro bytes indican el tamaño completo de la trama con el tipo de dato entero de Java ME y el resto de la trama incluye un byte que indica el tipo de mensajes y opcionalmente un cuerpo que varía según el tipo de mensaje.

6.3.3 Clase RoutingManager

Esta clase recibe instrucciones de ruteo proporcionadas por la clase `EventInterpreter` o por acciones locales. Las instrucciones se dan por medio de tres primitivas: *Broadcast*, *Send* y *Forward*.

Las llamadas a las primitivas reciben un arreglo de bytes, los cuales serán enviados a uno o más nodos dependiendo de la primitiva. Internamente, cada método encola la acción que debe desempeñar la red. Esta acción se decide usando la información de `Topology` que también mantiene la referencia a los flujos de salida. Un hilo dentro de `RoutingManager` recibe notificaciones cuando la cola deja de estar vacía. El hilo revisa la cola al despertar y toma un flujo de salida y un arreglo de bytes (el mensaje ya listo para su envío por una conexión dada); después escribe el arreglo en el flujo; estas acciones se repiten hasta que la cola esté vacía y entonces el hilo se vuelve a dormir.

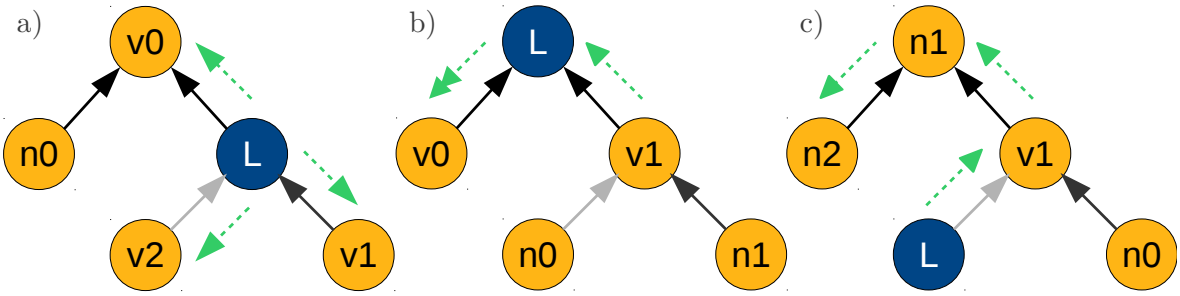


Figura 6.4: Primitivas de RoutingManager

La primitiva *Broadcast* tiene la función de propagar un mensaje en toda la red. Esta primitiva se llama en el nodo que desea difundir el mensaje por medio del método `broadCast` mostrado en la Tabla 6.2. Dicho nodo transmite el mensaje a todos sus vecinos y supone que estos se encargarán de que el mensaje llegue también a los nodos que no son vecinos del nodo original. Esta primitiva inicia una ejecución distribuida del algoritmo *blind flooding* [Garg, 2002]. La principal función de esta primitiva es difundir mensajes que contienen eventos de operaciones. La figura 6.4-a muestra un ejemplo en el que un nodo llamado L usa la primitiva *Broadcast* y envía a todos sus vecinos v un mensaje simbolizado por flechas punteadas.

La primitiva *Forward* supone que un mensaje llega desde un vecino y debe ser retransmitido. Como el mensaje viene desde un vecino dado, este vecino ya conoce dicho mensaje y no se le debe regresar, pero al resto de los vecinos si se les debe enviar. Esta primitiva es llamada por medio del método `forward` mostrado en la Tabla 6.2. El parámetro identificador en este método pertenece al vecino desde el que llegó el mensaje, i.e., al vecino al que no se le enviará el mensaje. Esta primitiva se utiliza en los eventos locales de conexión, puesto que si un vecino se conecta no tiene caso notificarle que se ha conectado. En la figura 6.4-b se muestra un ejemplo en el que el nodo L recibe un mensaje desde su vecino $v1$, denotado con una flecha punteada; después el mensaje es reenviado al resto de los vecinos ($v0$). El mensaje reenviado se denota por una doble flecha punteada en la figura 6.4-b.

La primitiva *Send* sirve para enviar un mensaje a un único nodo específico y es usada por medio del método `send` mostrado en la Tabla 6.2, en el que el parámetro identificador corresponde al nodo destino. Este método se tendrá que llamar en todos los nodos que formen parte del camino entre el nodo origen y el nodo destino, pues al llegar a un nodo que no es el destino, este debe reenviar el mensaje para que continúe su camino. Cada nodo sabe cuál es el siguiente nodo en el camino gracias a la información almacenada en `Topology`. En la figura 6.4-c se muestra un ejemplo de la primitiva *Send*. El nodo nombrado L manda un mensaje al nodo $n2$. Ya que L no es vecino de $n2$, L deberá mandar el mensaje por medio de un vecino, en este caso $v1$. Las flechas punteadas describen la ruta que seguirá el mensaje.

La información almacenada en `Topology`, que es consultada por las primitivas antes descritas, comprende la lista de los nodos vecinos y las listas de los nodos que

pueden ser alcanzados desde cada vecino. Para ejecutar la primitiva *Broadcast* solo se necesita saber cuales son los vecinos. Para ejecutar la primitiva *Forward* se necesita la misma información, además del nodo proporcionado como parámetro. Para ejecutar la primitiva *Send* se necesita buscar en todas las listas al nodo destino, el cual puede ser un vecino o uno de los nodos en las listas que tiene cada vecino. Cuando el destino es un vecino simplemente se le manda el mensaje; cuando el destino se encuentra en una de las listas de un vecino, se manda el mensaje a ese vecino, pero este tendrá que continuar ejecutando *Send*. La búsqueda dentro de las listas es efectuada por la clase `Topology`.

Los métodos de cada primitiva no realizan el envío por la red, solo encolan estos envíos para que un hilo diferente los ejecute. Por cada nodo al que se le enviarán bytes se encola el flujo de salida y los bytes que se enviarán. Los flujo se obtienen de `Topology` siguiendo el criterio mencionado en la descripción de cada primitiva. El hilo que revisa la cola solo encuentra, por cada registro desencolado, un flujo de salida en el cual escribir así como los bytes a escribir.

6.4 Manejo de mensajes recibidos

Los mensajes recibidos son procesados por la clase `EventInterpreter` de acuerdo al tipo de mensaje. Los mensajes de tipo evento genérico y evento de operación que son destinados al nodo local se deben pasar a las capas superiores, pero antes deben ser convertidos en objetos en vez de pasarse como bytes. La clase `Marshaling` se encarga de esta conversión.

6.4.1 Clase EventInterpreter

Esta clase concentra los procedimientos a ejecutar en todos los tipos de mensajes. Las operaciones y los mensajes son pasados a la clase `Marshaling` para obtener los objetos, los cuales posteriormente son pasados al método necesario, e.g., `enqueuOperation` en la clase `Cot`.

6.4.2 Clase Marshaling

Esta clase recibe un arreglo de bytes y debe regresar instancias de la clase `Operation`. `Marshaling` no debe tener en su código cada subclase de `Operation` porque se pueden agregar nuevas operaciones en objetos compartidos hechos a la medida. Por lo tanto, no es deseable tener que modificar esta clase. Para que `Marshaling` pueda delegar la instanciación de objetos `Operation`, las subclases de `Operation` deben contar con una interfaz común para que todas se puedan construir desde bytes. Esta interfaz está en la clase padre `Operation`. Los detalles específicos de cada operación son tratados por sus respectivas clases `Parameter`. Todas las subclases de `Parameter` deben poder crear nuevas instancias a partir de bytes, por lo que las nuevas clases

de operaciones que definan nuevos tipos de parámetros necesitan que estos parámetros implementen un método para crear instancias a partir de bytes. Las subclases de `Operation` y `Parameter` implementan un método que sigue el patrón de diseño *Factory*, el cual consiste en obtener nuevas instancias a partir de un método en vez de hacerlo desde el constructor. Este patrón facilita la instanciación y se utiliza para que `Marshaling` no requiera tener en su código todos los constructores necesarios. Los métodos de *Factory* son:

- `factory( idUser : int, vec : ContextVector, params : Parameter[] ) : Operation` en la clase `Operation`.
- `factory( name : String, input : DataInputStream) : Parameter` en la clase `Parameter`.

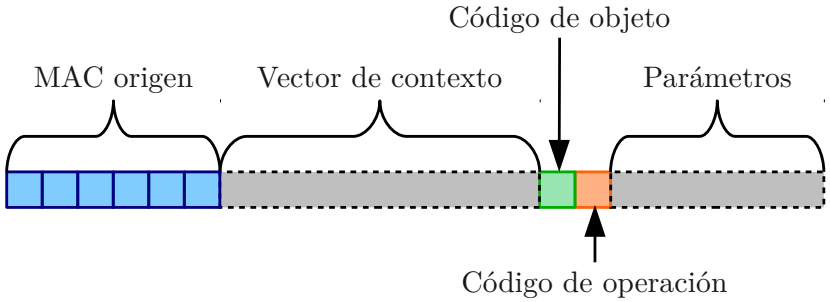


Figura 6.5: Formato de operación

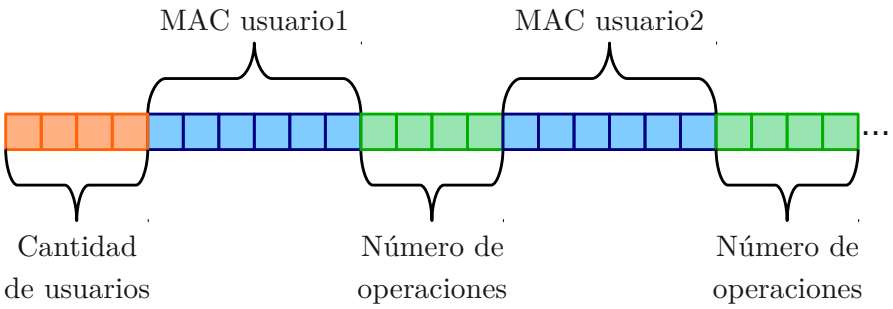


Figura 6.6: Formato de vector de contexto

En la figura 6.5 se muestra el cuerpo de una trama que representa una operación. El procedimiento para crear un nuevo objeto `Operation` es el siguiente:

1. Se lee la dirección MAC del nodo origen (6 bytes); mediante esta dirección se obtiene de `UsersData` el id del usuario correspondiente. Si no se conocía el usuario, éste es registrado junto con su dirección MAC al solicitar el id y como resultado `UsersData` regresa el nuevo id.

2. Se lee el vector de contexto, el cual es transmitido como se muestra en la figura 6.6: primero la longitud del vector y luego tuplas ⟨dirección MAC, número de operaciones⟩. A partir de estas direcciones MAC se obtienen identificadores locales por medio de `UsersData`.

3. Se lee el código del objeto compartido. Al inicializar las clases de la biblioteca CDCDM se asignan códigos a los objetos compartidos que se crean. Estos códigos se determinan en tiempo de compilación y son iguales en todos los nodos.

4. Se lee el código de la operación. Cada objeto compartido tiene una lista de las clases de sus operaciones; de esta lista se obtiene un objeto de la clase `Class`, el cual permite crear una instancia de la clase que representa, en este caso una operación. Sin embargo esta instancia no será usada como operación, sino que servirá para llamar al método `factory`. El objeto regresado por este método será la nueva instancia de la operación.

5. Se obtienen las clases de los parámetros desde la instancia temporal de la operación. Cada operación tiene una lista de las clases de sus parámetros. Similarmente al paso anterior, se obtienen instancias temporales de los parámetros en las que se llaman métodos *factory*. Internamente, en los métodos *factory*, se lee el resto de los bytes y se regresan las instancias de parámetros que serán usadas.

6. La nueva operación es instanciada pues ya se tiene todo lo necesario, i.e., el identificador del usuario origen, el vector de contexto con el orden local de los usuarios y las instancias de los parámetros de la operación.

Capítulo 7

Aplicación de prueba

Este capítulo describe una aplicación desarrollada para probar la biblioteca CDCDM, lo que se busca probar es la capacidad de facilitar el desarrollo de aplicaciones y el desempeño de la biblioteca CDCDM durante la ejecución de dichas aplicaciones. En la Sección 7.1 se presenta los requerimientos de la aplicación de prueba y en la Sección 7.2 se describe dicha aplicación. En la Sección 7.3 se detalla cómo ayudó la biblioteca CDCDM al desarrollo y finalmente en la Sección 7.4 se explican las limitaciones encontradas en la aplicación.

7.1 Requerimientos de una aplicación móvil colaborativa

Los grupos de usuarios móviles tienen el objetivo de trabajar de manera conjunta y coordinada, esta dinámica de trabajo en ocasiones se inicia de manera espontánea.

Para lograr este objetivo se propuso una aplicación colaborativa de edición de texto simple. Por medio de esta aplicación los usuarios podrán participar en la edición y visualizar las acciones de edición de los demás usuarios.

La aplicación propuesta debe cumplir los siguientes requerimientos:

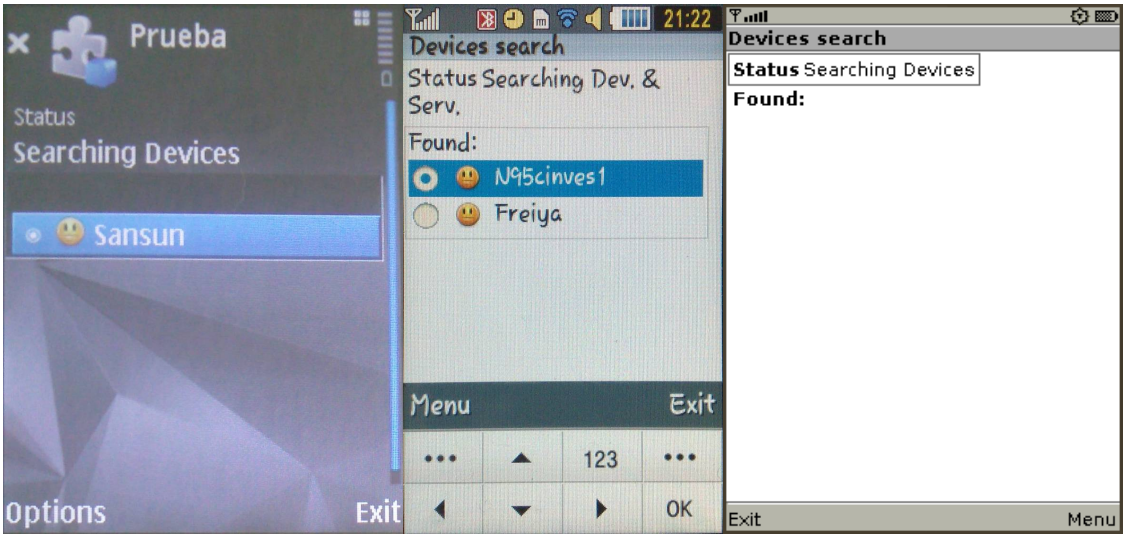
1. Poder producir documentos sencillos, e.g., borradores.
2. La interacción debe ser fluida y sin interrupciones por parte de la aplicación.
3. Los usuarios deben poder editar el mismo documento concurrentemente.
4. La versión del documento que tenga cada usuario debe ser coherente con las réplicas de los demás usuarios.
5. La interacción se debe poder iniciar espontáneamente.

7.2 Descripción de la aplicación de prueba

La aplicación Editor de Texto Móvil Colaborativo (ETMC) permite editar texto simple en dispositivos móviles que cuenten con Java ME. La biblioteca CDCDM provee funcionalidades de coherencia de datos, conciencia de grupo y comunicación de red. La funcionalidad que implementa la aplicación ETMC consta solamente de los menús necesarios para iniciar una conexión y de la forma utilizada para desplegar el texto que está siendo editado y la información relativa a los usuarios conectados. Dicha funcionalidad se describe con mayor detalle en las secciones 7.2.1 y 7.2.2 respectivamente.

7.2.1 Establecimiento de conexiones

Los dos tipos de conexiones que puede establecer la aplicación ETMC son Wi-Fi y Bluetooth. La espera de conexiones entrantes es implementada por la biblioteca CDCDM, como se explicó en la Sección 6.2.



(a) Nokia N95 (b) Samsung GT-S5560 (c) Emulador WTK 2.5.2

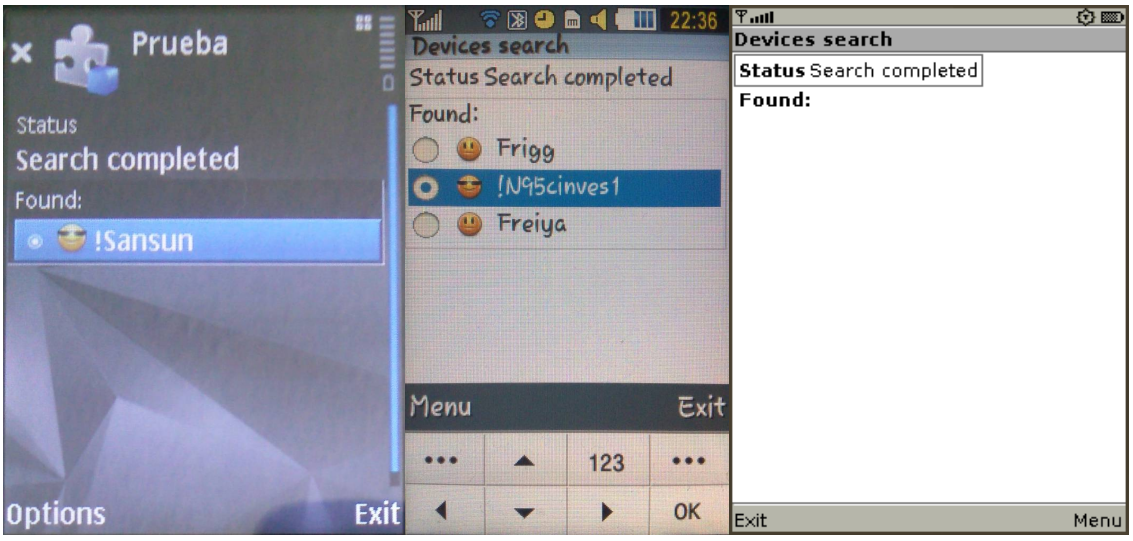
Figura 7.1: Búsqueda de dispositivos en curso

Para establecer conexiones por Bluetooth primero se requiere hacer un procedimiento de descubrimiento, el cual consta de dos pasos: 1) descubrir dispositivos y 2) descubrir un servicio en los dispositivos ya descubiertos. En la figura 7.1, se muestra la forma de búsqueda cuando el descubrimiento está en curso. El descubrimiento está implementado de forma tal que los dos pasos anteriores se pueden realizar concurrentemente, i.e., una vez que se descubre un dispositivo se puede iniciar la búsqueda de servicios en este dispositivo, aún cuando la búsqueda de dispositivos continúa o mientras que se busca servicios en otros dispositivos. Aunque la aplicación tiene la

capacidad de descubrimiento concurrente, no todos los dispositivos móviles pueden usar esta capacidad. Algunos dispositivos solo permiten hacer una búsqueda de servicios a la vez, mientras que otros dispositivos no inician la búsqueda de servicios hasta que han concluido la búsqueda de dispositivos.

Las diferentes capacidades de búsqueda concurrente se pueden apreciar en los dos dispositivos con los que se probó la aplicación. Estos dispositivos son los teléfonos Nokia N95 (figura 7.1a) y Samsung GT-S5560 (figura 7.1b). En la figura 7.1c se muestra el emulador del ambiente de desarrollo de Java ME, este emulador no tiene funcionalidad Bluetooth, como consecuencia no despliega resultados en las búsquedas. Ya que no es relevante cuando se discuten las conexiones Bluetooth no será mencionado.

Los dispositivos Samsung y Nokia pueden hacer solo una búsqueda de servicios a la vez. Esta capacidad se puede consultar por medio del método `getProperty` de la clase `LocalDevice`, usando la propiedad `bluetooth.sd.trans.max`. La diferencia entre estos dos dispositivos es que el teléfono Nokia no busca servicios hasta que ha terminado de buscar dispositivos, mientras que el teléfono Samsung si puede buscar dispositivos y servicios simultáneamente (figura 7.1b). Adicionalmente, se notó que el teléfono Nokia entraba en un *deadlock* cuando se intentaba iniciar una búsqueda de servicios desde el método que es llamado para notificar el descubrimiento de un dispositivo. Este problema se corrigió creando hilos distintos para las búsquedas de servicios y de dispositivos. Sin embargo, en el teléfono Nokia, la ejecución de la búsqueda de servicios no comienza antes del término de la búsqueda de dispositivos.



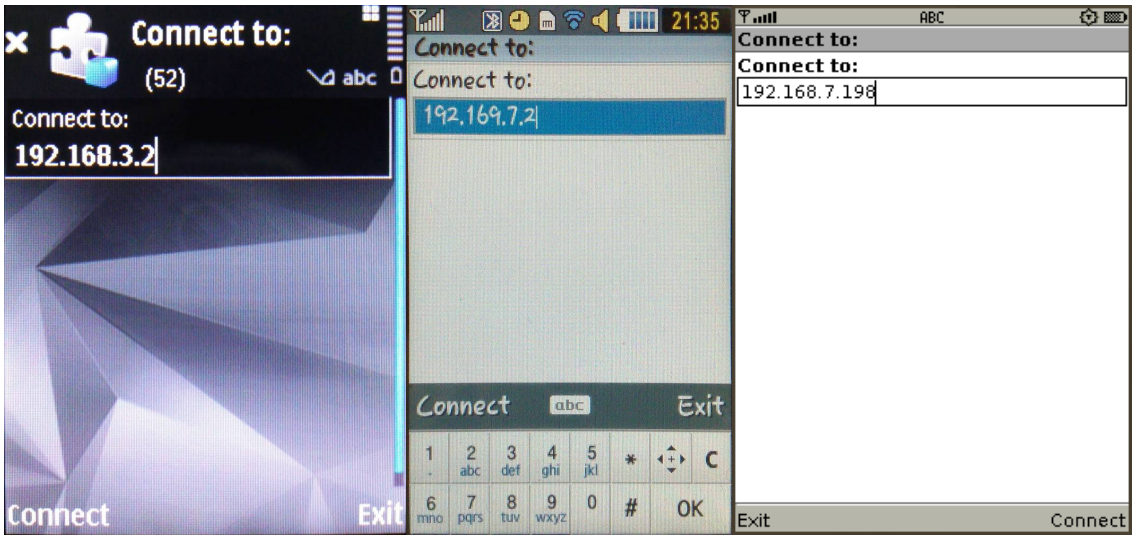
(a) Nokia N95 (b) Samsung GT-S5560 (c) Emulador WTK 2.5.2

Figura 7.2: Búsqueda de dispositivos concluida

En la figura 7.2 se muestra el resultado del procedimiento de descubrimiento. Los dispositivos encontrados se despliegan en una lista. Los dispositivos que cuentan con

el servicio buscado se muestran con un icono diferente. En la figura 7.2a se muestra la lista de dispositivos encontrados por el dispositivo Nokia, en este caso solo se encontró uno. En la lista junto al icono se muestra el nombre amigable Bluetooth del dispositivo encontrado y un carácter '!' concatenado al dicho nombre para indicar que se encontró el servicio buscado, el icono que se muestra es el que corresponde a los dispositivos que cuentan con el servicio buscado. En la figura 7.2b se muestra la lista de dispositivos encontrados por el dispositivo Samsumg, el cual encontró 3 dispositivos, uno de ellos con el servicio buscado, dicho dispositivo es el que se muestra seleccionado, como se puede ver el icono es diferente, el icono de los otros dos dispositivos denota que no tienen el servicio buscado.

La opción de conectarse aparece después de que se ha encontrado por lo menos un dispositivo con el servicio buscado. Una vez finalizada la búsqueda, se presenta la opción de buscar nuevamente.



(a) Nokia N95 (b) Samsung GT-S5560 (c) Emulador WTK 2.5.2

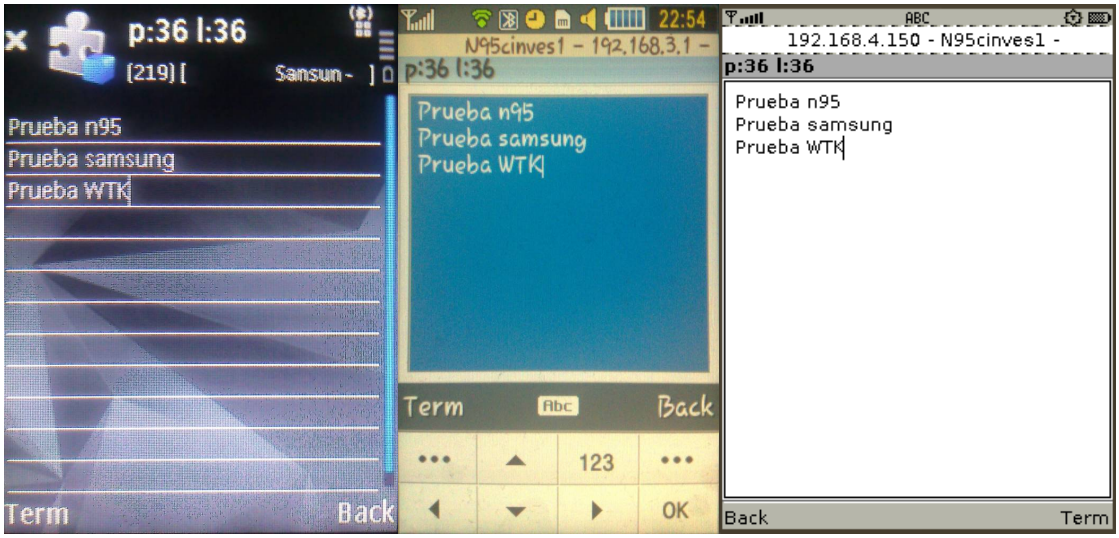
Figura 7.3: Conexión por TCP/IP

Las figura 7.3a, 7.3b y 7.3c muestran la forma de conexión por TCP/IP en los tres dispositivos utilizados. En esta forma se debe escribir la dirección IP del dispositivo al que se desea conectar. Para iniciar una conexión tipo *socket* se necesita estar previamente conectado a la red Wi-Fi, ya que algunas implementaciones de Java ME no preguntan al usuario que red Wi-Fi desea usar. Específicamente, el teléfono Nokia solicita al usuario seleccionar un punto de acceso si no estaba previamente conectado. Por su parte, el teléfono Samsung intentará conectarse al punto de acceso por omisión y producirá un error si no encuentra dicho punto de acceso.

Las formas anteriormente descritas son ejemplos que pueden ser reutilizadas en otras aplicaciones, sin necesidad de hacer modificaciones. La forma de búsquedas Bluetooth (figuras 7.1 y 7.2) dependen de los estándares MIDP 1.0 y JSR-82 en tanto

que la forma de TCP/IP (figura 7.3) requiere del estándar MIDP 2.0 (JSR-118).

7.2.2 Forma de edición de texto



(a) Nokia N95 (b) Samsung GT-S5560 (c) Emulador WTK 2.5.2

Figura 7.4: Forma de edición de texto

Después de que se establece exitosamente la conexión, la aplicación ETMC muestra la forma de edición de texto, mostrada en la figura 7.4. Esta forma contiene un objeto de la clase `TextBox` que forma parte de MIDP 1.0 y que no se requiere implementarla. También, se tiene un objeto de la clase `Ticker` en la forma de edición de texto. Esta clase proporciona una cinta de texto animada que se desplaza continuamente. El objeto `Ticker` muestra la lista de usuarios actualmente conectados. Esta lista es actualizada cuando ocurre un evento de conexión o desconexión. Dicha actualización se hace por medio de un objeto de la clase `ConcreteObserver`, como se explicó en la Sección 5.5.

El contenido del objeto `TextBox` es modificado por la interfaz de usuario y también por la biblioteca CDCDM. La API de MIDP no proporciona capacidades para recibir notificaciones de cambios en el texto de `TextBox`, por lo que la biblioteca CDCDM debe detectar estos cambios. La detección de cambios se explicará en la Sección 7.3.2.

7.3 Ventajas de la biblioteca CDCDM

Una aplicación que cuente con la misma funcionalidad que la aplicación ETMC normalmente tendría que implementar funcionalidad para: 1) servir las conexiones

entrantes, 2) rutear datos en la red *ad-hoc* entre Wi-Fi y Bluetooth, 3) mantener coherencia de datos, 4) notificar conexiones y desconexiones y 5) opcionalmente realizar descubrimiento por Bluetooth.

Usando la biblioteca CDCDM, la aplicación ETMC solo necesita implementar la siguiente funcionalidad: iniciar conexiones como cliente, registrar *observers* y construir la forma de edición de texto.

7.3.1 Interfaz con la biblioteca CDCDM

A continuación, se explica la interfaz usada entre la aplicación ETMC y la biblioteca CDCDM:

1. **Inicialización**. Antes de poder usar la biblioteca CDCDM, es necesaria la inicialización, la cual depende de qué objetos compartidos serán usados. La inicialización consiste de instanciar un objeto de la clase principal de la biblioteca CDCDM, que es `UsersData`. Posteriormente, se registra el objeto compartido (`SharedTextBuffer` en este caso) en la instancia de `UsersData`. Finalmente, se instancia un objeto de la clase `Hub`.

2. **Servir conexiones**. Antes de que se inicien las conexiones, los hilos que escucharán conexiones entrantes se deben instanciar e iniciar. Estos hilos pueden ser de la clase `BluetoothServerThread` o `TcpServerThread`. Se podría querer instanciar uno solo si el dispositivo que ejecute la aplicación no soporta a ambos.

3. **Iniciar conexiones**. Las formas descritas en la Sección 7.2.1 solo necesitan saber el identificador del servicio para Bluetooth o el número del *socket* para Wi-Fi. Los clientes obtendrán de estas formas una conexión abierta a la que registrarán en la instancia de `Hub`. En el otro lado de la conexión, el registro es efectuado automáticamente por los hilos antes mencionados.

4. **Registrar *observers***. Los *observers* permiten a la aplicación desplegar los cambios en el modelo de datos y en la lista de usuarios conectados. La aplicación debe registrar objetos de la clase `Observer` para recibir las notificaciones pertinentes. En este caso, la aplicación ETMC cuenta con un modelo de datos de *buffer* de texto simple que utiliza la clase `TextBufferObserver`. Para detectar los cambios de usuarios, la aplicación ETMC utiliza un objeto `ConcreteObserver`, el cual consulta la lista de usuarios y actualiza el objeto de la clase `Ticker` de la figura 7.4 cada vez que este recibe una notificación.

7.3.2 Detección de cambios locales

La interfaz de MIDP no notifica a las aplicaciones sobre cambios en un campo de texto. Normalmente el usuario inicia una transacción por medio de un comando, notificando él mismo a la aplicación cuando ha terminado de editar el campo. Ya

que es necesario detectar cambios activamente (*client pulling*), la biblioteca CDCDM provee esta funcionalidad.

Los cambios en el campo de texto se detectan por medio del algoritmo Hunt-McIlroy [Hunt and McIlroy, 1976], el cual es utilizado por la aplicación *diff*, usada comúnmente para detectar cambios en repositorios de código fuente. Este algoritmo se basa en el problema de LCS (*Longest Common Subsequence*) y provee como salida una lista de cambios entre dos cadenas. Estos cambios están expresados como operaciones de inserción y supresión.

La frecuencia con la que se detecten cambios no puede ser muy alta pues esto requeriría de muchos recursos de memoria y procesamiento. Además, la velocidad de escritura de un usuario normal no es muy alta por lo que no se requiere detectar cambios muy frecuentemente. Gracias a que se utiliza Transformación Operacional (OT), se toleran retrasos de duración arbitraria en el envío de las operaciones, pero esto solo es válido desde la perspectiva de la coherencia de datos. Sin embargo, desde la perspectiva de la conciencia de grupo, el retraso no debe ser perceptible para los usuarios. A partir de estas restricciones, se acota el intervalo de detección a pocos segundos, buscando no desperdiciar procesamiento ni memoria sin perjudicar la fluidez de la aplicación.

7.4 Limitaciones de la aplicación de prueba

Durante la fase de pruebas, se detectaron algunas limitaciones en los resultados obtenidos de la aplicación ETMC. Estas limitaciones son en parte consecuencia de que los componentes provistos por la API de MIDP no están dirigidos a proporcionar funcionalidad complicada, sino general y multiplataforma. Otras limitaciones se deben a que las implementaciones de diferentes fabricantes varían entre sí.

Las limitaciones observadas se describen a continuación:

- **Telepunteros**. Originalmente se planeó programar telepunteros que mostraran la posición de otros usuarios en el texto. Sin embargo, no es posible dibujarlos en un componente de tipo `TextBox`.

- **Conflicto con autocompletado**. El conflicto con el autocompletado se presenta en el teléfono Nokia. Cuando el usuario está introduciendo una palabra pero aun no ha terminado de introducirla y Java ME consulta la cadena en la forma de edición de texto. En esta situación, Java ME registra la parte de la palabra que ya ha sido escrita pero el procedimiento de autocompletado es interrumpido y el usuario no recibe más sugerencias de la palabra que estaba escribiendo, en vez el teléfono comienza a sugerirle una palabra nueva, esta palabra comenzando a partir de lo que el usuario escriba posteriormente a la consulta que hizo Java ME. Como consecuencia la parte que se había escrito podría no ser correcta al ser una sugerencia parcial y las sugerencias posteriores serán incorrectas al desconocer el principio de la palabra.

Además, cuando la aplicación inserta texto a la izquierda de la palabra antes que esta sea completada, el texto se desplaza acorde a la inserción, pero el cursor permanece en la misma posición.

- **Conflicto con *soft-keyboard***. Este conflicto se presenta con el teléfono Samsung, cuando se utiliza el teclado presentado por la pantalla táctil. Este teclado se despliega en otra vista que no es percibida por Java ME. El texto escrito en la vista del teclado táctil no es actualizado cuando Java ME efectúa cambios sobre `TextBox`, por lo que al terminar la edición en la vista del teclado táctil se sobrescribe todo el texto que había en `TextBox`, ignorando los cambios que podrían haber llegado desde otros dispositivos.

Para enfrentar estas limitaciones se propone desarrollar un componente especializado en la interfaz de usuario que detecte los eventos resultantes de la presión de botones y que pueda dibujar elementos arbitrarios. Esto es posible por medio de la clase `GameCanvas`, de la interfaz MIDP 2.0. La clase `GameCanvas` notifica a la aplicación los eventos de presión de botones y permite dibujar figuras en su superficie, e.g., telepunteros. Sin embargo, este componente no solo sería complicado de desarrollar sino también está fuera del alcance de la presente al formar parte de la interfaz gráfica tesis. Esta es una solución compromiso, puesto que a cambio de la funcionalidad extra provista por la clase `GameCanvas` se requiere que el dispositivo implemente un estándar más avanzado, i.e., MIDP 2.0 (JSR-118). Este estándar es el mismo que se requiere para hacer conexiones tipo `Socket`.

Capítulo 8

Conclusiones y trabajo futuro

En este capítulo se concentran las conclusiones obtenidas a lo largo del desarrollo de la presente tesis. Así mismo, se mencionan las principales contribuciones y algunas ideas para continuar el trabajo desarrollado hasta ahora. En la Sección 8.1 se hace una recapitulación del problema tratado y del trabajo realizado. En la Sección 8.2 se exponen las contribuciones y las conclusiones alcanzadas. Finalmente, en la Sección 8.3 se plantea algunas propuestas para continuar el trabajo desarrollado.

8.1 Recapitulación del trabajo realizado

En esta tesis se enfrentó el problema de desarrollar sistemas de edición colaborativa que facilitan la interacción cara a cara y que se ejecutan en dispositivos móviles. Estos sistemas privilegían a los espacios de producción, por lo que se desarrolló el soporte para la coherencia de los datos compartidos y parcialmente para la conciencia de grupo. Se dejaron fuera del alcance de esta tesis otras funcionalidades como la interfaz gráfica de usuario, la coordinación de colaboradores y la comunicación directa de colaboradores.

La propuesta para solucionar este problema fue desarrollar una biblioteca con orientación de *toolkit*, llamada biblioteca para el soporte de Coherencia de Datos Compartidos en Dispositivos Móviles (CDCDM). Dicha biblioteca tiene la funcionalidad de mantener la coherencia de datos compartidos. Para lograr este fin se investigaron las técnicas disponibles y se seleccionó la Transformación Operacional (OT) pues ésta enfrenta efectivamente un problema importante de los dispositivos móviles, referente a las desconexiones frecuentes e inesperadas. Los sistemas móviles colaborativos que facilitan la interacción cara a cara siempre necesitan de una red de conexión inalámbrica debido a la naturaleza móvil de los dispositivos. Este tipo red tiene limitaciones que ocasionan desconexiones frecuentes, por lo tanto, el mecanismo de coherencia debe lidiar con tales limitaciones.

La técnica Transformación Operacional fue propuesta hace más de 20 años y se han desarrollado varios algoritmos durante ese tiempo. De los algoritmos investigados, se seleccionó el algoritmo COT (Context-based Operational Transformation), el

cual cuenta además con una prueba formal. Otra ventaja de este algoritmo es que utiliza solo transformación de inclusión y no necesita transformación de exclusión, esta ventaja se explica en la siguiente sección.

Para solucionar problemas propios de los dispositivos móviles se consideraron las dimensiones de la movilidad, las cuales influyen en diferente grado en los sistemas móviles pero las que se identificaron como principales fueron: 1) fuente de poder limitada, 2) conectividad intermitente y 3) proliferación de plataformas. Con base en estas dimensiones se seleccionaron: 1) el tipo de red y 2) la plataforma de desarrollo.

También se tomó en cuenta una prestación importante de la red *ad-hoc*, i.e., la capacidad de trabajar en ambientes que carecen de infraestructura de red. Neyem señala a la prestación antes mencionada como necesaria para ciertos escenarios [Neyem et al., 2009]. La red *ad-hoc* también fue utilizada para implementar la colaboración espontánea, la cual fue propuesta (e implementada) por Roth [Roth, 2002].

La biblioteca propuesta fue diseñada para permitir la reutilización del código y fue probada por medio de una aplicación.

8.2 Conclusiones

La principal contribución de la presente tesis fue el software desarrollado para el tipo de interacción estudiado. Este software es reutilizable y podrá beneficiar investigaciones posteriores en esta área.

Otra contribución es que algunos de los problemas de implementación solucionados pueden ser tomados como ejemplo para desarrollos similares, e.g., el uso de los protocolos Bluetooth de forma tal que se tenga compatibilidad con distintos dispositivos. Estos ejemplos se pueden tomar tanto a nivel de código fuente como de diseño.

Una contribución secundaria y más abstracta es la arquitectura propuesta para este tipo específico de aplicaciones, ya que se basó en el análisis de los requerimientos particulares de los sistemas móviles.

Esta arquitectura fue propuesta para desarrollar la biblioteca CDCDM, la cual fue implementada y probada por medio de una aplicación de prueba.

Con la arquitectura propuesta se busca solucionar el problema de la inconsistencia de datos compartidos que surge en editores colaborativos, lo anterior enfocado a dispositivos móviles y a interacción cara a cara. Los resultados fueron exitosos puesto que la aplicación de prueba se comportó adecuadamente.

La interfaz gráfica de usuario está fuera del alcance de la biblioteca CDCDM, sin embargo, en la aplicación de prueba se pudo observar que los componentes provistos por MIDP son demasiado simples para implementar las funcionalidades de telepunteros, por lo tanto se detectó la necesidad de componentes en la interfaz gráfica que aporten información conciencia de grupo, ya sea que estos componentes se desarrollen como parte de las aplicaciones específicas o como parte de otra biblioteca.

Respecto a las técnicas para mantener la coherencia de datos se concluyó por medio de la implementación de la biblioteca CDCDM y las pruebas, que las capacidades de los dispositivos móviles actuales permiten utilizar la técnica OT sin retrasos

apreciables. Respecto a OT también se concluyó por medio del análisis de los algoritmos encontrados, que el algoritmo COT es el mejor para la reutilización de código, puesto que elimina la necesidad de la transformación por exclusión. La ventaja de eliminar esta transformación por exclusión se obtiene al programar nuevos objetos compartidos. En la arquitectura de la biblioteca CDCDM se planteó la posibilidad de reutilizar el algoritmo COT con nuevos objetos compartidos que siguen una interfaz definida. Al implementar estos nuevos objetos compartidos se deberá implementar las transformaciones y las operaciones, por lo que es deseable disminuir la codificación requerida. Por el lado de las transformaciones, al utilizar el algoritmo COT solo se necesita programar n^2 transformaciones de inclusión, mientras que al utilizar otros algoritmos de OT se requiere programar además n^2 transformaciones de exclusión. Por el lado de las operaciones, la transformación por exclusión requiere en algunos modelos de datos que las operaciones almacenen información cuando son transformadas [Sun and Ellis, 1998]. Utilizando el algorítmo COT también se evita tener que programar la funcionalidad para almacenar dicha información.

También se propuso una solución para el problema del tamaño variable de los vectores de contexto. Este problema se debe a que los algoritmos de OT suponen un tamaño fijo del vector de contexto pues el número de participantes no se considera variable. Sin embargo en esta implementación si es variable, ya que nuevos usuarios se pueden conectar. La solución utilizada consiste en una generalización y en separar las funcionalidades del vector de contexto y de OT. La generalización consiste en considerar que los vectores de contexto tienen un número de componentes igual al número de usuarios actualmente conocidos, aun cuando se almacene realmente menos componentes, los componentes que no se tienen realmente almacenados se consideran cero para poder desempeñar la funciones necesarias. Esta generalización es equivalente a suponer que los usuarios siempre formaron parte de la sesión colaborativa pero no habían participado.

Se utilizaron algunas ideas para enfrentar la heterogeneidad de los dispositivos móviles. Por un, lado la utilización de los estándares más bajos de Java ME (MIDP 1.0 y CLCD 1.0) complicó ligeramente el desarrollo, pero esta elección de estándares brinda la oportunidad de utilizar mayor variedad de dispositivos móviles. Por otro lado, se utilizó una red *ad-hoc* para comunicar entre sí dispositivos con adaptadores Wi-Fi y Bluetooth. De esta manera se soluciona parcialmente el problema de la heterogeneidad del hardware de red, bajo las suposiciones de que los dispositivos soportan conectividad local y hay algún dispositivo disponible para trabajar como puente.

Otro problema solucionado fue la falta de un mecanismo de serialización como el de Java SE (Standard Edition), el cual fue necesario para transmitir operaciones por la red *ad-hoc* de una forma que se pudieran interpretar igual en cada sitio. Este mecanismo se hizo por medio de métodos *factory* y listas que almacenan las clases usadas.

8.3 Ideas de trabajo futuro

Algunas mejoras posibles al diseño y a la de la biblioteca CDCDM son:

- **Soporte de conexiones HTTP**. Todos los teléfonos que cuentan con MIDP 1.0 soportan este tipo de conexión, sin embargo esta conexión no fue considerada porque presenta la mayor razón costo/beneficio de las conexiones analizadas. Este soporte requeriría desarrollar un servidor Web especial para la biblioteca CDCDM, pero el beneficio sería abarcar la totalidad de los dispositivos móviles con MIDP. La conexión HTTP funcionaría solo cuando se tenga acceso a la red celular de datos.

- **Soporte para descubrimiento en conexiones Wi-Fi y HTTP**. Las conexiones de este tipo no cuentan con mecanismos de descubrimiento, pero se podrían implementar por medio de un servidor central, en el caso de HTTP y en el caso de Wi-Fi por medio de multicasts UDP.

- **Portabilidad**. Java ME abarca una gran cantidad de sistemas operativos, e.g., RIM OS (dispositivos Blackberry), Windows Mobile, Symbian, etc. Sin embargo, algunos sistemas operativos no soportan Java ME, por lo que se podría desarrollar una implementación de la biblioteca CDCDM para otros sistemas operativos, un ejemplo prometedor es Android.

- **Más modelos de datos**. Actualmente solo se cuenta con el modelo de datos del *buffer* de texto simple y se pueden construir varias aplicaciones sobre este modelo. Sin embargo, algunas aplicaciones serían más convenientes de implementar sobre otros modelos de datos, e.g., listas [Sun and Ellis, 1998] o árboles [Davis et al., 2002].

- **Optimización OT**. Los datos almacenados por el estado del objeto compartido pueden crecer considerablemente cuando se tenga un gran número de usuarios o cuando se haya usado el sistema por mucho tiempo; en esta situación será necesario un esquema de recolección de basura.

- **Creación de objetos en tiempo de ejecución**. Actualmente los objetos compartidos se crean únicamente en tiempo de compilación, pero se podría presentar la necesidad de crear nuevos objetos en tiempo de ejecución, o borrar objetos antiguos para ahorrar espacio en memoria, lo cual requerirá de cambios a la biblioteca CDCDM.

Algunos posibles desarrollos externos a la biblioteca CDCDM son:

- **Más aplicaciones**. Con el fin de investigar y probar la colaboración móvil cara a cara, se pueden desarrollar más aplicaciones de forma fácil sobre los modelos de datos disponibles.

- **Interfaz gráfica de usuario**. Se detectó la carencia de componentes gráficos que soporten la colaboración; estos componentes se podrían desarrollar conforme se vayan necesitando o como parte de una nueva biblioteca.

Publicaciones

- Saucedo-Tejada, G. and Mendoza S. (2011). An Architecture for Supporting Face-to-Face Mobile Interaction. In *CCE 2011: 8th International Conference on Electrical Engineering, Computing Science and Automatic Control*, pages 792-797, Médida Yucatán, México. IEEE Computer Society.

Apéndice A

Vectores de contexto

Como se definió en la Sección 3.4 del Capítulo 3, el contexto de una operación es el estado del objeto compartido que se tenía cuando que se generó la operación. El estado del objeto compartido es el conjunto de operaciones que se han ejecutado sobre dicho objeto compartido hasta cierto momento. Por lo tanto, el contexto de una operación es el conjunto de las operaciones que preceden a dicha operación.

Desde luego, un contexto se puede representar como un conjunto, sin embargo existe una forma más eficiente de representarlo, la cual se deriva de que el estado del objeto compartido es un conjunto parcialmente ordenado, por lo tanto todos los contextos lo son también. Esta representación eficiente es el vector de contexto.

Un estado del objeto compartido y, por lo tanto, un contexto de operación, se pueden representar como un vector de enteros, el cual tiene un componente por cada usuario existente. En cada componente se almacena el número de operaciones que provienen del usuario correspondiente y que ya han sido ejecutadas localmente. De acuerdo a la condición CC1, descrita en la Sección 3.5 del Capítulo 3, las operaciones serán ejecutadas en orden, i.e., ninguna operación será ejecutada antes que alguna de sus predecesoras. De lo anterior, se puede concluir que si la operación n fue ejecutada localmente, entonces todas las operaciones $0...n-1$ ya habían sido ejecutadas. Por lo tanto, basta con un entero (n) para representar el conjunto de todas las operaciones provenientes de un usuario que han sido ejecutadas hasta el momento.

Estos vectores de contexto son una especialización de los relojes vectores de [Lamport, 1978] y tienen las mismas propiedades y operaciones. Particularmente, se usan las funciones de comparación, que servirán para comparar contextos de una forma más eficiente que si se compararan conjuntos.

En la Sección 5.3.3 también se describe una forma eficiente de realizar restas de conjuntos, suponiendo que ambos conjuntos son representables por medio de vectores de contexto y que el conjunto substraendo es menor o igual que el conjunto minuendo.

Bibliografía

[B'far, 2004] B'far, R. (2004). *Mobile Computing Principles: Designing and Developing Mobile Applications with UML and XML*. Cambridge University Press, New York, NY, USA.

[Bradley, 2002] Bradley, M. L. (2002). Transformation-Based Concurrency Control in Groupware Systems. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, 2002.

[Carstensen and Schmidt, 1999] Carstensen, P. H. and Schmidt, K. (1999). Computer supported cooperative work: New challenges to systems design. In *In K. Itoh (Ed.), Handbook of Human Factors*, pages 619–636.

[Davis et al., 2002] Davis, A. H., Sun, C., and Lu, J. (2002). Generalizing operational transformation to the standard general markup language. In *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, CSCW '02, pages 58–67.

[Dourish and Bellotti, 1992] Dourish, P. and Bellotti, V. (1992). Awareness and coordination in shared workspaces. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, CSCW '92, pages 107–114.

[Ellis and Gibbs, 1989] Ellis, C. A. and Gibbs, S. J. (1989). Concurrency control in groupware systems. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 399–407, New York, NY, USA. ACM.

[Ellis et al., 1991] Ellis, C. A., Gibbs, S. J., and Rein, G. (1991). Groupware: some issues and experiences. *Commun. ACM*, 34(1):39–58.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Garg, 2002] Garg, Ph.D., V. K. (2002). *Elements of distributed computing*. John Wiley & Sons, Inc., New York, NY, USA.

[Gordon, 1995] Gordon, V. C. (1995). A counterexample to the distributed operational transform and a corrected algorithm for point-to-point communication. Research CS-95-08, Dept. of Computer Science, University of Waterloo.

[Greenberg and Marwood, 1994] Greenberg, S. and Marwood, D. (1994). Real time groupware as a distributed system: concurrency control and its effect on the interface. In *CSCW '94: Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 207–217, New York, NY, USA. ACM.

[Grudin, 1994] Grudin, J. (1994). Computer-supported cooperative work: history and focus. *Computer*, 27(5):19 –26.

[Hendrie, 1998] Hendrie, C. (1998). Objects which break "a calculus for concurrent update". Research CS499, Dept. of Computer Science, University of Waterloo.

[Herskovic et al., 2009] Herskovic, V., Ochoa, S. F., and Pino, J. A. (2009). Modeling groupware for mobile collaborative work. In *Proceedings of the 2009 13th International Conference on Computer Supported Cooperative Work in Design*, pages 384–389, Washington, DC, USA. IEEE Computer Society.

[Hunt and McIlroy, 1976] Hunt, J. W. and McIlroy, M. D. (1976). An algorithm for differential file comparison. Technical Report CSTR 41, Bell Laboratories, Murray Hill, NJ.

[Knister and Prakash, 1993] Knister, M. J. and Prakash, A. (1993). Issues in the design of a toolkit for supporting multiple group editors. *Computing Systems*, 6(2):135–166.

[Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565.

[Li et al., 2000] Li, D., Zhou, L., and Muntz, R. R. (2000). A new paradigm of user intention preservation in realtime collaborative editing systems. In *ICPADS '00: Proceedings of the Seventh International Conference on Parallel and Distributed Systems*, page 401, Washington, DC, USA. IEEE Computer Society.

[Munson and Dewan, 1996] Munson, J. and Dewan, P. (1996). A concurrency control framework for collaborative systems. In *CSCW '96: Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 278–287, New York, NY, USA. ACM.

[Neyem et al., 2009] Neyem, A., Ochoa, S. F., Pino, J. A., and Franco, D. (2009). An architectural pattern for mobile groupware platforms. In *Proceedings of the Confederated International Workshops and Posters on On the Move to Meaningful Internet Systems: ADI, CAMS, EI2N, ISDE, IWSSA, MONET, OnToContent, ODIS, ORM, OTM Academy, SWWS, SEMELS, Beyond SAWSDL, and COMBEK 2009*, OTM '09, pages 401–410, Berlin, Heidelberg. Springer-Verlag.

[Nichols et al., 1995] Nichols, D. A., Curtis, P., Dixon, M., and Lamping, J. (1995). High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, UIST '95, pages 111–120, New York, NY, USA. ACM.

[Ressel et al., 1996] Ressel, M., Nitsche-Ruhland, D., and Gunzenhäuser, R. (1996). An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *CSCW '96: Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 288–297, New York, NY, USA. ACM.

[Rodríguez-Covili et al., 2011] Rodríguez-Covili, J., Ochoa, S. F., Pino, J. A., Herskovic, V., Favela, J., Mejía, D., and Morán, A. L. (2011). Towards a reference architecture for the design of mobile shared workspaces. *Future Gener. Comput. Syst.*, 27:109–118.

[Roth, 2000] Roth, J. (2000). 'dreamteam': A platform for synchronous collaborative applications. *AI & Society*, 14:98–119.

[Roth, 2002] Roth, J. (2002). Seven challenges for developers of mobile groupware. In *In: Workshop Mobile Ad Hoc Collaboration;, CHI 2002*.

[Roth, 2005] Roth, J. (2005). The resource framework for mobile applications. In Camp, O., Filipe, J., Hammoudi, S., and Piattini, M., editors, *Enterprise Information Systems V*, pages 300–307. Springer Netherlands.

[Singh, 1989] Singh, B. (1989). Invited talk on coordinaion systems. In *Organizational Computing Conference*, pages 13–14.

[Suleiman et al., 1998] Suleiman, M., Cart, M., and Ferrié, J. (1998). Concurrent operations in a distributed and mobile collaborative environment. In *Proceedings of the Fourteenth International Conference on Data Engineering*, ICDE '98, pages 36–45, Washington, DC, USA. IEEE Computer Society.

[Sun and Ellis, 1998] Sun, C. and Ellis, C. (1998). Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, CSCW '98, pages 59–68, New York, NY, USA. ACM.

[Sun et al., 1998] Sun, C., Jia, X., Zhang, Y., Yang, Y., and Chen, D. (1998). Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5:63–108.

[Sun and Sun, 2009] Sun, D. and Sun, C. (2009). Context-based operational transformation in distributed collaborative editing systems. *IEEE Trans. Parallel Distrib. Syst.*, 20(10):1454–1470.

[Valdes, 1993] Valdes, R. (1993). Text editors: Algorithms and architectures, not much theory, but a lot of practice. *Dr. Dobb's Journal*, pages 38–43.