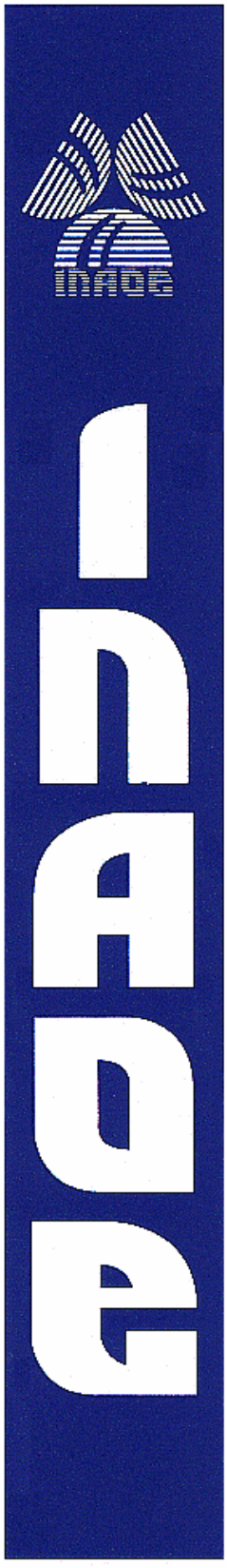

Mecanismo a Orden Causal Parcial sobre Canales de Comunicación no Fiables para la Sincronización de Flujos Continuos en Tiempo Real

Por

EDUARDO LÓPEZ DOMÍNGUEZ

Tesis sometida como requisito parcial
para obtener el grado de

MAESTRO EN CIENCIAS EN LA ESPECIALIDAD DE CIENCIAS COMPUTACIONALES

en el

Instituto Nacional de Astrofísica, Óptica y Electrónica
Enero 2007
Tonantzintla, Puebla

Supervisada por:

DR. SAÚL EDUARDO PORMARES HERNÁNDEZ

Para Marielena, Eduardo, Cinthia y Yesenia

Abstract

The synchronization of continuous media is a key issue in the development of distributed systems, such as videoconference, virtual reality, distance learning, among others. These services must be carried out in a synchronized manner with different media types, like continuous and discrete media. Due to the unreliable and asynchronous nature of networks, some problems can disrupt the synchronization. Such problems can include lost messages and delay jitter. In this work, we aim to solve the synchronization problem of real-time continuous media in unreliable networks. The detection and recovery of lost messages is achieved by the technique of forward error detection in a distributed form, which implies avoiding retransmission. As far as we know, our work is the first to propose a forward error recovery technique for the synchronization of continuous media with causality control. The mechanism is based on the temporal model proposed by Morales. The model determines logical dependencies between intervals to represent the basic temporal relations defined by Allen. On the other hand, In order to consider real-time constraints, we propose a distributed method to preserves real-time delivery constraints, avoiding the use of a global reference. Independent to each process and a local way, this method determine if a message has had a greater delay than its life-time.

Resumen

La sincronización de flujos continuos es un problema clave en el desarrollo de sistemas distribuidos, tales como videoconferencias, realidad virtual, aprendizaje a distancia entre otros. Estos servicios deben reproducir diferentes tipos de datos en forma sincronizada, tales como datos continuos y datos discretos. Debido a la naturaleza no fiable y asíncrona de las redes, algunos factores pueden romper la sincronización. Tales factores incluyen la pérdida y el retraso de mensajes (jitter). En este trabajo, nosotros resolvemos principalmente el problema de la sincronización de flujos continuos en tiempo real en canales de comunicación no fiables. La detección y recuperación de mensajes perdidos es llevada a cabo mediante la técnica de corrección de errores hacia adelante en forma distribuida, evitando la retransmisión. Hasta donde sabemos, nuestro trabajo es el primero en proponer una técnica de recuperación de errores hacia adelante para la sincronización de flujos continuos con control de causalidad. El mecanismo esta basado en el modelo de sincronización temporal propuesto por morales. El modelo determina las dependencias lógicas entre intervalos para representar las relaciones temporales básicas definidas por Allen. Por otra parte, con el objetivo de considerar restricciones de tiempo real, nosotros proponemos un método distribuido para preservar las restricciones de entrega en tiempo real, evitando el uso de una referencia global. Este método determina de forma local e independiente a cada proceso, si un mensaje ha sufrido un retraso mayor a su tiempo de vida.

Reconocimientos

Sin lugar a dudas mi total agradecimiento para Dios, que nunca me ha dejado solo y siempre me ha respondido cuando lo he necesitado, aun cuando algunas veces me he olvidado de darle gracias por su inmensa ayuda y sobre todo por su infinita comprensión.

A lo más hermoso que tiene el ser humano, a ti Madre querida que nunca dejaste que doblará las manos con nada, tú que siempre me brindaste ánimos y fortaleza con tu ejemplo y palabras. A ti Padre, siempre fuerte y con carácter, dándome a entender que con esfuerzo, trabajo y fe nada es imposible, y por último a mi hermana siempre viendo hacia adelante. A ustedes que me dieron la vida y su apoyo, mi total cariño y amor.

Agradezco enormemente al Dr. Saúl E. Pomares Hernández, mi asesor, sus consejos tanto en el ámbito profesional como en el personal, a usted Dr. Saúl le doy las gracias por su constante orientación y paciencia.

Agradezco también al Consejo Nacional de Ciencia y Tecnología (CONACyT) por el apoyo económico brindado al becario No. 189895.

Gracias al Instituto Nacional de Astrofísica, Óptica y Electrónica por la excelente formación académica otorgada y, los servicios y apoyos facilitados.

A mis compañeros y amigos, Edith, Diana, Rosita, Claudia, Adriana, Jorge, Raúl, Saúl, Esaú, Antonio, Juan, José, Luís que me brindaron su ayuda y amistad, muchas gracias.

Para terminar, agradezco a una persona muy especial que ha hecho que mi vida sea total felicidad, siempre apoyándome y entendiéndome aun en los momentos más difíciles. Gracias Yesenia por estar siempre a mi lado.

Tonantzintla, Puebla Eduardo López Domínguez

Enero 15, 2007

Índice General

Índice De Figuras

Índice De Tablas

Tabla de símbolos

SÍMBOLOS USADOS EN DEFINICIONES		SÍMBOLOS USADOS EN ALGORITMOS	
SÍMBOLO	**SIGNIFICADO**	**SÍMBOLO**	**SIGNIFICADO**
$\rightarrow$	Relación Causal	$\leftarrow$	Asignación de elementos en conjuntos
$\Leftrightarrow$	Doble implicación	/	Eliminación o resta de conjuntos
$\downarrow$	Relación de dependencia inmediata (IDR)	$\equiv$	Asignación a mensajes
$\wedge$	Conjunción	$\cup$	Unión
$\forall$	Cuantificador universal	=	Asignación de elementos con valores escalares
$\in$	Pertenencia		
$\neg$	Negación		
$\Rightarrow$	Implicación		
$\rightarrow_I$	Relación causal a nivel intervalo		
$\exists$	Cuantificador existencial		
|	Tal que (Descriptor)		
|||	Simultaneidad		
||	Paralelo		
$\times$	Producto cartesiano		

Capítulo 1

Introducción

Aplicaciones como videoconferencias, sistemas móviles, realidad virtual, entre otras, son ejemplos de sistemas distribuidos. Específicamente, un sistema distribuido se puede definir como un conjunto de componentes hardware/software con las siguientes características: la comunicación y coordinación se establece por medio del paso de mensajes, carecen de un reloj global y no poseen memoria compartida.

El desarrollo de sistemas distribuidos que intercambian flujos de datos continuos (audio y video) en tiempo real, ha tomado una gran relevancia en los últimos años. Ejemplos de multimedia distribuida los podemos encontrar en videoconferencias, difusión de radio o televisión en Internet, videotelefonía, aprendizaje a distancia, realidad virtual, trabajo cooperativo, entre otros.

Dentro de las principales características que deben cumplir los sistemas multimedia, se encuentran la sincronización de los flujos transmitidos y la preservación de las restricciones de entrega en tiempo real (*tiempo de vida*). La sincronización consiste en asegurar la correcta apariencia temporal de los flujos continuos (audio, video, animación, entre otros). Para contemplar las restricciones de entrega en tiempo real a cada mensaje se le asocia un *tiempo de vida*, después del cual sus datos son considerados no útiles. El *tiempo de vida* de un mensaje se puede definir como el periodo de tiempo que la aplicación en cuestión considera útil a los datos transmitidos en el mensaje. Si un mensaje sufre un retraso mayor a su *tiempo de vida* es considerado no útil por la aplicación y es descartado.

El problema de la sincronización de flujos continuos es aún más complicado cuando se consideran aspectos de comunicación reales, tales como pérdida y retraso de mensajes, los cuales son los dos principales problemas encontrados en los canales de comunicación.

Nuestro trabajo está centrado en la sincronización de flujos continuos sobre canales de *comunicación no fiables* (existe la pérdida de mensajes) *y asíncronos*, preservando las restricciones de entrega en tiempo real. Nos referimos por el término tiempo real a la creación y transmisión simultánea de flujos continuos. Esto implica que no existe pre-procesamiento ni almacenamiento previo de los flujos.

1.1 Problemática

En canales de comunicación reales, factores cómo la pérdida y el retraso arbitrario de los mensajes pueden romper la sincronización de los flujos continuos y perturbar las restricciones de entrega en tiempo real.

En este trabajo, abordamos el problema de la sincronización de flujos continuos cuando se produce la pérdida y el retraso arbitrario de mensajes, considerando restricciones de entrega en tiempo real entre mensajes de un mismo flujo y de diferentes flujos.

1.2 Objetivo

Desarrollar un algoritmo de sincronización de flujos continuos en tiempo real considerando características reales de redes como pérdida y retraso de mensajes.

1.3 Propuesta de solución

Nuestra propuesta de solución está dividida en 3 fases principales: La primera consiste en realizar la sincronización de flujos continuos usando el mecanismo presentado en [MOR05]. En la segunda fase, desarrollamos un mecanismo de recuperación hacia adelante de mensajes causales perdidos con la finalidad de llevar a cabo la sincronización en canales de comunicación no fiables. En la tercera y última fase, preservamos las restricciones de entrega en tiempo real de los flujos transmitidos mediante una novedosa forma para determinar si un mensaje ha excedido su *tiempo de vida*. A continuación damos una breve descripción de cada una de ellas, posteriormente en los siguientes capítulos haremos una presentación exhaustiva.

Fase 1. Mecanismo de sincronización: Para realizar la sincronización de los flujos continuos usamos el mecanismo de sincronización propuesto en [MOR05], el cual asume canales de *comunicación fiables* (no existe la pérdida de mensajes). El mecanismo de sincronización realiza el mapeo lógico de cualquier relación temporal entre dos flujos. El mapeo lógico expresa una relación temporal basándose únicamente en sus dependencias lógicas, descompone una relación temporal en cuatro segmentos identificando sus dependencias lógicas. El proceso para realizar el mapeo lógico toma cualquier par de intervalos en el sistema que componen un escenario temporal, y transforma cada par de intervalos en cuatro segmentos, los cuales son determinados acorde a la posible relación de precedencia de los eventos discretos que componen dichos intervalos. Los segmentos de datos son considerados nuevos intervalos. Una descripción más detallada del mecanismo de sincronización de flujos continuos es presentada en el capítulo 4.

Fase 2. Mecanismo de detección y recuperación causal hacia adelante: Con la finalidad de llevar a cabo la sincronización en *canales de comunicación no fiables*, desarrollamos un mecanismo de detección y recuperación causal hacia adelante de mensajes perdidos. La detección y recuperación es lograda mediante la técnica de corrección de errores hacia delante en forma distribuida. Las aplicaciones en tiempo real son sensibles al retraso, y consecuentemente, no hay tiempo para la retransmisión de mensajes perdidos, el mecanismo desarrollado recupera los mensajes perdidos evitando la retransmisión. La recuperación hacia adelante se realiza a través de la adición de redundancia. Dos tipos de redundancia son usados: *redundancia sobre el tipo de mensaje* y *redundancia sobre la información de control causal* enviada. Con el objetivo de disminuir la complejidad del problema, el mecanismo de detección y recuperación causal de mensajes perdidos fue desarrollado, primero a nivel de eventos discretos sobre el algoritmo causal mostrado en [POM02], el cual representa los conceptos base del algoritmo de sincronización de flujos continuos presentado en [MOR05]. Posteriormente, se extendió el algoritmo de sincronización de flujos continuos con el mecanismo de recuperación causal desarrollado a nivel de mensajes discretos. Una descripción a detalle del mecanismo de recuperación de mensajes discretos es mostrada en el capítulo 3. En el capítulo 4, presentamos cómo se incorpora el mecanismo de recuperación de mensajes perdidos, desarrollado en el capítulo 3, en el algoritmo de sincronización de flujos continuos.

Fase 3. Mecanismo para preservar las restricciones de entrega en tiempo real: Para contemplar las restricciones de entrega en tiempo real, consideramos que cada mensaje transmitido posee un *tiempo de vida* después del cual sus datos no son útiles para la aplicación. Si un mensaje sufre durante su transmisión un retraso mayor a su *tiempo de vida*, debe ser descartado. Para mantener las restricciones de entrega en tiempo real, proponemos un mecanismo distribuido que determina si un mensaje es recibido fuera de su *tiempo de vida*. El mecanismo desarrollado utiliza los relojes físicos de cada participante de forma independiente. No se asume ninguna referencia global dentro del sistema, es decir; los relojes físicos de los participantes no están sincronizados. En el capítulo 3, se presenta un mecanismo para preservar las restricciones de entrega en tiempo real a nivel de mensajes discretos. En el capítulo 4, con la finalidad de preservar las restricciones de entrega en tiempo real en el mecanismo de sincronización de flujos continuos, incorporamos el mecanismo desarrollado a nivel de mensajes discretos explicado en el capítulo 3.

1.4 Organización de la tesis

Este trabajo se encuentra dividido en 6 capítulos. El capítulo 1 contiene la introducción, problemática, objetivo y la solución propuesta descrita de forma general. Posteriormente, el capítulo 2 incluye el estado del arte donde se describen los principales trabajos que han sido desarrollados para lograr la sincronización de flujos continuos en canales de comunicación no fiables. Después, en el capítulo 3, describimos cómo la redundancia sobre la información de control enviada es utilizada para tolerar la pérdida de mensajes. Dentro de este capítulo, también, presentamos el esquema utilizado para determinar si un mensaje ha sufrido un retraso mayor a su tiempo de vida, ambos mecanismos son desarrollados sobre

un algoritmo causal *broadcast*, el cual considera únicamente la transmisión y recepción de mensajes causales en tiempo real. En el capítulo 4, especificamos cómo el mecanismo de detección y recuperación casual hacia adelante de mensajes perdidos, presentado en el capítulo 3, es aplicado al algoritmo de sincronización de flujos continuos. La finalidad de está extensión es lograr la sincronización aún cuando se produce la pérdida de mensajes perdidos. De igual forma, se describe cómo es aplicado sobre el mecanismo de sincronización, el esquema que preserva las restricciones de entrega en tiempo real en el algoritmo *broadcast*. En el capítulo 5, presentamos los resultados obtenidos de la emulación del mecanismo de sincronización de flujos continuos sobre canales de comunicación no fiables y asíncronos. Por último, en el capítulo 6 se exponen las conclusiones y el trabajo futuro de la tesis desarrollada.

Capítulo 2

Estado del Arte

Aplicaciones distribuidas como videoconferencia intercambian flujos de datos (audio y video) en tiempo real. En este tipo de aplicaciones se requiere de una transmisión asíncrona de los datos pero una reproducción síncrona por el lado del cliente. La sincronización consiste en asegurar la correcta apariencia temporal de los flujos continuos (audio y video), en otras palabras, lo que vemos debe corresponder con lo que escuchamos [WAH94].

Debido a la naturaleza no fiable y asíncrona de la red, algunos factores pueden romper la sincronización. Tales factores pueden incluir la perdida de mensajes y el retraso de mensajes durante la transmisión. Diversos enfoques se han planteado para realizar la sincronización de flujos continuos en canales de comunicación no fiables y asíncronos. Los trabajos que contemplan la pérdida de mensajes basan su mecanismo de tolerancia a la pérdida de información en alguna de las técnicas de recuperación existentes. Las técnicas usadas para la recuperación de mensajes perdidos se clasifican en dos principales categorías: Solicitud de repetición automática (*Automatic Repeat Request*, ARQ) y corrección de errores hacia adelante (*Forward error correction*, FEC). Trabajos que basan su mecanismo de recuperación en la técnica ARQ detectan el mensaje perdido y realizan la retransmisión del mensaje.

Basados en el participante que realiza la retransmisión, podemos clasificar estos trabajos en: retransmisión por emisor y retransmisión por destino. En estos trabajos, la retransmisión de los mensajes perdidos introduce retrasos que afectan la interactividad de la aplicación en cuestión. La técnica ARQ no es adecuada para aplicaciones en tiempo real, debido a que éstas son sensibles al retraso. En otras palabras, no hay tiempo para realizar la retransmisión de los mensajes perdidos. Por otra parte, existen algoritmos que basan su mecanismo de recuperación en la técnica de corrección de errores hacia adelante (FEC). La técnica FEC agrega información redundante al flujo de datos a transmitir. Con la información redundante el receptor puede recuperar los mensajes perdidos. Esta técnica es adecuada para aplicaciones en tiempo real, debido a que evita la retransmisión de la información perdida.

En este capítulo presentamos principalmente un estudio de los trabajos que resuelven la sincronización de flujos continuos considerando *canales de comunicación no fiables*. Dividimos el capítulo en dos secciones. La primera sección realiza una clasificación del estado del arte relacionado con la sincronización multimedia. Esta sección posiciona el

problema que esta tesis resuelve. En la segunda sección, se describen a detalle los trabajos que se enfocan a realizar la sincronización de flujos continuos cuando se produce la pérdida de mensajes.

2.1 Sincronización de flujos continuos en tiempo real

Se pueden clasificar los trabajos que intentan resolver el problema de la sincronización de flujos continuos (audio y video) en dos principales categorías: tiempo-real y en demanda. Esta investigación está orientada a la sincronización en tiempo real. La sincronización en tiempo real puede ser dividida en eventos continuos y eventos discretos. Los eventos continuos (flujos) son patrones repetitivos de eventos relacionados, los eventos discretos sincronizan actividades en un sistema distribuido por respuesta a eventos. Este trabajo está orientado al problema de la sincronización de flujos continuos.

La sincronización de flujos continuos se divide en intra-flujo e inter-flujo. La sincronización intra-flujo se refiere a la preservación de las dependencias temporales físicas entre mensajes de un mismo flujo. La sincronización inter-flujo está enfocada a preservar las dependencias lógicas y físicas entre diversos flujos. El objetivo de este trabajo contempla el problema de la sincronización inter-flujo.

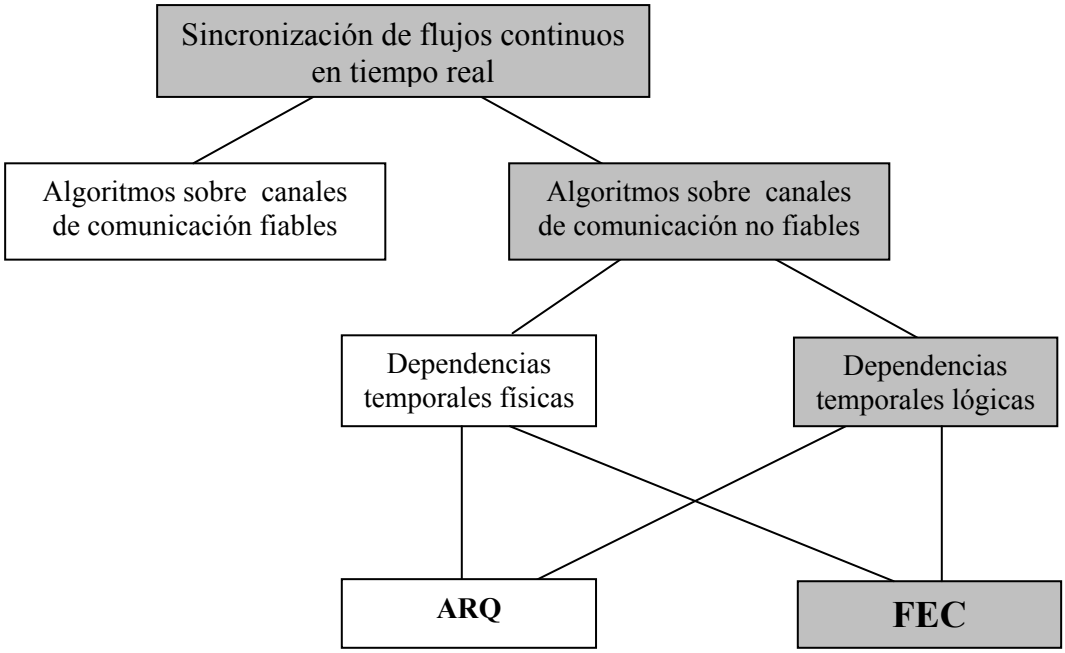

Figura 2.1. Clasificación de la sincronización de flujos continuos basado en el canal de comunicación. Los recuadros en gris señalan el enfoque del presente trabajo.

Basados en el tipo de canal de comunicación usado, se pueden clasificar los algoritmos que resuelven el problema de la sincronización de flujos continuos en tiempo real en dos principales categorías (figura 2.1.):

- Algoritmos sobre canales de comunicación fiables
- Algoritmos sobre canales de comunicación no fiables

La primera categoría asume que no existen mensajes perdidos durante su transmisión, por lo tanto, los trabajos en esta categoría no abordan los efectos negativos que causan los mensajes perdidos en la sincronización de flujos continuos. En la segunda categoría, encontramos los trabajos o algoritmos que contemplan la perdida de mensajes durante su transmisión, estos trabajos presentan mecanismos que contemplan la pérdida de información. A continuación presentaremos brevemente algunos de los trabajos que se encuentran en cada clasificación.

2.1.1 Sincronización sobre canales de comunicación fiables

Dentro de esta clasificación se encuentra [BEN00], en esta investigación se propone un protocolo de sincronización multimedia que permite una reproducción síncrona cuando la red posee desconocidos límites de retrasos en los mensajes. Este trabajo utiliza mensajes *feedback* (mensajes de retroalimentación), desde el receptor a la fuente, para informar del retraso sufrido desde el cliente al servidor y utilizan el manejo de *buffer* en los clientes para mantener la sincronización. Esta investigación supone que los mensajes transmitidos entre el servidor y el cliente toman el mismo camino y por lo tanto sufren el mismo retraso. En condiciones reales de red los retrasos de los mensajes son aleatorios e impredecibles. Al suponer una arquitectura cliente/servidor se genera un cuello de botella, y en consecuencia, se producen retrasos sobre la transmisión y recepción de los flujos, afectando la calidad de servicio de la aplicación y también limitando la escalabilidad. Todo el sistema se adecua al retraso máximo sufrido en un canal de comunicación, forzando a los demás participantes a contemplar el retraso máximo.

El concepto de tiempo virtual maestro (virtual master time, VMT) es otro mecanismo propuesto para la sincronización de flujos continuos [ABO99]. El VMT es una referencia de tiempo interna mantenida por un participante del sistema. Todos los participantes en el grupo sincronizan su reloj local físico con el VTM. Para realizar la sincronización entre flujos, cada mensaje es marcado con el tiempo de envío. Cuando el receptor recibe el mensaje, le agrega el retraso máximo que puede sufrir durante su transmisión y lo almacena en el *buffer*. El receptor retrasa la entrega del mensaje recibido a la aplicación hasta que su tiempo local alcance el valor obtenido de la suma del tiempo de envío y el retraso máximo del mensaje recibido. De esta forma, todos los receptores entregan los mensajes a la aplicación de manera simultánea para reproducirlos sobre el mismo tiempo. Este enfoque depende de la exactitud de sincronización de los participantes con el VMT en el grupo. Implícitamente en esta fase se agrega un error de precisión, el cual se incrementa en sistemas dispersos geográficamente, afectando la sincronización entre los flujos envíados en el grupo.

Otra propuesta para realizar la sincronización inter-flujos la podemos encontrar en [YAN99]. Se plantea el uso de un protocolo de sincronización de relojes físicos (*Network Time Protocol*, NTP) para establecer un tiempo global en el sistema distribuido. La sincronización multimedia está basada en el uso de la referencia global. La efectividad del algoritmo presentado en [YAN99] depende totalmente de la precisión del protocolo de sincronización de relojes. Al suponer una referencia global, inherentemente se agrega un error de precisión provocado por NTP y la asunción de una arquitectura de red jerárquica. En sistemas distribuidos dispersos geográficamente, el error de precisión entre los relojes físicos de cada participante es incremental, como consecuencia el establecimiento de un tiempo global se convierte en un problema complejo de resolver.

En la siguiente sección describiremos los trabajos enfocados a mantener la sincronización de flujos continuos aún en condiciones de pérdida de mensajes. Estos trabajos son de mayor relevancia a la propuesta de tesis, debido a la similitud de las condiciones de los canales de comunicación propuestos.

2.1.2 Sincronización sobre canales de comunicación no fiables

En la sección anterior presentamos algunos trabajos que consideran condiciones ideales en los canales de comunicación. Sin embargo, existen algunos ambientes donde necesitamos considerar canales de comunicación no fiables (existe la perdida y el retraso de mensajes), como es el caso de la transmisión de flujos continuos en tiempo real. Los algoritmos descritos en esta sección intentan resolver la sincronización de flujos continuos en tiempo real aun en condiciones de pérdida.

Estos trabajos los podemos dividir en dos categorías con base en el mecanismo usado para realizar la sincronización de flujos continuos en tiempo real (figura 2.1). Trabajos en la primera categoría utilizan dependencias de tiempo físico. Los trabajos de la segunda categoría se basan en dependencias de tiempo lógico. Debido a la importancia de estas dos categorías de trabajos para la presente tesis, dedicamos las siguientes secciones para presentar los trabajos más importantes.

2.2 Sincronización de flujos continuos en tiempo real sobre canales de comunicación no fiables

2.2.1 Dependencias de tiempo físico

Los algoritmos que se basan en dependencias de tiempo físico plantean el uso de una referencia global para establecer el tiempo de envío de los mensajes. El tiempo de envío junto con la información de los mensajes difundidos anteriormente en el sistema determinan el orden de entrega de los mensajes recibidos. Uno de los enfoques más utilizados para establecer una referencia global, es la sincronización de relojes fiscos. Cada participante del sistema sincroniza su reloj local físico con una referencia interna (miembro del sistema) o externa (GPS, reloj atómico), mediante algún algoritmo de sincronización de

relojes físicos [GUS89], [MIL95]. La efectividad de estos trabajos depende totalmente de la precisión del algoritmo de sincronización de relojes. Al suponer una referencia global, inherentemente se involucra un error de precisión provocado por la distinta velocidad a la que corren los relojes físicos. En sistemas distribuidos con retrasos variables en los canales de comunicación, el establecer un tiempo global se convierte en un problema difícil de resolver.

Para tolerar la pérdida de mensajes, los trabajos que utilizan dependencias de tiempo físico, basan sus mecanismos de tolerancia a la pérdida de información en alguna de las técnicas de recuperación existentes. Las técnicas de recuperación de mensajes perdidos se dividen en dos categorías (figura 1): solicitud de repetición automática (*Automatic Repeat Request*, **ARQ**) y la técnica de corrección de errores hacia adelante (*Forward Error Correction*, **FEC**). Los trabajos que utilizan ARQ recuperan los mensajes mediante la retransmisión del mensaje original. La retransmisión de los mensajes perdidos introduce retrasos, por lo tanto, la técnica ARQ no es adecuada para aplicaciones en tiempo real debido a que son muy sensibles al retraso. Un enfoque propuesto para aplicaciones en tiempo real que evita la retransmisión, es la técnica FEC. Los trabajos que usan la técnica FEC agregan información redundante al flujo de datos a transmitir. El receptor utiliza la información redundante para recuperar los mensajes perdidos. La técnica FEC ofrece exacta o aproximada reconstrucción de los datos transmitidos en caso de pérdida. Los trabajos en esta categoría son recomendados para sistemas en tiempo real debido a que evitan la retransmisión.

A continuación presentamos brevemente los trabajos más interesantes a nuestro juicío que utilizan ARQ y posteriormente describimos detalladamente los trabajos que utilizan FEC, los cuales son de mayor relevancia para el tema de tesis.

2.2.1.1 Solicitud de repetición automática (*Automatic Repeat Request, ARQ*)

Trabajos en esta clasificación usan la técnica de solicitud de repetición automática o alguna variante para recuperar los mensajes. Detectan la pérdida de un mensaje y recuperan el mensaje mediante la retransmisión. Estos trabajos los podemos dividir en dos categorías:

- En la primera categoría la retransmisión la realiza directamente el emisor del mensaje.
- En la segunda categoría la retransmisión del mensaje perdido es realizada de forma indirecta. Dentro de esta categoría encontramos distintas variantes, las cuales se diferencian por el método empleado para elegir al miembro del sistema que realiza la retransmisión. En algunos trabajos la retransmisión la realiza el participante con el menor retraso promedio al destino donde se perdió el mensaje. En el peor de los casos la retransmisión la realiza el emisor original del mensaje.

A continuación describimos algunos trabajos en donde la retransmisión la realiza el emisor original del mensaje.

Una extensión al protocolo de tiempo real (*Real Time Protocol*, RTP) para proporcionar *retransmisión selectiva* es presentada en [FEA02]. Su arquitectura está compuesta por un flujo *unicast* de video MPEG4 entre un servidor y un cliente. Información de las condiciones de la red (latencia, tasa de mensajes perdidos, congestión, etc.) son proporcionadas a través de los reportes del protocolo de control de tiempo real del receptor. Un video MPEG4 está formado por tres tipos de frames: I, P y B. Cada unidad de datos enviada al cliente representa un frame de video y es etiquetada con un número de secuencia, en forma ascendente. Sobre la recepción de un paquete, el cliente envía al servidor un mensaje de recepción exitosa del paquete (*acknowledgment*, ACK). El cliente puede detectar la pérdida de un mensaje mediante un salto en el número de secuencia. La solicitud de retrasmisión se realizará sólo si la unidad de datos perdida es un frame I. En caso contrario, la recuperación de los frames P y B es realizada mediante técnicas pos-procesamiento. En este trabajo, la retransmisión afecta la interactividad de la aplicación debido a que introduce retrasos, y por lo tanto, perturba la sincronización de los flujos.

En [DWY98] se propone un protocolo de transporte para el flujo de paquetes heterogéneos (*heterogeneous packet flows*, HPF) en un ambiente como Internet. En este protocolo se propone desacoplar el control de congestión del mecanismo de fiabilidad a diferencia de TCP. HPF soporta transmisión fiable para paquetes con alta prioridad y no fiable transmisión para paquetes con baja prioridad. Sólo cuando se envía un paquete con prioridad alta, el emisor lanza un temporizador. Un mensaje de recepción exitosa del paquete enviado debe llegar en el periodo de tiempo establecido por el temporizador. En caso contrario, el emisor retransmite el paquete perdido. Paquetes perdidos con prioridad baja nunca son retransmitidos. Un escenario de prueba es presentado utilizando frames de video MPEG, asignando prioridades altas a frames I.

Un protocolo fiable para flujos de video producidos en tiempo real es presentado en [HAS1998]. En este trabajo suponen una referencia global en el sistema distribuido mediante el uso del protocolo de tiempo de red (*Network Time Protocol*, NTP). Utilizando la referencia global, el receptor puede determinar si un mensaje ha excedido su tiempo de vida útil a la aplicación. El receptor detecta la pérdida de mensajes examinando el número de secuencia. Cuando un mensaje es perdido, una petición de retransmisión es realizada al emisor del mensaje. La retransmisión del mensaje perdido, sólo es realizada si el emisor determina que el mensaje retransmitido llegará a su destino dentro de su tiempo de vida útil. Debido a que [HAS1998] basa su mecanismo en una referencia global, implícitamente se agrega un error de precisión, lo cual afecta la sincronización. En sistemas distribuidos formados por participantes con retrasos variables en los canales de comunicación, el error de precisión es más grande, y por lo tanto, establecer una referencia global se vuelve un problema difícil a resolver.

Dentro de los trabajos que realizan la retransmisión por destino encontramos los siguientes.

Diversos trabajos plantean el uso de algoritmos *multicast* a nivel capa-aplicación para establecer una comunicación fiable entre participantes. Algunos trabajos proponen el uso de un protocolo *multicast* jerárquico construyendo un árbol lógico sobre una red física como Internet [BAN04, AMI00]. En estos trabajos los mensajes son transmitidos a todos o a un subconjunto de los participantes. Sobre la recepción del mensaje, los participantes adelantan o envían el mensaje a otros destinos. La propagación del mensaje es en forma de árbol y está basada en algoritmos de ruteo como los propuestos en [MOL91], [JIA95]. Mensajes de recepción exitosa (*acknowledgment*, ACK) son enviados por la llegada de los mensajes a su destino. Mensajes de no recepción (*negative acknowledgment*, NACK) son enviados para indicar la pérdida de mensajes. Cuando un NACK es difundido, el participante encargado de realizar la retransmisión del mensaje perdido es el más cercano en términos de latencia, al emisor del NACK. Este trabajo, al basar su mecanismo de recuperación en la retransmisión de los mensajes perdidos agrega retrasos extras, los cuales afectan la interactividad de la aplicación.

Otra variante de los algoritmos *multicast* forma un árbol lógico y divide en subgrupos a los participantes [HOF95, PAU96]. Cada subgrupo elige un participante como coordinador. Los coordinadores son los responsables de difundir un mensaje entre dos participantes de diferentes subgrupos. Un participante dentro de un subgrupo envía un mensaje a su coordinador. El coordinador envía el mensaje al grupo de coordinadores. Los coordinadores de cada subgrupo realizan el envió del mensaje a los miembros de su grupo. Cuando un mensaje es perdido por un miembro del sistema. El coordinador local es el encargado de retransmitir el mensaje perdido. Si el coordinador local no tiene en su buffer el mensaje perdido, realiza una solicitud de retransmisión al conjunto de coordinadores. En el peor de los casos, el emisor original retransmite el mensaje perdido. La retransmisión de mensajes perdidos agrega retrasos y afecta la interactividad de la aplicación en cuestión. Consecuentemente está estrategia no es adecuado para aplicaciones en tiempo real debido a que es muy sensible a los retrasos.

2.2.1.2 Corrección de errores hacia adelante (*Forward Error Correction*, FEC)

Otra técnica para recuperar los mensajes perdidos es la de corrección de errores hacia adelante. Dentro de los trabajos que utilizan la técnica FEC o alguna variante encontramos los siguientes.

El protocolo de transporte de tiempo real (*real time transport protocol*, RTP) utiliza esquemas de tipo FEC para brindar tolerancia a la perdida de información [PER00]. La cantidad de información redundante a agregar depende de las condiciones de pérdida de la red. Los reportes RTCP del receptor brindan información acerca de la tasa de pérdida en los canales de comunicación. El emisor RTP determina mediante esta información la cantidad de información redundante a agregar para recuperar los mensajes perdidos. El FEC de paridad es una de las más importantes técnicas de corrección de errores hacia delante empleada por RTP. En esta técnica un paquete FEC es generado a través del XOR de los

paquetes de datos originales. Los paquetes FEC son utilizados para detectar y recuperar los mensajes perdidos. La figura 2.2 describe el funcionamiento del esquema *FEC de paridad*.

En el ejemplo mostrado en la figura 2.2, la información redundante es obtenida del XOR de los paquetes 1, 2, 3 y 4. Si uno de los paquetes originales es perdido, puede ser recuperado a través del XOR de los 3 paquetes restantes y el paquete FEC. En este caso, el paquete 3 es perdido durante su transmisión, el receptor recupera el paquete perdido a partir del XOR de los paquetes 1, 2, 4 y FEC. Una descripción más detallada del FEC de paridad es presentada en el estándar definido por el RFC2733.

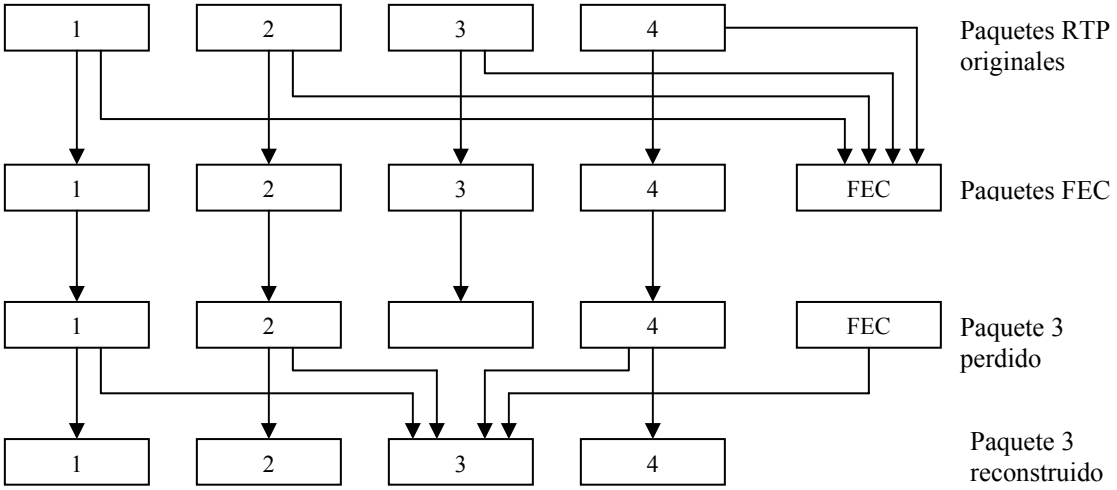

Figura 2.2. Funcionamiento del FEC de paridad.

Un confiable *multicast* para la transmisión de flujos de datos en tiempo real es presentado en [RUB98]. En este trabajo, se utiliza una combinación de FEC y ARQ para recuperar los paquetes perdidos. El emisor crea paquetes de recuperación a partir de bloques de paquetes, los cuales son los datos a difundir. En caso de pérdida de información, los paquetes de recuperación son utilizados por el receptor para recuperar los paquetes perdidos durante su transmisión. Cuando los paquetes de recuperación son insuficientes para recuperar totalmente los paquetes perdidos, el receptor difunde un NACK para solicitar la transmisión de paquetes de recuperación extra con la finalidad de lograr la completa entrega de la información pérdida. El encargado de realizar la creación y transmisión es el emisor original del flujo de datos. Otro enfoque similar es presentado en [NON97], cuya diferencia principal radica en el método utilizado para recuperar la información perdida cuando la información redundante es insuficiente. En [NON97] el receptor difunde un NACK al emisor original de los datos. El emisor en vez de enviar información redundante extra, envía los datos originales que fueron perdidos.

Utilizar un esquema FEC como el de paridad incrementa el ancho de banda necesario para transmitir un flujo continuo. Por otra parte, con un ancho de banda limitado la información redundante a agregar será condicionada por el ancho de banda disponible y puede no ser la necesaria para tolerar la perdida de los mensajes. En casos donde la pérdida de información

es provocada por la congestión de la red, agregar información redundante puede tener efectos negativos como empeorar la tasa de pérdida que puede soportar la información redundante. Otro aspecto importante a considerar es que la detección y recuperación de los paquetes perdidos es realizada hasta la recepción del paquete FEC. Si los paquetes FEC enviados sufren un retraso mayor que los paquetes de datos que ellos protegen, el receptor debe elegir entre reproducir los datos dañados o esperar la llegada del paquete FEC para recuperarlos. En el último caso, el retraso punto a punto es incrementado afectando principalmente a aplicaciones interactivas en las cuales es importante mantener un margen de retraso bajo. Todos estos factores afectan la interactividad y la sincronización de los flujos continuos en tiempo real.

2.2.2 Dependencias de tiempo lógico

Debido a que no es posible establecer una referencia global en un sistema distribuido de manera exacta para determinar el orden de envío de los mensajes, surgió el mecanismo basado en dependencias lógicas. Las dependencias temporales lógicas son establecidas por medio de variables numéricas. Cada variable representa el número de mensajes recibidos/enviados por los miembros del sistema. La información que proporcionan las variables es utilizada para establecer el orden de entrega de los mensajes. Dentro de los mecanismos más usados para preservar las dependencias lógicas encontramos los algoritmos causales. Estos algoritmos son utilizados cuando se desea realizar una sincronización en un ambiente distribuido. Basados en la técnica de recuperación utilizada por los algoritmos causales, los podemos dividir en dos grupos: en el primer grupo se encuentran los algoritmos causales que utilizan ARQ para recuperar los mensajes perdidos. En el segundo grupo se encuentran los algoritmos casuales que utilizan la técnica FEC.

A continuación presentamos algunos de los trabajos más importantes en cada grupo.

2.2.2.1 Algoritmos causales con ARQ

Trabajos en esta clasificación detectan la pérdida de un mensaje y recuperan el mensaje mediante la retransmisión. Podemos dividir estos trabajos en dos grupos: retransmisión por emisor y retransmisión por destino.

Algoritmos causales en donde la retransmisión la realiza el emisor original del mensaje perdido son descritos a continuación.

La propuesta del uso de un canal *multimodal* es realizada en [PLE05]. Un canal *multimodal* está formado de dos canales independientes, un canal transmite mensajes FIFO y el segundo canal trasmite mensajes causales. Los mensajes causales son utilizados para preservar las relaciones inter-flujo (sincronización). La pérdida de un mensaje es informada mediante el envío de un mensaje de no recepción (*negative acknowledgment*, NACK). La recuperación es realizada por el emisor, retransmitiendo el mensaje perdido al destino.

Experimentos realizados en este trabajo muestran cómo es incrementada la latencia debido a las retrasmisiones de mensajes causales perdidos, afectando la sincronización entre flujos.

En [NAK94] se define un formato o estructura para los mensajes transmitidos. Dentro de la estructura planteada, se encuentra un campo con un número de secuencia para cada participante. El número de secuencia representa la cantidad de mensajes enviados de cada miembro del sistema. Con la recepción de un mensaje, el número de secuencia correspondiente al emisor incrementa en uno. Sobre la recepción de un mensaje, un participante puede detectar mediante el análisis de los números de secuencia, la pérdida de un mensaje enviado por algún miembro del sistema. Al detectar un mensaje perdido, se realiza una solicitud de repetición al emisor del mensaje. La recuperación es realizada por el emisor, retransmitiendo el mensaje perdido.

El trabajo desarrollado en [TAC96a] propone un protocolo *broadcast* para flujos continuos en tiempo real. Este trabajo plantea la entrega de flujos continuos a la aplicación tolerando un máximo de mensajes perdidos. Los flujos continuos son transmitidos como un conjunto de mensajes. Un participante puede recibir sólo un subconjunto de los mensajes que forman un flujo, debido a la pérdida de algunos mensajes durante su transmisión. Si la cantidad de mensajes perdidos es mayor a la tolerada por la aplicación, una petición de retransmisión de los mensajes perdidos es enviada al emisor. Esta estrategia supone una referencia global para establecer restricciones de tiempo (tiempo de vida) entre los mensajes que forman un flujo. Sobre la recepción de cada mensaje, el receptor determina si el mensaje es recibido dentro de su tiempo de vida. Si el mensaje ha excedido su tiempo de vida es descartado.

A continuación describiremos los trabajos más importantes sobre algoritmos causales en donde la retransmisión es por destino:

Una arquitectura jerárquica es propuesta en [BAL97]. Este trabajo divide en grupos lógicos a los participantes de un sistema distribuido. Cada miembro sólo puede pertenecer a un grupo lógico. En cada grupo un participante es elegido en forma determinística para ser el servidor causal. Todos los servidores causales forman un grupo llamado grupo de servidores causales. La transmisión de un mensaje en miembros de un mismo grupo se realiza mediante un *brodcast* local. Para difundir un mensaje a un miembro de un grupo diferente. El servidor causal espera hasta que el mensaje haya sido entregado causalmente en el grupo local, y entonces hace un *broadcast* a todos los miembros del grupo de servidores. Cada servidor causal recibe el mensaje y espera hasta que el mensaje haya sido entregado causalmente en el grupo de servidores. El servidor causal que tenga en su grupo al miembro a quien va dirigido el mensaje, realiza un *broadcast* del mensaje en el grupo que coordina. Mensajes perdidos son retransmitidos por el servidor causal del grupo. Si el servidor causal no tiene almacenado el mensaje realiza una petición de retransmisión al grupo de servidores causales. En el peor de los casos, el emisor original del mensaje realiza la retransmisión del mensaje perdido.

En [TAC97] se propone un protocolo de ordenamiento causal considerando restricciones de tiempo real y tasas de pérdida distintas en cada canal de comunicación. Cada participante

conoce el retraso y tasa de pérdida de los mensajes entre todos los miembros del sistema. Para realizar el envío de un mensaje, el emisor realiza un *broadcast* en el sistema. El emisor establece un temporizador (*timer*) para cada destino acorde al retraso sufrido en el canal de comunicación. El emisor debe recibir la confirmación de recepción exitosa (ACK) de todos los miembros del sistema antes de que expire su correspondiente *timer*. La ACK del mensaje transmitido puede ser de manera indirecta o directa. De manera indirecta es realizada a través de la recepción de otro mensaje difundido por algún participante del sistema, el cual lleva dentro de su estructura la ACK de todos los mensajes recibidos. De forma directa, se recibe un mensaje de recepción exitosa del receptor. Al expirar un *timer* se realiza la retransmisión del mensaje perdido. La retransmisión la realiza el participante con el retraso menor con respecto al destino del mensaje. Se plantean otros dos mecanismos para evitar la pérdida de mensajes durante su transmisión. El primero se denomina replicación de emisor (*sender replication*). Replicación por emisor consiste en enviar varias copias de un mensaje para asegurar que al menos una llegue al destino. La segunda es llamada replicación por destino. Replicación por destino consiste en enviar los mensajes recibidos previamente junto con la difusión del siguiente mensaje.

2.2.2.2 Algoritmos causales con FEC

Nuestra investigación se ubica dentro de la categoría de algoritmos causales FEC y propone un algoritmo que usa la relación causal a nivel intervalo para lograr la sincronización entre flujos. El trabajo presentado en esta tesis es, hasta donde sabemos, el primero en proponer una técnica de recuperación de errores hacia adelante para preservar la sincronización de flujos continuos con control de causalidad. La recuperación hacia adelante se obtiene introduciendo redundancia. La información redundante agregada se basa en la información de control y en el tipo de mensaje. Para una descripción detallada del mecanismo de recuperación ver el capítulo 4, sección 4.4. El algoritmo propuesto se recupera de mensajes perdidos en forma distribuida, evitando la retransmisión. Nuestra propuesta es adecuada para aplicaciones en tiempo real debido a que posee la característica de recuperación hacia adelante.

Capítulo 3

Algoritmo causal en tiempo real tolerante a la pérdida de información

En aplicaciones distribuidas, los algoritmos causales son indispensables para el intercambio de información. El uso de los algoritmos causales para el desarrollo de los sistemas distribuidos proporciona un orden en la entrega de los mensajes difundidos, llamado orden causal. El ordenamiento causal lleva a cabo la entrega de los mensajes en el mismo orden en que fueron enviados en el sistema. Los algoritmos causales también son utilizados para realizar la sincronización de flujos continuos (audio y video) en sistemas distribuidos como videoconferencias, sistemas multimedia, entre otros. Características inherentes de los canales de comunicación como la pérdida de mensajes pueden alterar el orden causal, y por consecuencia, el desempeño de los sistemas.

En este capítulo presentamos un algoritmo causal que transmite mensajes discretos en tiempo real sobre canales de comunicación no fiables (los mensajes pueden ser perdidos durante su transmisión). Con el fin de tolerar la pérdida de mensajes, desarrollamos un mecanismo de recuperación causal hacia delante de mensaje perdidos, evitando la retransmisión, el cual fue publicado en [LOP05]. Por otra parte, con el objetivo de proporcionar restricciones de entrega en tiempo real, consideramos que cada mensaje tiene un *tiempo de vida,* Δ*,* después del cual el contenido del mensaje no es útil. El *tiempo de vida* de un mensaje es el periodo de tiempo que una aplicación considera útil a los datos transmitidos en el mensaje. Si un mensaje sufre un retraso mayor a su *tiempo de vida* es descartado. Para mantener las restricciones de entrega en tiempo real, presentamos un mecanismo original que determina si el retraso sufrido por un mensaje es mayor a su *tiempo de vida*.

El algoritmo causal descrito en este capítulo representa los conceptos base sobre los cuales el mecanismo de sincronización de flujos continuos presentado en [MOR05] fue

desarrollado. Con el objetivo de disminuir el grado de complejidad, se optó por desarrollar primero un mecanismo de detección y recuperación causal hacia adelante de mensajes perdidos sobre el algoritmo causal base, para posteriormente aplicarlo al algoritmo de sincronización de flujos continuos. De la misma manera, se desarrolló sobre el algoritmo causal base, un mecanismo para preservar las restricciones de entrega en tiempo real, para posteriormente aplicarlo al algoritmo de sincronización de flujos continuos. La integración del algoritmo de sincronización de flujos continuos con el mecanismo de recuperación hacia adelante de mensajes perdidos y el mecanismo para preservar las restricciones de entrega en tiempo real es descrito a detalle en el capítulo 4. El algoritmo causal en tiempo real tolerante a la pérdida de mensajes presentado en este capítulo, forma parte de la segunda fase de la propuesta de solución descrita en el capítulo 1.

Este capítulo está estructurado de la siguiente forma. En la primera sección presentamos las definiciones que permiten establecer un orden causal en un sistema distribuido. La sección 3.2 describe a detalle el funcionamiento del mecanismo de recuperación causal hacia delante de mensajes perdidos desarrollado a nivel de eventos discretos. El mecanismo que permite determinar si un mensaje ha sufrido un retraso mayor a su tiempo de vida es descrito en la sección 3.3.

3.1. Bases y definiciones

El algoritmo causal que se presenta en este capítulo utiliza como fundamentos para establecer un orden de entrega entre los mensajes la *relación happened-before* y la *relación de dependencia inmediata*, las cuales son descritas a continuación.

3.1.1 Relación *happened-before* para eventos

El orden causal fue desarrollado con el objetivo de disminuir la asincronía de los canales de comunicación en los sistemas distribuidos; así mismo, evita inconsistencias en la entrega de los mensajes debido al retraso arbitrario agregado durante su transmisión. El orden causal está basado en la relación de precedencia causal definida por Lamport [LAM78]. Esta relación es un orden parcial con respecto a los eventos de envío y entrega ejecutados por un conjunto de procesos. Si *a* y *b* son dos eventos, entonces *a* precede a *b*, denotado como a $\rightarrow$ *b*, si cumple las siguientes condiciones:

Definición 1. La relación causal $\rightarrow$ es la menor relación de orden parcial que satisface una de las siguientes condiciones:

- Si *a* y *b* son eventos producidos por el mismo proceso, y *a* se origina antes que *b*, entonces $a \rightarrow b$.
- Si *a* es el evento de envío de un mensaje *m* y *b* es el evento de entrega del mismo mensaje en otro proceso, entonces $a \rightarrow b$.
- Si existe un evento *c* tal que $a \rightarrow c$ y $c \rightarrow b$ entonces $a \rightarrow$b.

La precedencia causal es extendida a los mensajes en la siguiente forma: $m \rightarrow m$' sí y sólo sí *send*(*m*)$\rightarrow$ *send*(*m'*). La entrega causal ordenada permite asegurar el envío y recepción de mensajes causales en la comunicación en grupo. La entrega causal ordenada en la comunicación en grupo presenta dos casos: el caso *broadcast* (un grupo) y el caso multi-grupos, que incluye la superposición de grupos. En este trabajo presentamos sólo la definición del caso *broadcast*, que es la entrega causal en que estamos interesados (definición 2). Para el caso multi-grupos consulte [POM04]. La entrega causal para el caso *broadcast* es como sigue [BIR93]:

Definición 2. Entrega causal *broadcast* (un solo grupo):

Sí *send*(*m*)$\rightarrow$ *send*(*m*'), **entonces** $\forall\ k \in c$:
deliver(k,m) $\rightarrow$ *deliver*(*k,m'*)

La entrega causal *broadcast* asegura que si la difusión de un mensaje *m* causalmente precede la difusión de un mensaje *m'*, en un grupo *c*, entonces la entrega de *m* causalmente precede la entrega de *m'* para cualquier participante *k* que pertenece a *c*.

La relación de precedencia de mensajes denotado por $m \rightarrow m'$ es inducida por la relación de precedencia en eventos, definida por:

$$m \rightarrow m' \Leftrightarrow \text{send}(m) \rightarrow \text{send}(m')$$

3.1.2 Relación de dependencia inmediata (IDR):

La relación de dependencia inmediata es el umbral de propagación de la información de control CI. La IDR es la cantidad mínima de *CI*, necesaria y suficiente, que debe ser transmitida en cada mensaje para asegurar una entrega causal y por lo tanto mantener la coherencia en el sistema. Su definición es la siguiente:

Definición 3. Sea *M* el conjunto de mensajes del sistema y $m, m' \in M$. Entonces la relación de dependencia inmediata denotada como $\downarrow$ (IDR):

$$m \downarrow m' \Leftrightarrow [\, (m \rightarrow m') \wedge \forall\, m'' \in M, \neg\, (m \rightarrow m'' \rightarrow m')]$$

Así, un mensaje *m* directamente precede un mensaje *m'*, sí y sólo sí ningún otro mensaje perteneciente a *M* existe, tal que *m''* pertenece al mismo tiempo al futuro causal de *m,* y al pasado causal de *m'*.

Esta relación es importante debido a que muestra cómo el orden de entrega de los mensajes respeta el orden de su difusión para todos los pares de mensajes en IDR. Consecuentemente, mantiene la entrega causal para todos los mensajes.

Definición 4. Entrega causal *broadcast* usando relación de IDR:

$$\textbf{Sí } \forall\, m,m' \in M,\, m \downarrow m' \Rightarrow \forall k \in c : \textit{deliver}(\text{k},m) \rightarrow \textit{deliver}(k,m')$$
$$\textbf{entonces } m \rightarrow m' \Rightarrow \forall k \in c : \textit{deliver}(k,m) \rightarrow \textit{deliver}(k,m')$$

Información causal que incluye los mensajes que inmediatamente preceden a un mensaje dado es suficiente para asegurar una entrega causal de tal mensaje. Esta propiedad se ha mostrado en [POM04] (para la prueba formal consultar [POM02]).

En el escenario presentado en la figura 3.1a mostramos cómo la relación de dependencia inmediata une a cada mensaje, la necesaria y suficiente información de control para lograr un orden causal. Considere la difusión del mensaje m_5 ($send(m_5)$) tal que $m_1 \downarrow\, m_2 \downarrow\, m_3 \downarrow\, m_4 \downarrow\, m_5$.

La única información de control *CI* unida a m_2 con el objetivo de asegurar su entrega en orden causal corresponde a m_1, el cual es el único mensaje que tiene una relación de dependencia inmediata con m_2. El mensaje m_3, al ser concurrente con m_2, sólo necesita llevar unida información de control *CI* acerca de m_1 para ser entregado causalmente.
Continuando con la ejecución, el mensaje m_4 lleva unida información de control *CI* acerca de los mensajes m_2 y m_3, debido a que ambos poseen una IDR con m_4.

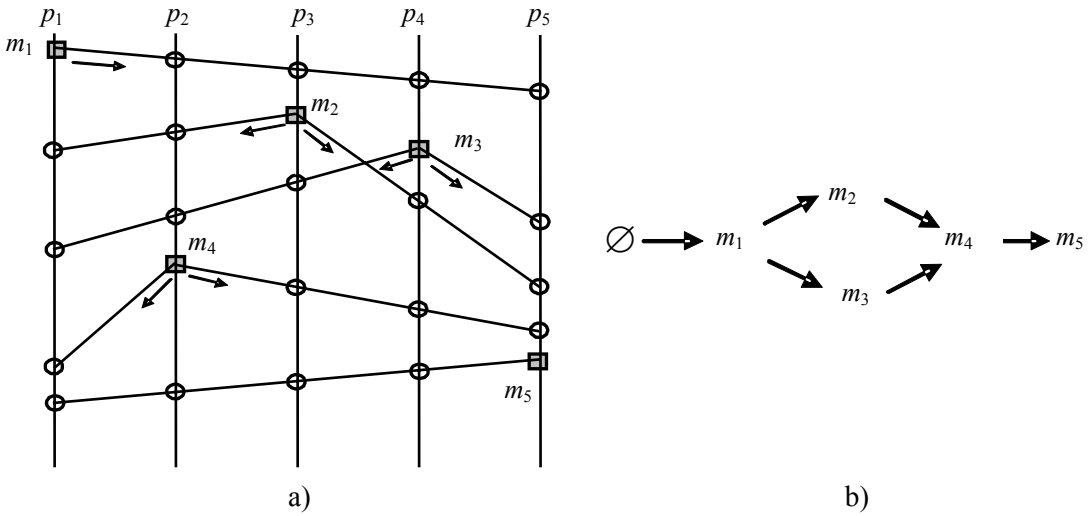

Figura 3.1. a) Escenario de un sistema b) Grafo de precedencia, p_1, p_2, p_3, p_4, p_5 representan los participantes o procesos que intercambian mensajes en el sistema, los círculos significan la entrega de los mensajes sobre los procesos y los cuadros el envío de los mensajes.

Por último, la única información de control unida a m_5 para que sea entregado en orden causal corresponde a m_4. En otras palabras, m_2 y m_3 serán entregados en orden causal sobre un proceso p_k después de que m_1 haya sido entregado (figura 3.1b). El mensaje m_4 será entregado una vez que m_2 y m_3 hayan sido entregados, y así sucesivamente. El orden casual $m_1 \rightarrow (m_2 \parallel m_3) \rightarrow m_4$ es mantenido a través de la relación de dependencia inmediata.

3.2 Algoritmo a orden causal con recuperación hacia adelante

Existen muchos trabajos concernientes a la entrega casual de mensajes en un sistema distribuido. La mayoría de estos algoritmos causales están diseñados para canales de comunicación fiables (no existen mensajes perdidos). Sin embargo, aplicaciones distribuidas como videoconferencias intercambian flujos de datos en tiempo real sobre canales de comunicación no fiables (algunos mensajes son perdidos). La pérdida de mensajes junto con la latencia son los dos principales problemas encontrados en modelos de red reales. Algunos trabajos resuelven el problema de la pérdida de mensajes utilizando la técnica de solicitud de repetición automática (*Automatic Repeat Request*, ARQ) o alguna variante. Una descripción más detallada de las propuestas que utilizan ARQ es proporcionada en el capítulo 2, sección 2. Aplicaciones en tiempo real no soportan la

retransmisión de mensajes perdidos debido a que son muy sensibles al retraso. Por otra parte, flujos de datos como audio y video toleran un cierto grado de mensajes perdidos, el cual es especificado en las restricciones de QoS del sistema.

En esta sección se propone un algoritmo causal para ser aplicado a redes de comunicación no fiables. El algoritmo causal desarrollado considera la posibilidad de mensajes perdidos durante su transmisión, la pérdida de información podría alterar la entrega causal de los mensajes transmitidos en el sistema. El algoritmo es capaz de restablecerse en forma descentralizada cuando se produce la pérdida de mensajes. Se extendió el algoritmo causal *broadcast* mínimo presentado en [POM02] para ser aplicado a canales de *comunicación no fiables*. El algoritmo mínimo está basado en la relación de dependencia inmediata (IDR). La IDR identifica la necesaria y suficiente información de control para ser unida a cada mensaje con el objetivo de asegurar el orden causal en *canales de comunicación fiables*.

Con la finalidad de soportar la pérdida de mensajes, se introdujo redundancia sobre la información de control unida por mensaje. En nuestro caso, la redundancia se define como el número de veces que la información de control *CI* acerca de un mensaje causal es enviada en el sistema. La redundancia está basada sobre la *distancia causal* entre mensajes. La *distancia causal* es el número más grande de mensajes causales que refleja el pasado causal entre dos mensajes. La distancia causal agrega información de control *CI extra* a cada mensaje transmitido con la finalidad de incrementar el grado de tolerancia a la pérdida de información. Con la IDR los mensajes envían información de control sólo acerca de su predecesor inmediato. La IDR indica una distancia causal de 1. Con una distancia casual más grande ($distancia\_causal \geq 2$) la información de control unida a cada mensaje es mayor. El objetivo de incrementar la distancia causal es unir a cada mensaje una mayor cantidad de información causal. El principal beneficio es incrementar el grado de tolerancia de mensajes perdidos. Formalmente se define como sigue:

Definición 5. Sean m y $m' \in M$ arbitrarios tal que $m \rightarrow m'$. Definimos la distancia causal entre m y m', la cual denotaremos por $d(m,m')$ como el entero n más grande tal que para alguna secuencia de mensajes (m_i, $i= 0,...,n$) con $m = m_0$ and $m' = m_n$, tenemos $m_i \downarrow m_{i+1}$ para todo $i = 0,...,n-1$.

Para ejemplificar el concepto de distancia causal, considere la difusión de los mensajes en la figura 3.1b con una distancia causal igual a 2. Ahora, la información de control *CI* unida a m_4 es acerca de los mensajes m_2 y m_3, con una distancia causal de 1, y de m_1 con una distancia causal de 2. EL mensaje m_5 lleva unida información de control acerca de m_4 con distancia causal de 1 y de m_2 y m_3 con distancia causal de 2. En la siguiente sección se presentará un estudio más detallado de la distancia causal en los casos de mensajes seriales y mensajes concurrentes.

Nuestro algoritmo es adecuado para ser usado en sistemas de tiempo real debido a que posee la característica de recuperación sin la retransmisión de mensajes perdidos. El trabajo presentado es, hasta donde sabemos, uno de los primeros algoritmos causales que utiliza un

mecanismo de recuperación causal de errores hacia adelante. El mecanismo de detección y recuperación causal hacia adelante de mensajes perdidos, descrito en este capítulo, representa la base del mecanismo de recuperación causal del algoritmo de sincronización de flujos continuos sobre *canales no fiables*, el cual se describe en el capítulo 4.

3.2.1 Modelo del sistema

La aplicación bajo consideración, para el algoritmo descrito en esta sección, está compuesta de un conjunto de procesos $P=\{p_1,p_2,....,p_n\}$ que se comunican sólo por el intercambio de mensajes. Los procesos están conectados a través de canales de comunicación con las siguientes características:

- La red es considerada no fiable, i.e. Los mensajes pueden ser perdidos durante su transmisión hacia su destino.

- Los mensajes difundidos pueden sufrir retrasos arbitrarios, pero finitos, durante su transmisión.

Cada participante funciona de forma independiente y realiza sus acciones con base en la información que ha recibido de los miembros del sistema.

3.2.2 Análisis de la relación entre la distancia causal y la información de control enviada

En el análisis realizado al comportamiento de la IDR sobre el algoritmo mínimo presentado en [POM02], encontramos que existe inevitable e inherente redundancia de información en el caso de la difusión de mensajes concurrentes. Considerando esta característica del algoritmo, introducimos redundancia extra cuando el número de mensajes concurrentes enviados es menor a la distancia casual establecida previamente. El concepto de distancia causal fue primero introducido para detectar la IDR [POM04], la cual presenta una distancia causal de uno.

Hay dos casos posibles en la transmisión de eventos causales: el caso serial y el caso concurrente.

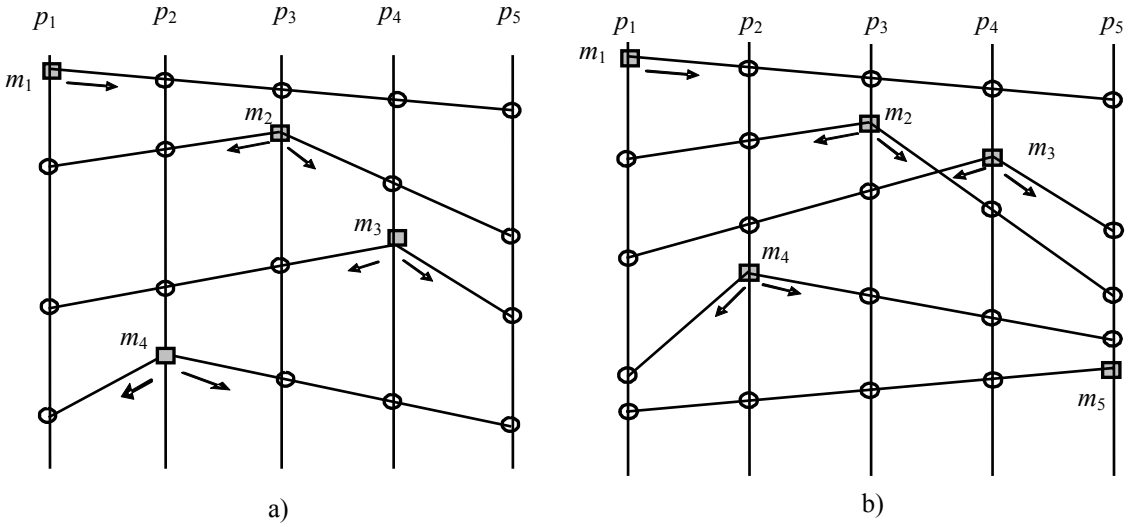

Figura 3.2. a) Diagrama de mensajes seriales, b) Diagrama de mensajes concurrentes

En el caso serial, la única información de control *CI* unidad a m_4, corresponde a m_3; ver figura 3.2a. El mensaje m_3 es el único que tiene una distancia causal de 1 con m_4. En este caso, la redundancia no existe en la información de control enviada, por lo tanto, es imposible recuperar el sistema cuando un mensaje es perdido.

En el caso de mensajes concurrentes (figura 3.2b), la redundancia es directamente proporcional al numero de mensajes concurrentes. En nuestro caso, la redundancia determina el número de veces que la información de control acerca de un mensaje causal es enviada en el sistema. Por ejemplo, usando una distancia causal de 1 (figura 3.2b), los mensajes concurrentes m_2 y m_3 envían información de control acerca de m_1, el cual tiene una distancia causal de 1 con ambos mensajes. Como podemos ver en este ejemplo, la información de m_1 es enviada dos veces. En este caso, si el mensaje m_1 es perdido durante su transmisión, el sistema es causalmente recuperado a través de la información proporcionada por cualquiera de los dos mensajes concurrentes recibidos. Con el objetivo de incrementar el grado de tolerancia a la pérdida de información, se introdujo redundancia extra en cada mensaje transmitido en el sistema.

Por ejemplo, en el caso serial usando una distancia causal igual a 2, el sistema puede recuperarse causalmente en la presencia de un mensaje perdido. Con una distancia igual a 1, los mensajes sólo envían información acerca de su predecesor inmediato. En la figura 3.2a, m_2 envía información sólo acerca de m_1, m_3 envía información sólo acerca de m_2. Por último, m_4 envía información sólo acerca de m_3.

Con una distancia causal de 2, la información de control enviada por mensaje corresponde a los mensajes que tengan una distancia causal igual o menor a 2 con el mensaje a enviar. Por ejemplo, para el caso serial (figura 3.2a), el mensaje m_3 debe enviar información de control (*CI*) acerca de los mensajes m_2 y m_1, con una distancia causal de 1 y 2 respectivamente. Por último, el mensaje m_4 ahora debe enviar información de control acerca de m_3 y m_2 con una distancia causal de 1 y 2 respectivamente.

En el caso de mensajes concurrentes con una distancia de 2 (figura 3.2b), el mensaje m_2 debe llevar unida información de control acerca de m_1. Debido a que el mensaje m_3 es concurrente al mensaje m_2 únicamente lleva unida información de control acerca de m_1. Continuando con la ejecución del escenario, el mensaje m_4 sólo lleva unida información de control acerca de m_2 y m_3, ambos con una distancia causal de 1. En este caso, el mensaje m_4 no lleva unida información de control acerca de m_1 (a pesar de que m_1 pose una distancia causal de 2 con respecto a m_4) debido a que la información de control de m_1 ha sido enviada dos veces en el sistema. Hacemos notar que el comportamiento del sistema determina de manera dinámica la información de control a enviar en cada mensaje, por lo tanto, el mecanismo sólo agrega información redundante cuando es necesaria.

3.2.3 Descripción del procedimiento de recuperación causal hacia adelante

Con la finalidad de explicar cómo la redundancia es aplicada para incrementar la tolerancia a la pérdida de información, se presenta el siguiente escenario (figura 3.3).

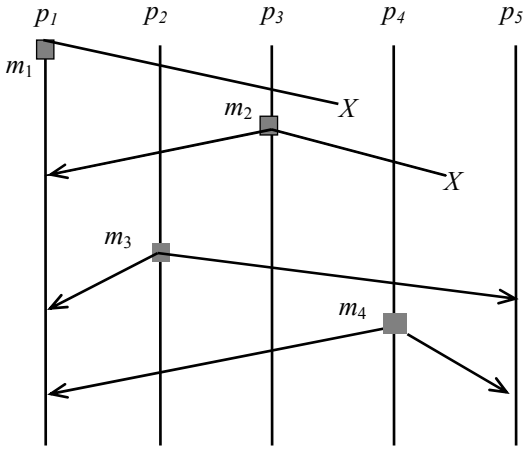

Figura 3.3. Escenario con pérdida de información

Considere la ejecución del escenario presentado en la figura 3.3 con una distancia causal de 2 ($d(m,m')= 2$). El mensaje m_1 es perdido durante su transmisión hacia los procesos p_4 y p_5. El proceso p_3, después de la recepción del mensaje m_1, difunde el mensaje m_2 con información de control acerca de m_1. El proceso p_4 recibe m_2 y a través de la información unida a m_2 es capaz de detectar que el mensaje m_1 ha sido perdido. Por lo tanto, p_4 procede

a actualizar la información que tiene con respecto a p_1 y entrega el mensaje m_2. Continuando con la ejecución del escenario, el proceso p_2 transmite el mensaje m_3. El mensaje m_3 lleva unida información causal de control acerca de los mensajes m_2 y m_1 con distancias causales de 1 y 2 respectivamente. El proceso p_5 no ha recibido los mensajes m_1 y m_2 debido a su pérdida durante su transmisión. Con la recepción del mensaje m_3, el proceso p_5 analiza la información de control unida a m_3 y puede determinar que dos mensajes han sido perdidos. Por lo tanto, p_5 procede a actualizar la información que posee acerca de los procesos p_1 y p_3 y entrega el mensaje m_3. Por último, el proceso p_4 envía el mensaje m_4 con información de control acerca de m_3 y m_2 con distancias causales de 1 y 2 respectivamente. Sobre la recepción del mensaje m_4 en el proceso p_5 se puede concluir que el orden causal $m_1 \rightarrow m_2 \rightarrow m_3 \rightarrow m_4$ ha sido preservado aún cuando se produce mensajes perdidos.

En esta primera parte de la extensión al algoritmo *broadcast* presentado en [POM04], por simplicidad, se considera que si un mensaje llega fuera de su orden o está retrasado, se determina que el mensaje ha sido perdido y es automáticamente descartado. En la sección 3.3 se plantea cómo el algoritmo es fácilmente adaptado para considerar un retraso máximo (*deadline*) asociado a cada mensaje.

3.2.4 Estructuras de datos del Algoritmo

Las estructuras de datos usadas en este algoritmo tienen algunos cambios en comparación con las estructuras usadas en [POM04]. La estructura *CI* fue modificada agregándole el campo de distancia causal *d*. Las estructuras usadas en el algoritmo son:

- El $VT(p)$ es el *vector de tiempo*. Para cada proceso p existe un elemento $VT(p)[j]$ donde j es un identificador de proceso. El tamaño de VT es igual al número de procesos en el grupo. $VT(p)$ contiene la vista local que el proceso p tiene de la historia causal de los mensajes difundidos del sistema. En particular, el elemento $VT(p)[j]$ representa el número de mensajes recibidos del identificador j. Tiene una vista total si, en un instante t, contiene la información del último mensaje transmitido por cada proceso. Tiene una vista parcial si contiene información de un subconjunto de los procesos. Es por medio de la estructura $VT(p)$ que se garantiza la entrega causal a nivel intervalo.

- La estructura de la información de control CI(p) es un conjunto de entradas (k, t, d). Cada elemento en CI(p) denota un mensaje que puede no ser entregado en orden causal por el participante p. La entrada (k, t, d) representa la difusión de un mensaje por parte del participante k en un tiempo local lógico $t = VT(p)[k]$, y d representa potencialmente la distancia causal.

- La estructura de un mensaje m es una cuadrupla $m = (i, t, message, H(m))$, donde:

  - i es el identificador del participante,

- $t=VT(p)[i]$ es el tiempo local lógico sobre el participante *i,*
- *message* es el mensaje en cuestión,
- $H(m)$ contiene un conjunto de entradas (k,t), las cuales representan mensajes que tienen una distancia causal menor o igual a la definida en el algoritmo.

- La variable *dist_def* es la distancia causal predeterminada, adaptable a las condiciones de pérdida de los canales de comunicación.

3.2.5 Especificación del Algoritmo

Algoritmo 1. Especificación de algoritmo *broadcast* con recuperación causal hacia adelante

1.	**Inicializacion**
2.	$VT(p)[j] = 0 \ \forall \ j{:}1\ldots n.$
3.	$CI(p) \leftarrow \varnothing$
4.	**For each diffusion of message** send(m) **at** p
5.	$VT(p)[i] = VT(p)[i] +1$
6.	For each $(k,t,d) \in CI(p)$
7.	$(k,t,d) \leftarrow (k,t,d+1)$
8.	$H(m) \leftarrow H(m) \cup (k,t)$
9.	endfor
10.	$m \equiv (i,\ t = VT(p)[i]$, datos, $H(m))$
11.	**Diffusion :** send(m) /* envio del mensaje m */
12.	$CI(p) \leftarrow CI(p) \cup (k,t,d=0)$
13.	For all $(k,t,d) \in CI(p)$ if $d = dist\_def$. Then
14.	$CI(p) \leftarrow CI(p) / (k,t,d)$
15.	endif
16.	endfor
17.	**For each reception** receive(m) **at** p
	$m \equiv (k, t,$ *message*, $H(m))$
	/ Para asegurar la entrega causal de m */*
	/ Condición de entrega */*
18.	if not ($t = VT(p)[k] +1$ and $\forall \ (l,x) \in$ H(m): $x \leq VT(p)[l]$) then
	/ Detención de mensajes perdidos y actualización de los vectores */*
19.	For all $(l,x) \in$ H(m)
20.	if ($x > VT(p)[l]$) then
21.	$VT(p)[l]=x$
22.	endif
23.	endif
	/* Entrega causal de mensajes */

24.	**Delivery**: delivery(m)
25.	$VT(p)[k] = VT(p)[k] +1$
26.	For all $(l,x) \in H(m)$ if $\exists d : (l,x,d) \in CI(p)$ then
27.	$(l,x,d) \leftarrow (l,x,d+1)$
28.	endif
29.	$CI(p) \leftarrow CI(p) \cup \{(k,t,d=0)\}$
30.	For all $(l,x,d) \in CI(p)$ if d=dist_def then
31.	$CI(p) \leftarrow CI(p) / (l,x,d)$.
32.	endif
33.	endfor

3.2.6 Ejemplo del funcionamiento del algoritmo

Considere el grupo de participantes $g = \{p_1, p_2, p_3, p_4, p_5\}$ y la difusión del mensaje m_4 al proceso p_1. Antes de la entrega de m_4 a p_1, el $CI_1 = \{(1,1,0)\}$ y el $VT_1(p_1)=(1,0,0,0,0)$ (figura 3.4). Estos valores son deducidos de la ejecución del algoritmo 1, para analizar la ejecución completa ver tabla 3.1.

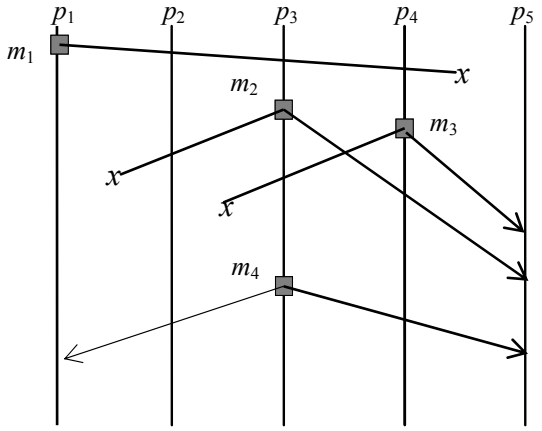

Figura 3.4. Escenario con mensajes perdidos

Difusión del mensaje m_4 por el proceso p_3.

Línea 5. El valor del vector de tiempo lógico $VT_3(p_3)[3]$ es incrementado en uno. El vector de tiempo lógico asegura la entrega secuencial de todos los mensajes m difundidos por el mismo participante p_k.

Líneas 6-9. Se incrementa el valor de la distancia causal d para cada tripleta de $CI_3=\{(3,1,1),(4,1,1)\}$. Esto es hecho con el objetivo de conocer cuántas veces los mensajes, que tienen un orden causal con respecto a m_4, han sido difundidos. En la **línea 8** se asigna al $H(m)$ sólo la dupla (k,t) de cada elemento de CI_3, $H(m)=\{(3,1),(4,1)$. Estas entradas en $H(m)$ son las responsables de la detección de mensajes perdidos, y también son utilizadas

para actualizar el vector de tiempo lógico con la finalidad de mantener un estado causal consistente.

Línea 11. Difusión del mensaje m_4=(3,2,*event*, {(3,1),(4,1)}), send(m_4).

Línea 12. El CI_3 es actualizado con información de control acerca del mensaje m_4, CI_3 ={(3,1,1),(4,1,1),(3,2,0)}.

Continuando con la ejecución del algoritmo, en la **línea 13** se busca una tripleta con distancia igual a 2 (d=2), si existe es borrada (línea 15) del CI_3. Cuando una tripleta en CI_k tiene una distancia causal d = 2, indica que la información de control de un mensaje ha sido transmitida en la red dos veces y no es necesaria enviarla otra vez. En este caso, CI_3 no tiene una tripleta con d =2, por lo tanto, ninguna acción es necesaria.

Entrega del mensaje m_4 a el proceso p_1.

Cada vez que un participante recibe un mensaje, el participante verifica que el mensaje satisfaga la condición de entrega causal. La condición de entrega causal garantiza que el mensaje sea entregado en un completo orden causal. En este caso, el mensaje m_4=(3,2,*event*,{(3,1),(4,1)}) no satisface la condición de entrega causal (**Línea 19**) debido a que t = 2 y $VT(p_1)[3]$=1, por lo tanto las condiciones t=$VT(p)[k]$ y $t \leq VT(p)[\underline{l}]$ no son verdaderas. Al detectar la pérdida de mensajes, el algoritmo ejecuta el procedimiento de recuperación (**líneas 21-23**) y procede a actualizar el vector de tiempo lógico de p_1. En este paso, el vector $VT(p_1)$ es actualizado con información causal contenida en $H(m)$, resultando en VT(p_1)=(1,0,1,1,0).

Línea 25. El mensaje m_4 es causalmente entregado y en la **línea 26** el vector de tiempo lógico es incrementado por uno en $VT(p_1)[3]$ dando como resultado $VT_1(p_1)$=(1,0,2,1,0).

El *CI* del proceso P_1 es actualizado en la siguiente forma.

Línea 27. Si (l,x) existe en $H(m)$ el cual corresponde a una tripleta (k,t,d) en CI_1 donde l = k y t = x, se incrementa el valor de d en uno. En este caso, el mensaje m_4 no tiene en su $H(m)$ alguna dupla que corresponda con un elemento en CI_1.

Línea 31. Se actualiza CI_1 agregando información de control de m_4, resultando en CI_1={(1,1,0),(3,2,0)}.

Línea 32-35. Por último, se verifica que no exista alguna tripleta con d=2 en CI_1. Si existe, se borra esta tripleta. En este ejemplo en el CI_1 no hay ninguna tripleta con d=2; por lo tanto ninguna acción es realizada.

En la siguiente sección se extenderá el algoritmo tolerante a fallas desarrollado en esta sección con el objetivo de incorporar restricciones de entrega en tiempo real entre los mensajes transmitidos en el sistema.

Tabla 3.1 Ejecución del algoritmo *broadcast* con recuperación causal hacia adelante

p_1	p_2	p_3	p_4	P_5
VT_1=(0,0,0,0,0) $CI_1 \leftarrow \varnothing$	VT_2=(0,0,0,0,0) $CI_2 \leftarrow \varnothing$	VT_3=(0,0,0,0,0,) $CI_3 \leftarrow \varnothing$	VT_4=(0,0,0,0,0) $CI_4 \leftarrow \varnothing$	VT_5=(0,0,0,0,0) $CI_5 \leftarrow \varnothing$
Diffusion(m_1) VT_1=(1,0,0,0,0) m_1=(1,1,event, $\varnothing$) send(m_1) CI_1={(1,1,0)}				
	reception(m_1) delivery(m_1) VT_2=(1,0,0,0,0) CI_2={(1,1,0)}	reception(m_1) delivery(m_1) VT_3=(1,0,0,0,0) CI_3={(1,1,0)}	reception(m_1) delivery(m_1) VT_3=(1,0,0,0,0) CI_3={(1,1,0)}	**lost message m_1**
		Diffusion(m_2) VT_3=(1,0,1,0,0) CI_3={(1,1,1)} $H(m)$={(1,1)} m_2=(3,1,event,(1,1)) send(m_2) CI_3={(1,1,1),(3,10)}	diffusion(m_3) VT_4=(1,0,0,1,0) CI_4={(1,1,1)} H(m)={(1,1)} m_2=(4,1,event,(1,1)) send(m_3) CI_4={(1,1,1),(4,10)}	
lost message m_2	reception(m_2) delivery((m_2) VT_2=(1,0,1,0,0) CI_2={(1,1,1),(3,1,0)}		reception(m_2) delivery(m_2) VT_4=(1,0,1,1,0) CI_4={(4,1,0),(3,1,0)}	reception(m_3) **/*Detection lost message m_1 and update of vectors)*/** VT_5=(1,0,0,0,0) **/*causal devlivery*/** VT_5=(1,0,0,1,0) CI_5={(4,1,0)}
	lost message m_3	reception(m_3) delivery(m_3) VT_3=(1,0,1,1,0) CI_3={(3,1,0),(4,1,0)}		reception(m_2) delivery(m_2) VT_5=(1,0,1,1,0) CI_5={(3,10),(4,1,0)}
lost message m_3		Diffusion(m_4) VT_3=(1,0,2,1,0) CI_3={(3,1,1),(4,1,1)} H(m)={(3,1),(4,1)} m_4=(3,2,event,{(3,1), (4,1)} send(m_4) CI_3={(3,1,1),(4,1,1), (3,2,0)}		
reception(m_4) **/*Detection lost message m_2 and m_3 and update of vectors)*/** VT_1=(1,0,1,1,0) **/*causal devlivery*/** VT_1=(1,0,2,1,0) CI_1={(1,1,0),(3,2,0)}	reception(m_4) **/*Detection lost message m_3 and update of vectors)*/** VT_2=(1,0,1,1,0) **/*causal devlivery*/** VT_2=(1,0,2,1,0) CI_2={(1,1,1),(3,1,1) (3,2,0)}		reception(m_4) delivery(m_4) VT_4=(1,0,2,1,0) CI_4={(4,1,1),(3,1,1), (3,2,0)}	reception(m_4) delivery(m_4) VT_5=(1,0,2,1,0) CI_5={(3,1,1),(4,1,1)} (3,2,0)}

3.3 Orden causal con restricciones de entrega en tiempo real

Aplicaciones en tiempo real, tales como teleconferencias, ambientes virtuales, requieren estrictas restricciones de tiempo sobre la entrega de los datos. Por ejemplo, muchas aplicaciones toleran un retraso máximo de 200ms (*tiempo de vida*, Δ) o menos entre los mensajes transmitidos por sus participantes [Gau99]. Mensajes que sufren un retraso mayor al tolerado por la aplicación son considerados como no útiles y son descartados. Preservar las restricciones de tiempo en la entrega de las unidades de datos transmitidas es fundamental para la interactividad de la aplicación en cuestión.

Diversos algoritmos causales con restricciones de entrega en tiempo real han sido desarrollados [BAL94, ADE95, PER03]. El método utilizado por estos trabajos para mantener las relaciones temporales entre los mensajes es el siguiente: se establecen una referencia global en sistema a través de un protocolo de sincronización de relojes físicos. Los participantes emplean esta referencia global para determinar si un mensaje es útil a la aplicación. Al tratar de establecer un reloj global, implícitamente se agrega un error de precisión. El error de precisión incrementa en sistemas distribuidos dispersos geográficamente. Por lo tanto, establecer una referencia global se vuelve un problema difícil a resolver. La figura 3.5 representa el mecanismo que utiliza un reloj global para calcular el tiempo de vida de los mensajes [BAL94].

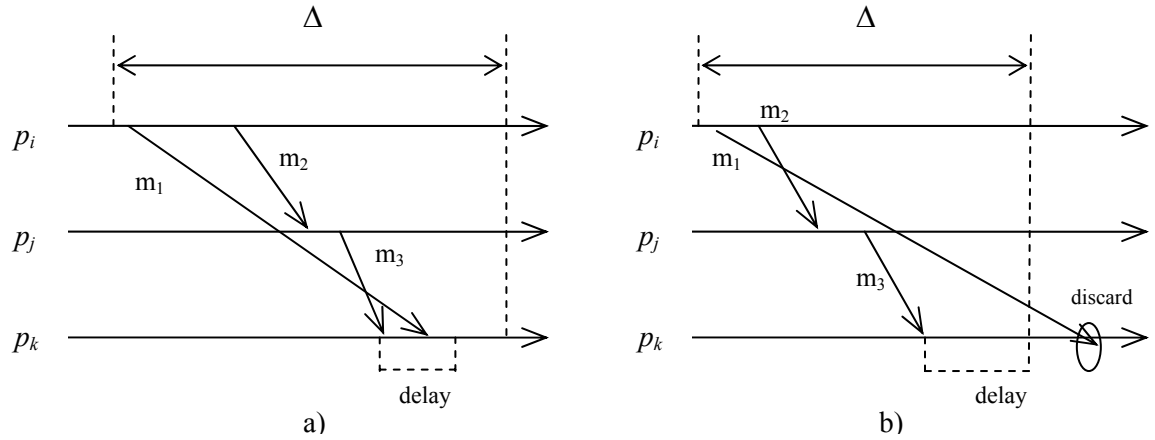

Figura 3.5. Cálculo del tiempo de vida de un mensaje asumiendo un reloj global

En la figura 3.5a, el mensaje m_1 sufre un retraso mayor durante su transmisión que el mensaje m_3. El mensaje m_3 es puesto en espera hasta que m_1 es recibido debido a que m_1 precede causalmente a m_3. En este caso, m_1 fue recibido antes de que su tiempo de vida se venciera y es entregado, posteriormente el mensaje m_3 es entregado. En la figura 3.5b, El mensaje m_1 sufre un retraso mayor a su tiempo de vida y es descartado cuando es recibido. En el ejemplo mostrado por la figura 3.5, la referencia de tiempo usada es global para los 3 participantes.

Nuestro enfoque

Por otra parte, nuestro trabajo propone un mecanismo original distribuido para detectar las relaciones temporales entre mensajes, evitando la sincronización de relojes físicos de cada participante con alguna fuente externa o interna al sistema. Cada participante emplea su reloj local físico de manera autónoma para determinar si un mensaje es útil a la aplicación o debe ser descartado. Este enfoque lo hemos denominado método distribuido para el cálculo del tiempo de vida.

El periodo de tiempo que una aplicación considera útil los datos transportados por un mensaje es llamado *tiempo de vida* (*timelife*, Δ). El *tiempo de vida*, Δ, de un mensaje depende de la información transmitida y de la aplicación en cuestión. Por ejemplo, el tiempo de vida para mensajes de video en una videoconferencia es de 100ms. En nuestro trabajo, el *tiempo limite* (*deadline*) para algún mensaje *m* se calcula con base en el tiempo de entrega de su predecesor inmediato *m'*, el cual tiene una distancia causal de 1 con *m* ($d(m,m') = 1$), más el tiempo de vida de *m* (*deadline = tiempo de entrega* de *m' + tiempo de vida* de *m*) . Un mensaje que es recibido sobre su destino después de su *deadline* es descartado. Por motivos de simplicidad, asumimos un único tiempo de vida para todos los mensajes (en caso de mensajes con diferentes tiempos de vida, podemos asumir el mínimo entre los mensajes como único tiempo de vida para ser respetado por todos los mensajes).

En la siguiente sección se describirá más a detalle el método distribuido para el cálculo del tiempo de vida propuesto en este trabajo. Comenzando con la descripción del mecanismo utilizado para preservar las relaciones temporales en eventos seriales y continuando con los eventos transmitidos concurrentemente.

3.3.1 Modelo del sistema

El algoritmo presentado en esta sección asume las siguientes características en los canales de comunicación:

- La red es considerada no fiable. Los mensajes pueden ser perdidos durante su transmisión hacia su destino.

- Los mensajes difundidos pueden sufrir retrasos arbitrarios e impredecibles durante su transmisión.

- Restricciones de entrega en tiempo real son consideradas entre los mensajes transmitidos en el sistema. Existe un retraso máximo (*tiempo de vida*) que un mensaje puede sufrir sin degradar la calidad de servicio de la aplicación. Cuando un mensaje es recibido con un retraso mayor a su *tiempo de vida* se considera no útil y es descartado.

Dos características importantes del modelo contemplado son la no suposición acerca de la sincronización de relojes físicos entre los miembros del sistema, ni la existencia de memoria compartida. Cada participante funciona de forma independiente y realiza sus acciones con base en la información que ha recibido de los miembros del sistema.

3.3.2 Método distribuido para el cálculo del tiempo de vida

El principio del *método distribuido para el cálculo del tiempo de vida* se basa en la vista parcial que un participante tiene en un sistema distribuido. Cada participante utiliza su reloj local físico, de forma independiente, para preservar las restricciones de entrega en tiempo real de los mensajes difundidos en el sistema. Si un mensaje es recibido después de su *deadline* se considera no útil a la aplicación y es descartado.

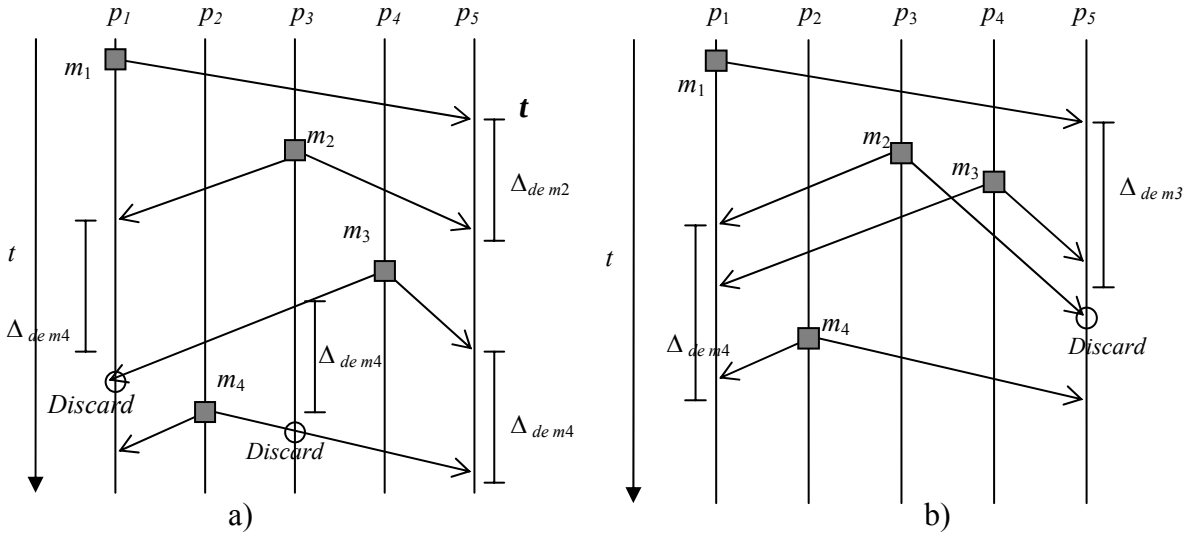

Figura 3.6. Método distribuido en a) eventos causales seriales y b) eventos causales concurrentes.

A continuación se describirá el método distribuido para el cálculo del tiempo de vida aplicado a los dos posibles casos en la transmisión de eventos causales. En el primer caso, los eventos causales son transmitidos en forma serial (figura 3.6a). Los eventos seriales tienen la característica de tener una distancia causal de 1 entre ellos. El *deadline* para un mensaje serial se calcula con base en su predecesor inmediato, el cual tiene una distancia causal de 1 con el mensaje recibido.

Los procesos entregan los mensajes sobre un tiempo *t* denominado *tiempo de entrega*. El *tiempo de entrega* es obtenido del reloj local físico del participante p_i. En la figura 3.6a, el mensaje m_1 es entregado sobre el proceso p_5 en un tiempo *t* (*tiempo de entrega*). El proceso p_5 calcula el *deadline* para algún mensaje m_i, el cual tiene una distancia causal igual a 1 con respecto a m_1 ($d(m_1,m_i) = 1$). En este ejemplo, el mensaje m_2 tiene una distancia causal de 1 con m_1. El *deadline* de m_2, sobre el proceso p_5, es igual a la suma del *tiempo de entrega t* de m_1 más el *tiempo de vida* (Δ) de m_2. El mensaje m_2 es recibido adentro de su *deadline* y es

entregado. Un mensaje que llega sobre su destino después de su *deadline* es descartado para evitar una degradación mayor a la aplicación. Los mensajes m_3 y m_4 son descartados, sobre los procesos p_1 y p_3 respectivamente, debido a que han sido recibidos después de su *deadline* (figura 3.6a).

En el caso de eventos causales concurrentes (figura 3.6b) no existe relación de orden causal. En consecuencia, no se pueden establecer restricciones de entrega en tiempo real entre eventos concurrentes. En este caso, los mensajes m_2 y m_3 no tienen restricciones de entrega en tiempo real entre ellos, debido a que son mensaje concurrentes (figura 3.6b). Un proceso p_k puede recibir primero m_3 y entregar después m_2 o viceversa. Los mensajes m_2 y m_3 tienen una distancia causal de 1 con respecto al mensaje m_1. Por lo tanto, el *deadline* para los mensajes m_2 y m_3 es calculado con base en el *tiempo de entrega t* de m_1 más el Δ. Sobre el proceso p_5, el mensaje m_3 es recibido adentro de su *deadline* y es entregado a la aplicación.

Por otra parte, el mensaje m_2 es recibido después de su *deadline* y es descartado. En figura 6b, el mensaje m_4 tiene una distancia causal de 1 con respecto a los mensajes m_2 y m_3. Con el objetivo de lograr una mayor precisión y mejorar la calidad de servicio, sobre el proceso p_1, el *deadline* de m_4 es calculado con base en el *tiempo de entrega* de m_2 más el Δ de m_4. Si se calculara el *dealdine* de m_4 con el *tiempo de entrega* de m_3, el *deadline* de m_4 sería mayor que el retraso máximo que puede tolerar m_2 con respecto a m_4. Por cuestiones de simplicidad, nosotros consideramos que los mensajes causales enviados en el sistema tienen el mismo *tiempo de vida*, después del cual son considerados no útiles para la aplicación en cuestión.

3.3.3 Algoritmo causal tolerante a fallas con restricciones de entrega

En esta sección extendemos el algoritmo *broadcast* con recuperación causal hacia delante, descrito en la sección 3.2, incorporando restricciones de entrega en tiempo real entre los mensajes difundidos en el sistema. Las restricciones de entrega en tiempo real son preservadas mediante el método distribuido para el cálculo del tiempo de vida descrito en la sección 3.3.1. Con la finalidad de explicar el comportamiento del algoritmo en condiciones de redes reales, como pérdida y retrasos arbitrarios de mensajes, se examinará el siguiente escenario.

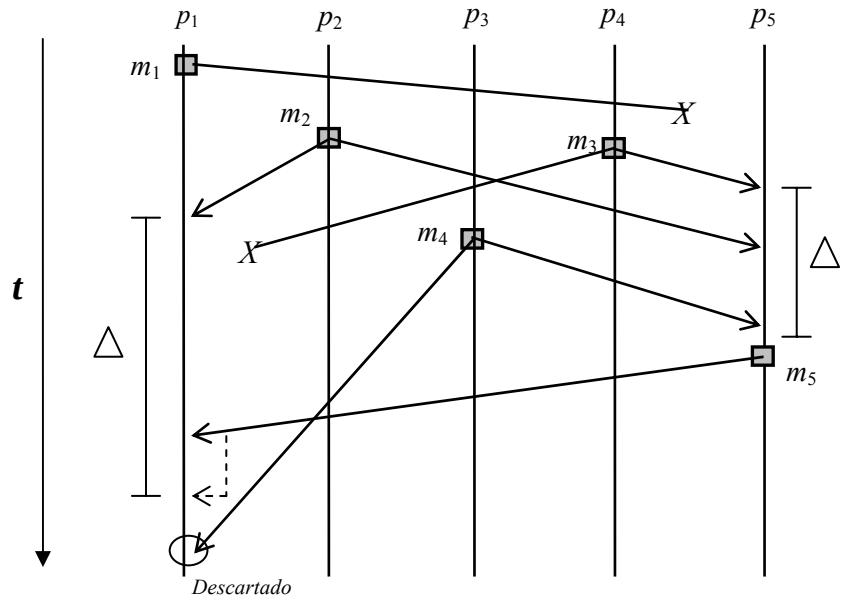

Figura 3.7. Escenario con pérdida de mensaje y restricciones de entrega en tiempo real

Considere el escenario presentado en la figura 3.7, con una distancia causal igual a 2 (d=2). El tiempo de vida, Δ, de cada mensaje depende del tipo de información transmitida en el sistema. Por ejemplo, para mensajes que transportan audio en una videoconferencia es de 100ms [ADEL95]. Por cuestiones de simplicidad, consideramos que el tiempo de vida es igual para todos los mensajes.

La ejecución del escenario empieza con la difusión del mensaje m_1, el cual es recibido por todos los participantes o procesos a excepción del participante p_5. Los participantes p_2 y p_4, después de la recepción de m_1, envían los mensajes m_2 y m_3 respectivamente. Ambos mensajes llevan información de control acerca de m_1. El participante p_5 recibe m_3 y a través de la información de control unida a m_3 es capaz de detectar que el mensaje m_1 ha sido perdido, p_5 procede a actualizar la información que tiene acerca del participante p_1 y entrega el mensaje m_3. Continuando con la ejecución del escenario, el participante p_5 calcula el *deadline* para algún mensaje m_i con una relación de dependencia inmediata con m_3. En este caso, el mensaje m_4 tiene una IDR con respecto a m_3. El deadline para m_4, sobre el participante p_5, es igual al *tiempo de entrega* de m_3 más el *tiempo de vida*, Δ, de m_4. El mensaje m_4 es recibido sobre p_5 dentro de su *deadline* y es entregado. Antes de la recepción de m_4, el participante p_5 recibe m_2, el cual es concurrente a m_3, debido a que m_2 y m_3 son concurrentes no existe restricción de entrega en tiempo real entre ellos, y por lo tanto, m_2 es entregado.

Por otra parte, el mensaje m_3 es perdido durante su transmisión hacia p_1. Después de la recepción del mensaje m_4, el participante p_5 difunde el mensaje causal m_5. El participante p_1 recibe m_5 antes que m_4. El mensaje m_5 no puede ser entregado debido a que el mensaje m_4 lo precede causalmente ($m_4 \rightarrow m_5$). p_1 retrasa la entrega de m_5 hasta el vencimiento del

deadline de m_4 con la finalidad de recibir en este tiempo el mensaje m_4. En este caso, el mensaje m_4 no es recibido dentro de su deadline. Al expirar el deadline de m_4 se considera perdido y m_5 es entregado. El mensaje m_5 lleva información de control acerca de los mensajes m_4 (con una distancia casual igual a 1) y de los mensajes m_2 y m_3 (con una distancia causal igual a 2). p_1 detecta la pérdida del mensaje m_3 con la información de control unida a m_5 y actualiza la información que posee acerca de p_4. Por último, el mensaje m_4 es recibido fuera de su deadline y es descartado.

El orden casual $m_1 \rightarrow m_2 \rightarrow m_3 \rightarrow m_4 \rightarrow m_5$ es logrado por algoritmo casual aún cuando se produce la pérdida de mensajes. Las restricciones de entrega en tiempo real entre los mensajes son verificadas a través del método distribuido para el cálculo del tiempo de vida.

3.3.3.1 Estructuras de datos del algoritmo

En esta sección, se describen las estructuras de datos que el algoritmo utiliza para lograr el orden casual en canales de comunicación no fiables (pérdida de mensajes) y asíncronos (los retrasos de los mensajes son aleatorios e impredecibles).

- El $VT(p)$ es el *vector de tiempo*. Para cada proceso p existe un elemento $VT(p)[j]$ donde j es un identificador de proceso. El tamaño de VT es igual al número de procesos en el grupo. $VT(p)$ contiene la vista local que el proceso p tiene de la historia causal de los mensajes difundidos del sistema. En particular, el elemento $VT(p)[j]$ representa el número de mensajes recibidos del identificador j. Tiene una vista total si, en un instante t, contiene la información del último mensaje transmitido por cada proceso. Tiene una vista parcial si contiene información de un subconjunto de los procesos. Es por medio de la estructura $VT(p)$ que se garantiza la entrega causal.

- La estructura de la información de control CI(p) es un conjunto de entradas (k, t, d). Cada elemento en CI(p) denota un mensaje que puede no ser entregado en orden causal por el participante p. La entrada (k, t, d) representa la difusión de un mensaje por parte del participante k en un tiempo local lógico $t = VT(p)[k]$, y d representa potencialmente la distancia causal.

- La estructura de un mensaje m es una cuadrupla $m = (i, t, message, H(m))$, donde:

  - i es el identificador del participante,

  - $t=VT(p)[i]$ es el tiempo local lógico sobre el participante i,

  - *message* es el mensaje en cuestión,

  - $H(m)$ contiene un conjunto de entradas (k,t), las cuales representan mensajes que tienen una distancia causal menor o igual a la definida en el algoritmo.

- La variable *dist_def* es la distancia causal predetermina.
- La estructura *queue* es un conjunto de mensajes. Cada elemento en *queue* tiene asignado un determinado *tiempo de espera*. Cuando el *tiempo de espera* de un mensaje expira, el mensaje debe ser entregado a la aplicación. Se debe actualizar el vector lógico y la estructura CI.

- *last_rec* contiene el último mensaje recibido en el participante p_i. La estructura last_rec es utilizada para detectar si un mensaje recibido es paralelo al último mensaje entregado.

- La variable *serial_deadline* representa el deadline para mensajes seriales.

- La variable *parallel_deadline* representa el deadline para mensajes en paralelo.

3.3.3.2 Especificación del algoritmo

El algoritmo presentado en la tabla 6 es la unión del algoritmo *FEC causal* con el principio del *método distribuido para el cálculo del tiempo de vida*. Preserva el orden causal entre los mensajes difundidos por los participantes en condiciones reales de red. El algoritmo es capaz de recuperarse hacia adelante de mensajes perdidos. Mantiene las restricciones de entrega en tiempo real de los mensajes transmitidos utilizando el método distribuido para el cálculo del tiempo de vida desarrollado en la tesis. Es adecuado para aplicaciones en tiempo real debido a que evita la retransmisión de la información pérdida.

Algoritmo 2. Mecanismo tolerante a fallas con restricciones de entrega en tiempo real

1.	**Initially**
2.	$VT(p)[j] = 0 \ \forall \ j{:}1 \ldots n.$
3.	$CI(p) \leftarrow \varnothing$
4.	*queue* $\leftarrow \varnothing$
5.	*last_rec* $\leftarrow \varnothing$ */* Estructura que lacena el ultimo mensaje recibido */*
6.	*deadline = big_value*
7.	*serial_deadline* = 0 */* tiempo para mensajes en serie */*
8.	*parallel_deadline* = 0 */* tiempo para mensajes en paralelo */*
9.	dist_def = input_var */* adaptable a las condiciones de la red */*
10.	**For each diffusion of message** send(m) **at** p
11.	$VT(p)[i] = VT(p)[i] + 1$ */* vector de tiempo logico */*
12.	For each $(k,t,d) \in CI(p)$ */* construccion de H(m) */*
13.	$(k,t,d) \leftarrow (k,t,d+1)$
14.	$H(m) \leftarrow H(m) \cup (k,t)$
15.	Endfor
16.	$m \equiv (i,\ t = VT(p)[i],$ *message,* $\mathrm{H}(m))$
17.	**Diffusion** send(m)

```
18.  CI(p)←CI(p) ∪ {(k,t,d=0)}   /* agregando informacion de contro a enviar*/
19.  if ∃ (k,t,d)  ∈ CI(p)  | d = dist_def. Then
20.       CI(p)←CI(p) / (k,t,d)     /* Borrando elementos ein CI con d = distancia def. */
21.  endif
22.  Update( )    /* actualizacion de tiempo físico */
23.  last_rec ← (i, t)
24. endfor.

25. For each reception receive(m) at p
26.  m ≡ (k,t,message, H(m))
27.  /* Condicion de entrega causal de  m */
28.  if  (( t = VT(p)[k] +1) or ( t  >  VT(p)[k] +1)) then
29.    if  not ( x ≤ VT(p)[l]∀ (l,x)∈ H(m)) then
30.        if (time_arrive_msg > deadline ) then  /*Si el mensajes esutil o no*/
31.            /*  Deteccion de mesajes perdidos*/
32.            ∀ (l,x) ∈ H(m) if ∃ (x>VT(p)[l]) then
33.              VT(p)[l]=x
34.            Endif
35.            Deliver(m)    /* entrega del mensaje recibido*/
36.            Act_vec_CI ( ) /*Actualizacion del CI */
37.          Else
38.            Put_message_in_queue(m, deadline)  /*mensajes encolados*/
39.        Endif
40.      Else
41.        ∀ (l,x) ∈ H(m)  if /∃ (y,z) ∈ last_rec | l = y,x = z then
42.            serial_deadline = deadline  /* Checar cuando el mensaje es  */
43.            deadline = parallel_deadline      /* el último mensaje recibido  */
44.        Endif
45.        If (time_arrive_msg > deadline ) then  /* Verificando el deadline*/
46.            Discard (m)
47.            deadline =  deadline + Δ
48.            VT(p)[k] = VT(p)[k] +1
49.          Else
50.            Deliver(m)    /* Enregando el mensaje recibido*/
51.            Act_vec_CI ( ) /*Actualiza el CI con la informacion recibida */
52.        Endif
53.    Endif  /* end of  if  not ( x ≤ VT(p)[l] ∀ (l,x) ∈ H(m)) then */
54.   Else
55.    Discard (m)  /*Descartando el mensaje*/
56.  Endif  /* end of  if  (( t = VT(p)[k] +1) or ( t  >  VT(p)[k] +1)) then*/
57. Endfor
```

```
/* construction of m' = (k, t, message, H(m), limit_time)  */
Put_message_in_queue(m, limit_time){   /* pone un menssage en espera*/
      If ( limit_time = big_value ) then
            limit_time = current_time + Δ
            m' = m ∪ (limit_time )
        Else
            m' = m ∪ (limit_time)
            queue = queue ∪ m'
      endif
}
/*  m=(k,t,message, H(m), time )  */
check_timer {    /* Verificando si el deadline del mensaje entregado ha expirado */
      ∀m ∈ queue
            if ∃ limit  time ∈ m | current  time > limit  time then
                  if ( x > VT(p)[l] ∀ (l,x) ∈ H(m))
                        VT(p)[l]=x
                  Endif
            Act_vec_CI ( )
            Endif
}
Act_vec_CI ( ){    /* update the CI*/
      VT(p)[k] = VT(p)[k] +1
      ∀ (l,x) ∈ H(m)  if ∃ (k,t,d) ∈ CI(p) | k=l,t=x
            (k,t,d) ←(k,t,d+1)      /*incrementando en 1 la distancia causal*/
      Endif.
      CI(p)←CI(p) ∪ {(k,t,d=0)}
      ∀ CI(p) if ∃ (k,t,d)  ∈ CI(p)  | d=dist_def then
            CI(p)←CI(p) / (k,t,d).
      Endif
      Update( )
      last_rec = (k,t)
}
Update( ) {
      parallel_deadline = deadline
      if ( last_rec = = ∅ ) then
            deadline = current_time + Δ
        Else
            serial_deadline = deliver_time m + Δ
            deadline= serial_deadline
      Endif
}
```

3.3.3.3 Ejemplo del funcionamiento del algoritmo

En el escenario presentado en la figura 3.8 mostramos el funcionamiento del algoritmo. Los participantes o procesos transmiten eventos causales con restricciones de entrega en tiempo real. En este escenario son aplicados los principios de recuperación causal hacia delante para tolerar la pérdida de mensajes. Para preservar las relaciones temporales entre mensajes se utiliza el enfoque del método distribuido para el cálculo del tiempo de vida.

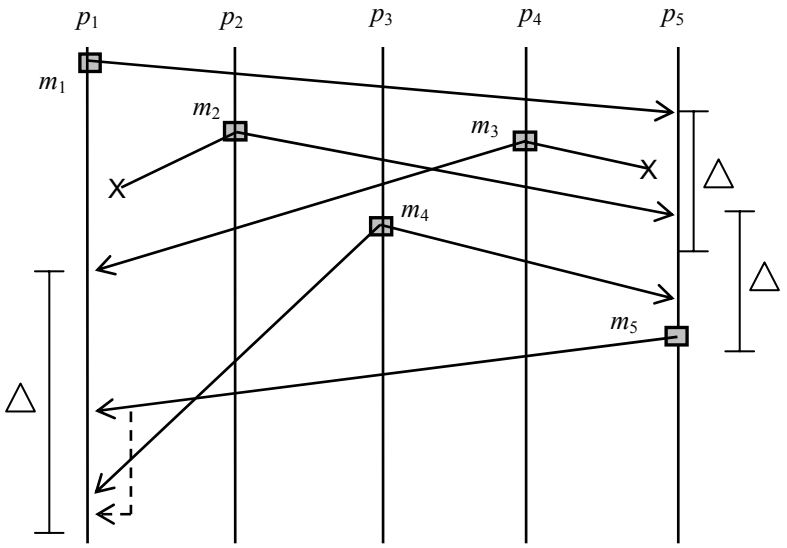

Figura 3.8. Escenario considerando pérdida y retraso aleatorio de mensajes.

Antes de la entrega de m_1 a p_5, los valores para sus variables y estructuras son $CI_5 = \varnothing$, VT_5= (0,0,0,0,0), *parallel_deadline* = 0, *deadline* = *big_value*, *dist_def* = 2 (tabla 3.2). Estos valores representan la parte de inicialización del algoritmo 1 (**Líneas 1-9**). El valor expresado por *big_value* indica que el participante no ha recibido ni enviado un mensaje en el sistema.

Entrega del mensaje m_1 sobre p_5.

Cada vez que un participante recibe un mensaje verifica si el mensaje satisface la condición de entrega FIFO y la condición de entrega causal establecida por el emisor del mensaje. En este caso el mensaje m_1 ={1,1,event, $\varnothing$} satisface la condición de entrega FIFO $t = VT(p_5)[k]+1$ debido a que t=1 y $VT(p_5)[1]$=0 (**Línea 28**).

Línea 29. Ahora se verifica si el mensaje m_1 satisface la condición de entrega causal. Como m_1 tiene su historial causal $H(m)$ igual a nulo no necesita la entrega de otro mensaje para ser entregado. El orden FIFO y causal es cumplido por m_1. Con la finalidad de mantener las restricciones de entrega en tiempo real, el siguiente paso es comprobar si m_1 ha sido recibido antes del vencimiento de su deadline.

Línea 45. El mensaje m_1 satisface las restricciones de entrega en tiempo real debido a que es el primer mensaje difundido en el sistema y no tiene relación temporal con ningún otro mensaje. El mensaje m_1 es entregado (**Línea 50).**

Línea 51. Actualizamos las estructuras VT_5 y CI_5 con la información contenida en m_1 llamando a la función Act_vec_CI( ) (**Líneas 78-89**), obteniendo los siguientes resultados CI_5= {(1,1,0)}, VT_5 = (1,0,0,0,0).

Línea 87. Calculamos el *deadline* para el siguiente mensaje que tenga como predecesor inmediato a m_1 con la llamada de la función *measure_deadline()* (**Líneas 90-98**). La variable *parallel_deadline* toma el valor de *big_value* (Línea 91). *Parallel_deadline* representa el *deadline* para mensajes en paralelo con m_1. El *deadline* para mensajes con una distancia causal de 1 con m_1 (seriales con respecto a m_1) es el valor representado por la variable de tiempo *serial_deadline*. Por ser m_1 el primer mensaje recibido la variable *serial_deadline* es igual al tiempo de entrega de m_1 (*deliver_time*) más el tiempo de vida (*timelife*) de los mensajes transmitidos Δ **(**Líneas 92-93**).**

Entrega de m_2 sobre p_5

El mensaje m_2 = (2,1,event,{(1,1)}) es recibido en un tiempo *t* (*time_arrive_msg*) sobre $\boldsymbol{p_5}$. m_2 satisface la condición de entrega FIFO y la condición causal (**Líneas 28-29**). Ahora verificamos si m_2 es un mensaje serial o paralelo al último mensaje recibido en p_5.

Línea 41. Si existe algún (*l*,*x*) en *H*(*m*) el cual corresponda a una dupla (*y*, *z*) sobre *last_rec* donde *l*=*y* y *x*=*z*, el mensaje m_2 es serial al último mensaje entregado, en caso contrario es considerado paralelo. En este caso, el mensaje m_2 es serial a m_1 y la variable *deadline* conserva el valor expresado por *serial_deadline*. Comparamos si m_2 es recibido antes de su *deadline* (**Línea 45**). En la figura 3.6 observamos que m_2 es recibido antes de su *deadline*, por lo tanto, es entregado causalmente (**Línea 50**).

Línea 51. Actualizamos las estructuras VT_5 y CI_5 con la información contenida en m_2, obteniendo los siguientes datos CI_5= {(1,1,1), (2,1,0)}, VT_5 = (1,1,0,0,0).

Línea 87. Calculamos el *deadline* para el siguiente mensaje que tenga como predecesor inmediato a m_2 (**Líneas 90-98**). Ahora *parallel_deadline* toma el valor de la variable *deadline* (Línea 91). *Parallel_deadline* representa el *deadline* para mensajes en paralelo con m_2. El *deadline* para mensajes con una distancia causal igual a 1 con m_2 es el valor representado por *serial_deadline*. El nuevo valor para la variable *serial_deadline* es igual al tiempo de entrega de m_2 (*deliver_time* m_2) más el tiempo de vida (Δ) de los mensajes transmitidos (**Líneas 95-96**).

Entrega del mensaje m_5 sobre p_1

El mensaje m_5 = (5,1,event,{(1,1),(2.,1),(3,1)}) es recibido sobre $\boldsymbol{p_5}$ antes de la recepción de m_4**,** la condición entrega causal (**Línea 29**) no se cumple debido a que m_4 precede a m_5. En la **línea 30** analizamos si el *deadline* para m_4 ha expirado. Cuando el deadline de un mensaje expira consideramos que el mensaje se ha perdido y se realiza el procedimiento de detección y recuperación hacia adelante descrito en la sección 3.3.3 (**Líneas 32-34**).

En el escenario presentado en la figura 6, el deadline de m_4 no ha expirado, por lo tanto, el mensaje m_5 es puesto en espera (**Línea 38**). Dos posibles acciones pueden sacar un mensaje de la cola de espera. Una es la entrega de los mensajes que preceden causalmente al mensaje encolado. La segunda es la expiración del *deadline* de los mensajes que preceden causalmente al mensaje puesto en espera. En este caso ocurre el primer escenario, el mensaje m_4 es recibido antes del vencimiento de su *deadline* y es entregado. Ahora el mensaje m_5 satisface la condición de entrega causal y puede ser entregado. En la tabla 3.2, describimos la ejecución completa del mecanismo especificado en el algoritmo 3. El algoritmo es capaz de recuperarse hacia adelante de mensajes perdidos, evitando la retransmisión. Mostramos como el principio del método distribuido para el cálculo del tiempo de vida preserva las restricciones de entrega en tiempo real entre los mensajes, sin utilizar una referencia global. En la tabla 3.2, se indica cómo las variables y estructuras van cambiando su valor para mantener el orden causal aun en condiciones de pérdida y retrasos de mensajes.

Tabla 3.2. Ejecución del algoritmo 3 del escenario presentado en la figura 3.8.

p_1	p_2	p_3	p_4	p_5
INICIALIZACION VT_1=(0,0,0,0,0) $CI_1 \leftarrow \varnothing$ $queue_1 \leftarrow \varnothing$ $last\_rec_1 \leftarrow \varnothing$ time = big_value $serial\_time_1 = 0$ $parallel\_time_1$=big_value $dist\_def_1 = 2$	**INICIALIZACION** VT_2=(0,0,0,0,0) $CI_2 \leftarrow \varnothing$ $queue_2 \leftarrow \varnothing$ $last\_rec_2 \leftarrow \varnothing$ $time_2$ = big_value $serial\_time_2 = 0$ $parallel\_time_2$ =big_value $dist\_def_2 = 2$	**INICIALIZACION** VT_3=(0,0,0,0,0,) $CI_3 \leftarrow \varnothing$ $queue_3 \leftarrow \varnothing$ $last\_rec_3 \leftarrow \varnothing$ $time_3$ = big_value $serial\_time_3 = 0$ $parallel\_time_3$=big_value $dist\_def_3 = 2$	**INICIALIZACION** VT_4=(0,0,0,0,0) $CI_4 \leftarrow \varnothing$ $queue_4 \leftarrow \varnothing$ $last\_rec_4 \leftarrow \varnothing$ $time_4$ = big_value $serial\_time_4 = 0$ $parallel\_time_4$=big_value $dist\_def_4 = 2$	**INICIALIZACION** VT_5=(0,0,0,0,0) $CI_5 \leftarrow \varnothing$ $queue_5 \leftarrow \varnothing$ $last\_rec_5 \leftarrow \varnothing$ $time_5$ = big_value $serial\_time_5 = 0$ $parallel\_time_5$=big_value $dist\_def_5 = 2$
diffusion(m_1) VT_1=(1,0,0,0,0) H(m) = ∅ m_1=(1,1,event, ∅) send(m_1) CI_1={(1,1,0)} $parallel\_time_1$=big_value $time_1$= time_delivery+Δ $last\_rec_1$=(1,1)				
	reception(m_1) delivery(m_1) VT_2=(1,0,0,0,0) CI_2={(1,1,0)} $parallel\_time_2$ = big_value $deadline_1$= time_delivery	reception(m_1) delivery(m_1) VT_3=(1,0,0,0,0) CI_3={(1,1,0)} $parallel\_time_3$=big_value $deadline_1$= time_delivery	reception(m_1) delivery(m_1) VT_4=(1,0,0,0,0) CI_4={(1,1,0)} $parallel\_time_4$=big_value $deadline_1$= time_delivery	reception(m_1) delivery(m_1) VT_5=(1,0,0,0,0) CI_5={(1,1,0)} $parallel\_time_5$=big_value $deadline_1$= time_delivery

	$m_1 + \Delta$ $time_2 = deadline_1$ $last\_rec_2 = (1,1)$	$m_1 + \Delta$ $time_3 = deadline_1$ $last\_rec_3 = (1,1)$	$m_1 + \Delta$ $time_4 = deadline_1$ $last\_rec_4 = (1,1)$	$m_1 + \Delta$ $time_5 = deadline_1$ $last\_rec_5 = (1,1)$
	diffusion(m_2) VT_2=(1,1,0,0,0) CI_2={(1,1,1)} H(m)={(1,1)} m_2=(2,1,event,(1,1)) send(m_2) CI_3={(1,1,1),(2,1,0)} $parallel\_time_2 = deadline_1$ $deadline_2$= time_delivery $m_2 + \Delta$ $time_2 = deadline_2$ $last\_rec_2 = (2,1)$		diffusion(m_3) VT_4=(1,0,0,1,0) CI_4={(1,1,1)} H(m)={(1,1)} m_3=(4,1,event,(1,1)) send(m_3) CI_4={(1,1,1),(4,1,0)} $parallel\_time_4 = deadline_1$ $deadline_2$= time_delivery $m_3 + \Delta$ $time_4 = deadline_2$ $last\_rec_4 = (3,1)$	
lost message m_2		reception(m_2) delivery(m_2) VT_3=(1,1,0,0,0) CI_4={(1,1,1),(2,1,0)} $parallel\_time_3 = deadline_1$ $deadline_2$= time_delivery $m_2 + \Delta$ $time_3 = deadline_2$ $last\_rec_3 = (2,1)$	reception(m_2) delivery(m_2) VT_4=(1,1,0,1,0) CI_4={(4,1,0), (2,1,0)} $parallel\_time_4 = deadline_1$ $deadline_2$= time_delivery $m_2 + \Delta$ $time_4 = deadline_2$ $last\_rec_4 = (2,1)$	**lost message m_3**
/* m_3 is received before its deadline expire*/ reception(m_3) delivery(m_3) VT_1=(1,0,0,1,0) CI_1={(1,1,1),(4,1,0)} $parallel\_time_1 = deadline_1$ $deadline_2$= time_delivery $m_3 + \Delta$ $time_1 = deadline_2$ $last\_rec_1 = (4,1)$	reception(m_3) delivery(m_3) VT_2=(1,1,0,1,0) CI_2={(2,1,0),(4,1,0)} $parallel\_time_2 = deadline_1$ $deadline_2$= time_delivery $m_3 + \Delta$ $time_2 = deadline_2$ $last\_rec_2 = (4,1)$	reception(m_3) $serial\_time = deadline_2$ $time_3 = deadline_1$ delivery(m_3) VT_3=(1,1,0,1,0) CI_3={(2,1,0),(4,1,0)} $parallel\_time_3 = deadline_1$ $deadline_2$= time_delivery $m_3 + \Delta$ $time_3 = deadline_2$ $last\_rec_3 = (4,1)$		**/* m_2 is received before its deadline expire*/** reception(m_2) delivery(m_2) VT_5=(1,1,0,0,0) CI_5={(1,1,1),(2,1,0)} $parallel\_time_5 = deadline_1$ $deadline_2$= time_delivery $m_2 + \Delta$ $time_5 = deadline_2$ $last\_rec_5 = (2,1)$
		Diffusion(m_4) VT_3=(1,1,1,1,0) CI_3={(2,1,1),(4,1,1)} H(m)={(2,1),(4,1)} m_4=(3,1,event,{(2,1), (4,1)} send(m_4) CI_3={(2,1,1),(4,1,1), (3,1,0)} $parallel\_time_3 = deadline_2$ $deadline_3$= time_delivery $m_4 + \Delta$ $time_3 = deadline_3$ $last\_rec_3 = (3,1)$		
Delayed message m_4	reception(m_4) delivery(m_4) VT_2=(1,1,1,1,0) CI_2={(2,1,1),(4,1,1), (3,1,0)} $parallel\_time_2 = deadline_2$ $deadline_3$= time_delivery $m_4 + \Delta$ $time_2 = deadline_3$ $last\_rec_2 = (3,1)$		reception(m_4) delivery(m_4) VT_4=(1,1,1,1,0) CI_4={(4,1,1),(2,1,1), (3,1,0)} $parallel\_time_4 = deadline_2$ $deadline_3$= time_delivery $m_4 + \Delta$ $time_4 = deadline_3$ $last\_rec_4 = (3,1)$	reception(m_4) **/*Detection lost message m_3 and update of vectors)*/** VT_5=(1,1,1,1,0) CI_5={(1,1,1),(2,1,1) (3,1,0)} $parallel\_time_5 = deadline_2$ $deadline_3$= time_delivery $m_4 + \Delta$

				$time_5 = deadline_3$ $last\_rec_5 = (3,1)$
				Diffusion(m_5) VT_5=(1,1,1,1,1) CI_5={(1,1,2),(2,1,2), (3,1, 1)} H(m)={(1,1),(2,1),(3,1)} m_4=(5,1,event,{(1,1), (2,1), (3,1)} send(m_5) CI_3={(3,1,1), (5,1,0)} $parallel\_time_5 = deadline_3$ $deadline_4$= time_delivery $m_5 + \Delta$ $time_5 = deadline_4$ $last\_rec_5 = (5,1)$
/* m_5 is received before deadline's m_4 and m_5 is joined to queue */ reception(m_5) /* Can't delivery causally because $m_4 \rightarrow m_5$ */ Put_message_in_queue(m_5,$time_1$)	reception(m_5) delivery(m_5) VT_2=(1,1,1,1,1) CI_2={(4,1,1), (3,1,1)} $parallel\_time_2 = deadline_3$ $deadline_4$= time_delivery $m_5 + \Delta$ $time_2 = deadline_4$ $last\_rec_2 = (5,1)$	reception(m_5) delivery(m_5) VT_3=(1,1,1,1,1) CI_3={(4,1,1), (5,1,0)} $parallel\_time_3 = deadline_3$ $deadline_4$= time_delivery $m_5 + \Delta$ $time_3 = deadline_4$ $last\_rec_3 = (5,1)$	reception(m_5) delivery(m_5) VT_4=(1,1,1,1,1) CI_4={(4,1,1),(3,1,1), (5,1,0)} $parallel\_time_4 = deadline_3$ $deadline_4$= time_delivery $m_5 + \Delta$ $time_4 = deadline_4$ $last\_rec_4 = (5,1)$	
/* m_4 is received before its deadline expire*/ reception(m_4) **/*Detection lost message m_2 and update of vectors)*/** VT_3=(1,1,0,1,0) Delivery (m_4) VT_1=(1,1,1,1,0) CI_1={(1,1,1), (4,1,1), (3,1,0)} $parallel\_time_1 = deadline_2$ $deadline_3$= time_delivery $m_4 + \Delta$ $time_1 = deadline_3$ $last\_rec_1 = (3,1)$ **/* Now the message m_5 can be delivered */** Delivery (m_5) VT_1=(1,1,1,1,1) CI_1={(4,1,1), (3,1,1), (5,1,0)} $parallel\_time_1 = deadline_3$ $deadline_4$= time_delivery $m_5 + \Delta$ $time_1 = deadline_4$ $last\_rec_1 = (5,1)$				

Capítulo 4

Mecanismo de sincronización de flujos continuos tolerante a la pérdida de mensajes

La sincronización de flujos continuos (audio y video) es una parte crítica para garantizar la apropiada presentación en algunas aplicaciones. Tales aplicaciones incluyen: videoconferencias, realidad virtual, trabajo colaborativo, entre otros. Diversos trabajos intentan resolver el problema de la sincronización de flujos continuos distribuidos; sin embargo, están lejos de resolver el problema. El problema de la sincronización es aún mas complicado cuando se consideran aspectos de comunicación reales en un sistema distribuido, tales como retrasos de mensajes y pérdida de mensajes, los cuales son los dos problemas principales encontrados en canales de comunicación no fiables y asíncronos.

En este capítulo, se resuelve principalmente el problema relacionado con la sincronización de flujos continuos en tiempo-real sobre canales de comunicación reales (existe pérdida y retraso de mensajes). Nos referimos por el término *tiempo real* a la creación y transmisión simultánea de flujos continuos, no existe un paso de pre-procesamiento ni previo almacenaje de los flujos continuos.

Se propone un mecanismo para lograr la sincronización de flujos continuos distribuidos en tiempo real en *canales de comunicación no fiables y asíncronos*. Dividimos el desarrollo del mecanismo en tres fases. En la primera fase, presentamos un modelo de sincronización temporal que describe cómo se realiza la sincronización en *canales de comunicación fiables*, el cual fue propuesto en [MOR05]. El modelo trabaja en dos niveles abstractos. En el nivel alto, la duración temporal es tomada en cuenta por representar los segmentos de flujos continuos como intervalos. En el nivel bajo, trabaja con intervalos, considerando que un intervalo está compuesto de un conjunto de mensajes ordenados secuencialmente. Se muestra que es suficiente asegurar una causalidad parcial entre los puntos finales (inicio y fin) para asegurar un orden causal a nivel intervalo. La segunda fase del mecanismo está

centrada en cómo llevar acabo la sincronización de flujos continuos aún en condiciones de pérdida de mensajes.
Proponemos un mecanismo tolerante a fallas. El mecanismo tolerante a fallas está basado en la técnica de detección y recuperación hacia adelante de mensajes perdidos, descrito en el capítulo anterior (sección 3.2), en forma distribuida evitando la retransmisión de la información. Este trabajo, es el primero en proponer una técnica de corrección de errores hacia adelante con control de causalidad para la sincronización de flujos continuos. La recuperación hacia adelante es lograda a través de la adición de redundancia. Dos tipos de redundancia son usadas: redundancia sobre el tipo de mensaje y redundancia sobre la información de control enviada [LOP05]. El segundo tipo de redundancia es calculado con base en la distancia causal entre eventos y es adaptable a las condiciones de pérdida de los canales de comunicación.

La tercera y última fase del mecanismo preserva las restricciones de entrega en tiempo real de los flujos transmitidos. Aplicamos el enfoque expresado por el método distribuido para el cálculo del tiempo de vida, el cual ha sido descrito a detalle en el capítulo 3, sección 3.3. Las relaciones temporales entre mensajes, de un mismo flujo y de diferentes flujos, son mantenidas mediante el uso de los relojes locales físicos de cada participante en forma independiente. En otras palabras, ninguna suposición es hecha acerca de la sincronización de los relojes físicos de los participantes. Cada participante determina de forma independiente si un mensaje ha sido recibido dentro de su tiempo de vida. Si un mensaje es recibido después de su tiempo de vida se considera no útil a la aplicación y es descartado.

Esta propuesta es una extensión al trabajo desarrollado en [MOR05]. La extensión está centrada principalmente en dos aspectos. El primero es el desarrollo de un mecanismo de recuperación hacia delante de mensajes perdidos. El segundo es con respecto a la preservación de las restricciones de entrega en tiempo real que los flujos continuos deben mantener.

Este capítulo está estructurado de la siguiente forma: En la sección 4.1, se describe el ambiente donde consideramos que el mecanismo de sincronización desarrollado en esta tesis puede ser ejecutado. Bases y definiciones son presentadas en la sección 4.2. El modelo de sincronización temporal que expresa las relaciones temporales basado sólo en la identificación de las dependencias de precedencia lógicas es descrito en la sección 4.3. En la sección 4.4, se presenta un mecanismo de detección y recuperación causal hacia adelante, el cual permite lograr la sincronización de flujos continuos cuando se produce la pérdida de mensajes. Finalmente, en la sección 4.5, se describe el mecanismo que preserva las restricciones de entrega en tiempo real a nivel intra-flujo e inter-flujo.

4.1. Modelo del sistema

Procesos. La aplicación bajo consideración está compuesta de un conjunto de procesos $P = \{p_1,p_2,....,p_n\}$ que se comunican sólo por el intercambio de mensajes. Los procesos están conectados a través de canales de comunicación con las siguientes características:

- La red es considerada no fiable. Los mensajes pueden ser perdidos durante su transmisión hacia su destino.

- Los mensajes difundidos pueden sufrir retrasos arbitrarios e impredecibles durante su transmisión.

- Las restricciones de entrega en tiempo real son consideradas entre los mensajes transmitidos en el sistema. Existe un retraso máximo (tiempo de vida) que un mensaje puede sufrir sin degradar la calidad de servicio de la aplicación. Cuando un mensaje es recibido con un retraso mayor a su tiempo de vida se considera no útil y es descartado.

Mensajes. Consideremos un conjunto finito de mensajes M, donde cada mensaje $m \in M$ es identificado por una tupla $m = (p,x)$ donde $p \in P$ es el emisor de m, denotado por $Src(m)$ y x es el reloj lógico local para mensajes enviados por p, cuando m es transmitido. El conjunto de destinos de un mensaje m es siempre P.

Eventos. Sea m un mensaje, y denotamos por $send(m)$ la emisión del evento m por $Src(m)$, y por $delivery(p,m)$ el evento de entrega de m al participante $p \in P$. El conjunto de eventos asociados a M es entonces el conjunto $E = \{ send(m) : m \in M \} \cup \{delivery(p,m) : m \in M \wedge p \in P\}$.

Intervalos. Consideremos un conjunto finito I de intervalos, donde cada intervalo $A \in I$ es un conjunto de mensajes $A \subseteq M$ enviados por un participante $p=Part(A)$, definido por el mapeo $Part:I \rightarrow P$. Formalmente, tenemos $m \in A \Rightarrow Src(m)=Part(A)$. Debido al ordenamiento secuencial de $Part(A)$, tenemos que para todo $m,m' \in A$, $m \rightarrow m'$ or $m' \rightarrow m$. Denotamos por a^- y a^+ los únicos mensajes de A, tal que para todo $m \in A$, tenemos que $a^- \neq m$ y $a^+ \neq m \Rightarrow a^- \rightarrow m \rightarrow a^+$. Los mensajes a^- y a^+ son los puntos límites de A. Nosotros asumimos en este trabajos que $a^- \neq a^+$.

Dos características importantes del modelo contemplado son la no suposición de la sincronización de relojes físicos entre los miembros del sistema, ni la existencia de memoria compartida. Cada participante funciona de forma independiente y realiza sus acciones con base en la información que ha recibido de los miembros del sistema.

4.2 Bases y definiciones

En esta sección presentamos algunas definiciones, usadas en el modelo de sincronización desarrollado en [MOR05], para llevar a cabo la sincronización entre flujos continuos a nivel inter-flujo.

4.2.1 Relación *Happened-Before* sobre intervalos

Identificamos y definimos dos posibles relaciones de precedencia a nivel intervalo. Estas dos relaciones están basadas en la relación *happened-before* para eventos individuales (Definición 1) presentado el capítulo 3. Las dos relaciones identificadas para intervalos son la relación causal y la relación simultánea. Para más detalles de estas definiciones, consultar [MOR05]. Iniciamos con la definición de la relación causal para ser aplicada a intervalos con base en sus puntos límites (*endpoints*).

Definición 5. La relación $\rightarrow_I$ es válida si las siguientes dos condiciones son satisfechas:

1. $A \rightarrow_I B$ if $a^+ \rightarrow_{M'} b^-$

2. $A \rightarrow_I B$ if $\exists C \mid (a^+ \rightarrow_{M'} c^- \wedge\ c^+ \rightarrow_{M'} b^-)$

Donde a^+ y b^- son los eventos (o mensajes) de inicio y fin de A y B respectivamente, c^- y c^+ son los eventos de inicio y fin de C, y $\rightarrow_{M'}$ es el orden causal parcial introducido sobre $M' \subseteq M$, donde M', en nuestro caso, es el subconjunto compuesto por lo mensajes de los extremos de los intervalos en I.

La relación de simultanneidad se define como sigue:

Definición 6. Dos intervalos A, B son simultáneos " ||| " si la siguiente condición se satisface:

$$A \,|||\, B \Rightarrow\ a^- \parallel b^- \wedge a^+ \parallel b^+$$

Lo anterior significa que un intervalo A puede ocurrir al "mismo tiempo" que otro intervalo B. En este trabajo consideramos la definición 6 como la relación complemento de la relación causal a nivel intervalo (definición 5). Esto significa que un intervalo puede únicamente preceder o ser simultáneo a otro intervalo sobre un tiempo dado.

Definición 7. Entrega causal *broadcast* para intervalos basada en los puntos finales o extremos.

Sí $(a^+, b^-) \in A \times B$, $send(a^+) \rightarrow send(b^-) \Rightarrow \forall p \in P$, $deliver(p, a^+) \rightarrow deliver(p, b^-)$ entonces

$$\forall p \in P \Rightarrow deliver\ (p,A) \rightarrow_I deliver\ (p,B)$$

4.3 Modelo de sincronización temporal

Con el objetivo de lograr la sincronización entre flujos continuos en un sistema distribuido, utilizamos un modelo para determinar las relaciones temporales basado sólo en la identificación de las dependencias de precedencia lógicas [MOR05]. El modelo traslada un escenario temporal para ser expresado en términos de las relaciones de precedencia a nivel intervalo (definido en la sección 4.3.1). A esta translación se le denomina *mapeo lógico*. En este trabajo, el mapeo lógico descompone un escenario temporal en segmentos de datos (intervalos) que son organizados acorde a su posible relación de precedencia.

4.3.1 Mapeo lógico

El proceso de crear el mapeo lógico involucra tomar cualquier par de intervalos en el sistema que componen un escenario temporal, y transformar cada par en cuatro segmentos de datos, los cuales son determinados acorde a su posible relación de dependencia de los eventos discretos que los componen. Estos segmentos de datos, de acuerdo a nuestra definición, son considerados nuevos intervalos. Los intervalos resultantes son expresados únicamente en términos de la relación *happened before* y la relación de *simultaneidad*.

Con el objetivo de considerar las siete relaciones básicas, sus inversas y mantener el modelo simplificado, identificamos primero los intervalos X y Y de cada par de intervalos en el sistema. El intervalo X será el intervalo con el primer punto final más a la izquierda, y el intervalo Y será el intervalo restante. Esto se realiza con el objetivo de asegurar que cualquier par cumpla $x^- \rightarrow y^-$ or $x^- \parallel y^-$ todo el tiempo. Una vez que los intervalos X y Y son identificados, el modelo segmenta cada par en cuatro subintervalos (A,B,C y D) (ver tabla 1). Cuando los subintervalos son identificados, procedemos a construir la estructura general $S=A\rightarrow_I W\rightarrow_I B$, donde W determina la existencia de una simultaneidad entre el par de intervalos.

Tabla 4.1. El proceso del mapeo lógico[1]

$\forall(X, Y)$	$\in$	$I \times I$	
$A(X,Y)$	$\leftarrow$	$-\{x \in X : x \rightarrow y^-\}$ $-\ \emptyset$	Para $x^- \rightarrow y^- \ \vee$ Para otros casos
$B(X,Y)$	$\leftarrow$	$-\{x \in X : y^+ \rightarrow x\}$ $-\{y \in Y : x^+ \rightarrow y\}$ $-\ \emptyset$	Para $y^+ \rightarrow x^+ \ \vee$ Para $x^+ \rightarrow y^+ \ \vee$ Para otros casos
$C(X,Y)$	$\leftarrow$	$-\ X-(A(X,Y) \cup B(X,Y))$	
$D(X,Y)$	$\leftarrow$	$-\ Y-B(X,Y)$	
$W(X,Y)$	$\leftarrow$	$C \parallel\parallel D$	
$S(X,Y)$	$\leftarrow$	$A \rightarrow_I W \rightarrow_I B$	

[1] En nuestro caso hemos considerado que un intervalo puede estar vacío. En tal caso, se aplican las siguientes propiedades:

- $\emptyset \rightarrow_I A \vee A \rightarrow_I \emptyset = A$
- $A \parallel\parallel \emptyset \vee \emptyset \parallel\parallel A = A$

Acorde a la tabla 4.2, se puede expresar cualquier posible escenario temporal basándonos únicamente en la relación *happened before* sobre intervalos y la relación de *simultaneidad* sobre intervalos. Esta característica es el punto central del modelo de sincronización.

Tabla 4.2. Relaciones temporales de Allen y su mapeo lógico.

Relaciones de Allen	Relaciones Temporales	Mapeo Lógico	Mapeo Lógico Expresado en Puntos finales
X *before* Y	xxxxx yyyy	$A \rightarrow_I B$	$a^+ \rightarrow b^-$
Y *after* X			
X *meets* Y	xxxx yyyy	$A \rightarrow_I (C \mid\mid\mid D) \rightarrow_I B$	$a^+ \rightarrow c^-$, $a^+ \rightarrow d^-$ $c^- \parallel d^-$, $c^+ \parallel d^-$ $c^+ \rightarrow b^-$, $d^+ \rightarrow b^-$
Y *meet-by* X			
X *overlaps* Y	xxxxxxxx yyyyyyyyy		
Y *overlap-by* X			
X *includes* Y	yyyy xxxxxxxxx		
Y *during* X			
X *starts* Y	xxxxx yyyyyyyyyyy	$(C \mid\mid\mid D) \rightarrow_I B$	$c^- \parallel d^-$, $c^+ \parallel d^+$ $c^+ \rightarrow b^-$, $d^+ \rightarrow b^-$
Y *started by* X			
X *finishes* Y	yyyyy xxxxxxxxx	$A \rightarrow_I (C \mid\mid\mid D)$	$a^+ \rightarrow c^-$, $a^+ \rightarrow d^-$ $c^- \parallel d^-$, $c^+ \rightarrow d^+$
Y *finished-by* X			
X *equals* Y	xxxxxxxx yyyyyyyy	$C \mid\mid\mid D$	$c^- \parallel d^-$, $c^+ \parallel d^+$

4.4 Mecanismo de sincronización con recuperación causal hacia adelante

El mecanismo de sincronización propuesto está basado en el modelo temporal de sincronización presentado en la sección anterior (ver tabla 4.1). El mecanismo lleva acabo el mapeo lógico y asegura la reproducción de los intervalos con base en sus dependencias lógicas. En esta sección, se describe la extensión realizada al mecanismo de sincronización propuesto en [MOR05]. La extensión se centra principalmente en proporcionar un mecanismo de detección y recuperación causal hacia delante de mensajes perdidos. El propósito es lograr un mecanismo de sincronización tolerante a la perdida de información. Hacemos notar que el algoritmo presentado en esta sección fue publicado en [LOP06].

Internamente, el mecanismo de sincronización utiliza dos tipos de mensajes: mensajes causales y mensajes FIFO. Los mensajes causales están divididos en: *mensajes begin*, *cut* y *end*. Los mensajes *begin* y *end* son los extremos izquierdo y derecho de los intervalos originales, el mensaje *cut* es un mensaje de control usado por el algoritmo para informar acerca de la segmentación de un intervalo. Mensajes FIFO (*fifo_p*) son sólo usados entre los extremos (*begin* y *end*) de un intervalo. Los mensajes causales también satisfacen localmente el orden FIFO.

4.4.1 Ejemplo del Mapeo Lógico

En la figura 4.1 mostramos cómo el mecanismo de sincronización realiza el mapeo lógico entre dos intervalos. En este escenario, los participantes transmiten sus mensajes en *canales de comunicación fiables* (no existen mensajes perdidos).

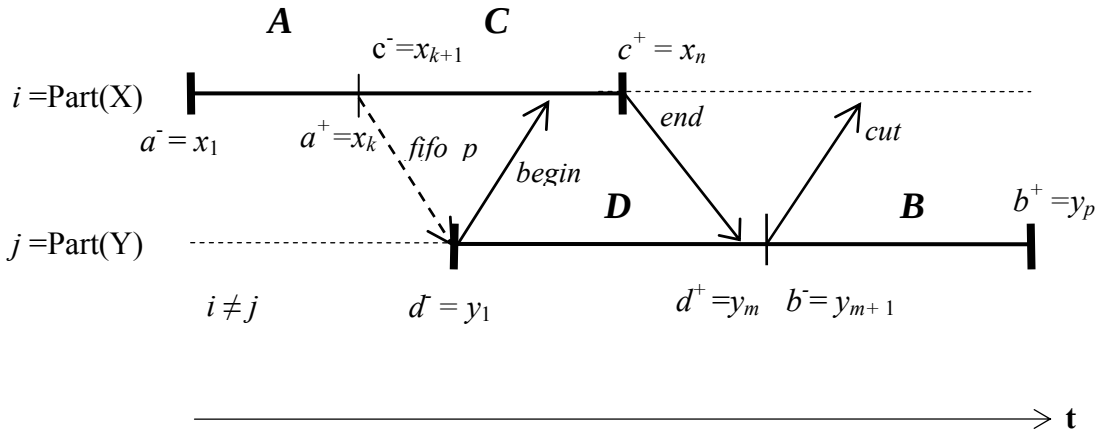

Figura 4.1. Construcción del mapeo lógico $A \rightarrow_I (C \,|||\, D) \rightarrow_I B$ para la relación *overlaps*.

En este ejemplo, el segmento *A* primero debe de ser determinado. Para realizar esto, identificamos el limite causal izquierdo a^- como el primer elemento del intervalo *X* ($x^- = x_1$), y el limite casual derecho a^+ igual a x_k. El punto final derecho a^+ es determinado por el último mensaje *fifo_p* recibido por el participante *j* antes del evento de envío del mensaje *begin*, *send*(y^-), (Líneas 19-24, Algoritmo 3). Una vez que conocemos los límites causales de *A*, podemos determinar el conjunto de mensajes que lo conforman, $A= \{x_1, x_2, \ldots. x_k\}$. Después de que el intervalo A es identificado, procedemos a distinguir los límites causales de *C* y *D*. Sobre este punto, podemos identificar los límites causales izquierdos $c^- = x_{k+1}$ y $d^- = y_1$, pero sólo hasta el evento de envío (*send*) y entrega (*deliver*) del punto final derecho x^+ podemos identificar los puntos finales derechos de *C* y *D*. Con el evento de envío (Líneas 15-18) del mensaje *end* (send ($x^+ = x_n$)), establecemos que $c^+ = x_n$, y por lo tanto, $C=\{x_{k+1}, x_{k+2},\ldots,x_n\}$. Sobre la recepción de x^+ por el participante *j*, el algoritmo envía un mensaje *cut* (Líneas 145-1147), el cual establece el final del intervalo *D* ($d^+ = y_m$) y el inicio del intervalo *B* ($cut = b^- = y_{m+1}$). Ahora, podemos determinar los elementos que forman el intervalo $D =\{ y_1, y_2,\ldots, y_m\}$. Finalmente, con el envío de y^+ (Líneas 15-18), tenemos que $b^+= y_p$, y consecuentemente, $B=\{ y_{m+1}, y_{m+2},.., y_p\}$.

4.4.2 Propuesta del Enfoque de Recuperación hacia Adelante

Como se describió en la sección anterior, los mensajes causales son usados para lograr la segmentación propuesta en el modelo temporal. En otras palabras, sincronizan los intervalos mediante sus relaciones de precedencia lógica. La pérdida de un mensaje causal interrumpe la segmentación entre intervalos, y por lo tanto, deteriora la sincronización. Para tolerar la pérdida de algún mensaje causal *begin*, *end* o *cut*, introducimos información redundante. En el algoritmo, tenemos dos clases de redundancia:

- *Redundancia sobre el tipo de mensaje*, la cual es aplicada al mensaje *begin* y al mensaje *cut.* En este tipo de redundancia, algunos mensajes *fifo_p* consecutivos de un mensaje *begin* o *cut* son enviados como copias del mensaje en cuestión. Si un mensaje *begin* o *cut* es perdido, el primer mensaje *fifo_p* es tomado como el mensaje causal perdido. Los mensajes *fifo_p* enviados como copias pueden contener información de control causal actualizada. El número de mensajes *fifo_p* enviados consecutivamente como copias de un mensaje causal no es establecido por el mecanismo, aunque un estudio realizado en [PER03] muestra que la probabilidad de que un mensaje pueda ser perdido disminuye considerablemente a partir de cinco o más mensajes consecutivos.

- *Redundancia sobre la información de control causal*. Este tipo de redundancia es con respecto a la información de control causal, la cual está basada en la distancia causal, ver capítulo 3, definición 5. Por ejemplo, para mensajes que tienen una relación de dependencia inmediata, la distancia causal es igual a uno. Si consideramos una distancia causal más grande (mayor que uno), incrementamos la redundancia en la información de control enviada en el sistema. El principal beneficio es que se incrementa el grado de tolerancia a la pérdida de mensajes. La detección y recuperación de mensajes perdidos es como sigue: cuando ocurre un evento de envío (*send*) de un mensaje causal *begin*, *end* o *cut*, la información de control agregada corresponde a los identificadores de mensajes causales que tienen una distancia causal igual o menor con respecto al evento de envío (*send*) involucrado. Con esta información redundante, es posible detectar y recuperar hacia adelante la información de control de los posibles mensajes perdidos (figura 4.4). La información de control redundante acerca de un mensaje causal es agregada sólo si la distancia causal establecida es igual o menor al número de veces que la información acerca de este mensaje ha circulado en el sistema. De esta manera, nuestro mecanismo puede ajustar la información redundante acorde al comportamiento del sistema.

A continuación describiremos a detalle cómo la *redundancia sobre el tipo de mensaje* es utilizada por el mecanismo de recuperación, para lograr la sincronización en el caso de la pérdida de mensajes *begin* y *cut*. Posteriormente, presentamos cómo la *redundancia sobre la información de control causal* es utilizada para lograr la sincronización en el caso de mensajes causales *end* perdidos.

Para explicar el funcionamiento del mecanismo de recuperación causal hacia adelante (algoritmo 3, sección 4.4.3), a continuación presentamos las principales estructuras de datos empleadas para llevar a cabo la recuperación causal hacia adelante, y consecuentemente, la sincronización de los flujos continuos.

4.4.2.1 Estructuras de datos

Principales estructuras de datos empleadas por el mecanismo de recuperación hacia adelante, presentado en el algoritmo 3 (sección 4.4.3), para llevar acabo la sincronización de flujos continuos en canales de comunicación no fiables.

- El $VT(p)$ es el *vector de tiempo.* Para cada proceso p existe un elemento $VT(p)[j]$ donde j es un identificador de proceso. El tamaño de VT es igual al número de procesos en el grupo. $VT(p)$ contiene la vista local que el proceso p tiene de la historia causal de los mensajes difundidos del sistema. En particular, el elemento $VT(p)[j]$ representa el número de mensajes recibidos del identificador j. Tiene una vista total si, en un instante t, contiene la información del último mensaje transmitido por cada proceso. Tiene una vista parcial si contiene información de un subconjunto de los procesos. Es por medio de la estructura $VT(p)$ que se garantiza la entrega causal.

- La estructura de la información de control $CI(p)$ es un conjunto de entradas (k, t, d). Cada elemento en $CI(p)$ denota un mensaje que puede no ser entregado en orden causal por el participante p. La entrada (k, t, d) representa la difusión de un mensaje por parte del participante k en un tiempo local lógico $t = VT(p)[k]$, y d representa potencialmente la distancia causal.

- La estructura de un mensaje m es una dupla $m = (i, t, message, H(m))$, donde:

  - i es el identificador del participante,
  - $t=VT(p)[i]$ es el tiempo local lógico sobre el participante i,
  - *message* es el mensaje en cuestión,
  - $H(m)$ contiene un conjunto de entradas (k,t), las cuales representan mensajes que tienen una distancia causal menor o igual a la definida en el algoritmo,

- La variable *dist_def* es la distancia causal predeterminada.

- La estructura de datos *last_cut_del* es un conjunto de mensajes m, los cuales representan los mensajes causales *cut* enviados por los participantes del sistema. Mediante *last_cut_del* podemos determinar la pérdida de un mensaje *cut*.

4.4.2.2 Mecanismo de recuperación cuando un mensaje *begin* es perdido

A continuación, explicaremos cómo la *redundancia sobre el tipo de mensaje* es usada para lograr la segmentación cuando un mensaje *begin* es perdido; ver figura 4.2. El proceso de crear el mapeo lógico es hecho en línea identificando los límites causales de los segmentos en cuestión de izquierda a derecha. El mecanismo de recuperación hacia adelante, para los distintos casos presentados en las siguientes secciones, se realiza siguiendo los pasos descritos en el algoritmo 3.

En este ejemplo, el segmento *A* primero debe ser determinado. Para realizar esto, identificamos el límite causal izquierdo a^- como el primer elemento del intervalo X ($x^- = x_1$), y el límite casual derecho a^+ igual a x_k. El punto final derecho a^+ está determinado por el último mensaje *fifo_p* recibido (Líneas 86-88) por el participante *j* antes del evento de envío del mensaje *begin* (*send*(y^-)) (Líneas 19-23). Una vez que conocemos los límites causales de *A*, podemos determinar el conjunto de mensajes que lo conforman, $A = \{x_1, x_2, \ldots x_k\}$.

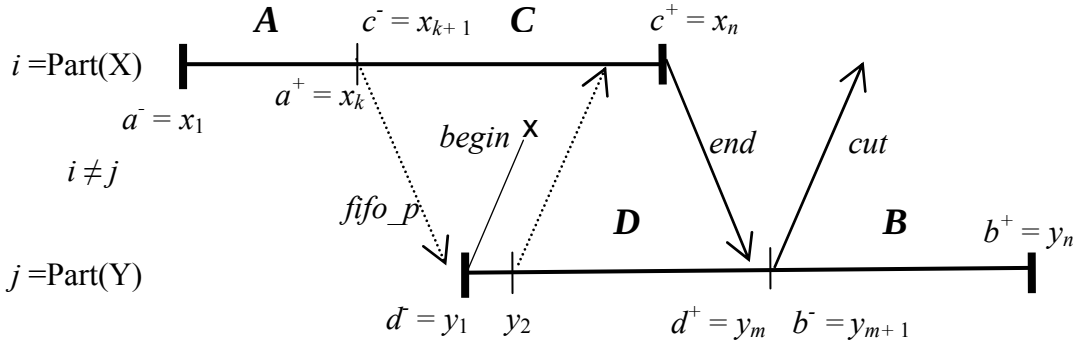

Figura 4.2. Recuperación hacia adelante sobre el proceso *i* ($A \rightarrow_I (C \,|||\, D) \rightarrow_I B$)

Después de que el intervalo A es identificado, procedemos a distinguir los límites causales de *C* y *D*. Sobre este punto, podemos identificar los límites causales izquierdos $c^- = x_{k+1}$ y $d^- = y_1$, pero sólo hasta el evento de envío (*send*) y entrega (*deliver*) del punto final derecho x^+ podemos identificar los puntos finales derechos de *C* y *D*. En este escenario, el mensaje causal y_1 es perdido durante su transmisión hacia el proceso *i*. La detección y recuperación causal hacia adelante de y_1, sobre el proceso *i*, es a través del mensaje FIFO y_2, el cual es una copia de y_1. En nuestro caso, formamos copias de un mensaje *begin* uniendo al siguiente mensaje FIFO (el cual es consecutivo al mensaje *begin* en cuestión) la misma o actualizada información de control que fue unida al mensaje causal *begin* enviado. Las líneas que construyen las copias del mensaje causal *begin* en el algoritmo son las siguientes:

```
28   ∀(x, l) ∈ reg(p)
29    If ∃ (s,t,d) ∈ cop_CI(p) | x=s and l≥t then
30        cop_CI(p) ← cop_CI(p) /  ( x, l, d )
31    Endif
```

```
32      cop_CI(p) = cop_CI(p)  ∪ last_fifo(p)
34    H(m)_Construction(cop_CI(p),copy)
```

Las líneas 28-34 determinan los elementos que formarán el $H(m)$, el cual será unido al mensaje y_2 para convertirlo en una copia del mensaje causal y_1 (*begin*). Continuando con la ejecución del escenario, el proceso *i* detecta la pérdida de y_1 sobre la recepción de y_2 y toma al mensaje FIFO y_2 como el mensaje causal *begin* del intervalo *Y*, el procedimiento es realizado en el algoritmo como sigue: en la línea 92 se detecta la pérdida del mensaje *begin* con la condición $VTI(p)[i] = 0$, la cual es verdadera por que no se ha recibido ningún mensaje del flujo transmitido por el participante *j*. Para llevar acabo la entrega del mensaje FIFO y_2 como un mensaje causal, la variable *causal* es puesta en 1 (línea 94), y consecuentemente, en la función *Update_CI_and_Last_fifo*(*m*), (líneas 131-163) actualizamos los datos recibidos del proceso *j* con información de control del mensaje y_2. Debido a que el mensaje FIFO y_2 se ha convertido en el inicio (mensaje causal *begin*) del intervalo *Y*, las líneas 111-118 verifican si cumple el orden causal, en este ejemplo el mensaje y_2 cumple el orden causal y es entregado.

```
92    If ( TP = fifo_p and VTI(p)[i] = 0 ) Then     /* Mensajes fifos usados como mensajes causales */
93        VTI(p)[ i ] = VTI(p)[ i ] + 1
94        Causal = 1

111         If not ( t' ≤ VT(p)[l]) ∀ ( l, t' ) ∈ H(m)  then       /* Condición de entrega causal */
112           If (t' > VT(p)[l]) ∀ ( l, t' ) ∈ H(m)  /*Detección de menajes perdidos*/
113             VT(p)[l] = t'      /* Actualización de vectores*/
114             If  ( TP = = cut or TP = cop_cut ) then
115               Send ( cut )     /*Deteccion */
116             Endif     /* end of If  ( TP = = cut ) then */
117           Endif
118         Endif
```

Siguiendo con el desarrollo del escenario, el envío de x_n ($send(x^+ = x_n)$) (Líneas 16-17) establece que $c^+ = x_n$, y consecuentemente, $C = \{x_{k+1}, x_{k+2}, \ldots, x_n\}$. Sobre la recepción de x^+ por el proceso *j* (Líneas 119-124), el algoritmo envía un mensaje *cut* (Líneas 145-147), el cual establece el fin del intervalo *D* ($d^+ = y_m$) y el inicio del intervalo *B* ($cut = b^- = y_{m+1}$). Como resultado, tenemos que el intervalo $D = \{y_1, y_2, \ldots, y_m\}$. Finalmente, con el evento de envío de y^+ (Líneas 16-17), tenemos que $b^+ = y_n$, y consecuentemente, $B = \{y_{m+1}, y_{m+2}, .., y_n\}$.

4.4.2.3 Mecanismo de recuperación cuando un mensaje *cut* es perdido

Otro caso en donde la *redundancia sobre el tipo de paquete* se aplica para lograr la segmentación es cuando un mensaje *cut* es perdido. En el escenario presentado en la figura 4.3, describimos a detalle el funcionamiento del mecanismo de recuperación causal cuando un mensaje *cut* es perdido.

En este ejemplo, consideremos que el segmento *A* y los límites causales izquierdos de *C* y *D* ($c^- = x_{k+1}$ y $d^- = y_1$, respectivamente) han sido identificados. Continuando con la ejecución, sólo hasta el evento de envío (*send*) y entrega (*deliver*) del punto final derecho x^+ podemos identificar los puntos finales derechos de *C* y *D*. Con el evento de envío del mensaje *end* (send($x^+ = x_n$)), establecemos que $c^+ = x_n$, y por lo tanto, $C=\{x_{k+1}, x_{k+2},\ldots,x_n\}$.

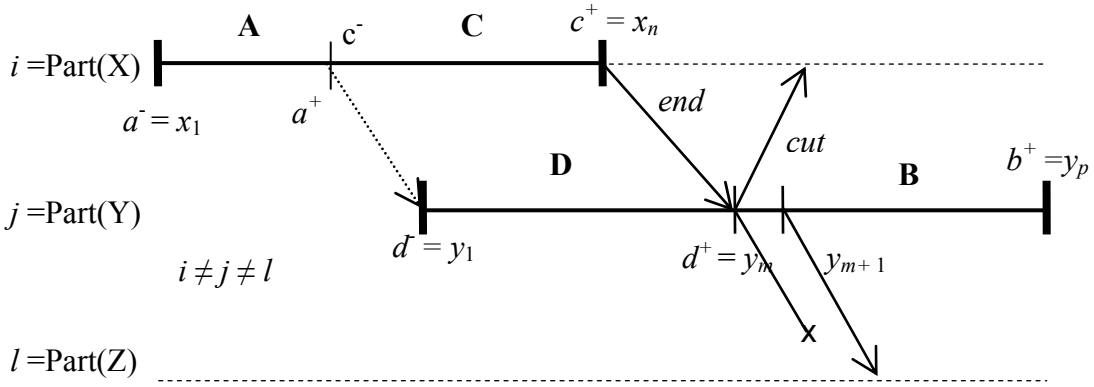

Figura 4.3. Recuperación causal hacia delante sobre el proceso *l* ($A \rightarrow_I (C \,|||\, D) \rightarrow_I B$)

Sobre la recepción de x^+ por el participante *j*, el algoritmo envía un mensaje *cut* (y_m) (líneas 145-147, algoritmo 3), el cual establece el final del intervalo *D* ($d^+ = y_m$) y el inicio del intervalo *B* ($cut = b^- = y_{m+1}$). En este ejemplo, el mensaje causal *cut* (y_m) es perdido durante su transmisión hacia el proceso *l*. La detección y recuperación causal hacia adelante de y_m, sobre el proceso *l*, es a través del mensaje FIFO y_{m+1} el cual es una copia de y_m. En nuestro caso, formamos copias de un mensaje *cut* uniendo al siguiente mensaje FIFO (el cual es consecutivo al mensaje *cut* en cuestión, ver figura 4.3) la misma o actualizada información de control que fue unida al mensaje causal *cut* perdido. Las líneas que construyen copias de un mensaje causal *cut* en el algoritmo son las siguientes:

```
38   If ( copies_of_cut_message  < num_cop )  then
39       H(m)_Construction(CI(p),copy)
40       copies_of_cut_message = copies_of_cut_message  + 1
41       TP = cop_cut
42    Else
```

La condición expresada en la línea 38 determina el número de mensajes *fifo_p* enviados como copias del mensaje causal *cut*. En la línea 39 se determina la información de control que será unida al mensaje *fifo_p* con la finalidad de convertirlo en una copia del mensaje *cut*. Sobre la línea 41, el mensaje *fifo_p* es etiquetado como *cop_cut*. El mensaje *cop_cut* es utilizado por el receptor para detectar y recuperar un mensaje causal *cut* perdido.

Continuando con la ejecución del escenario, el proceso *l* detecta la pérdida de y_m sobre la recepción de y_{m+1}, y toma al mensaje y_{m+1} como el mensaje causal *cut* del intervalo *Y,* el procedimiento en el algoritmo es realizado de la siguiente manera: En las líneas 97-98 analizamos si existe un mensaje causal *cut* (*m'*) enviado por el mismo participante del mensaje y_{m+1}. En este ejemplo, el participante *l* no ha recibido ningún mensaje causal *cut*, por lo tanto, detectamos la pérdida del mensaje causal y_m y consideramos al mensaje y_{m+1} como el mensaje *cut* enviado por el participante *j* (líneas 152-153).

97 $\forall\ m' \in last\_cut\_del(p)$
98 If $\exists\ j \in m' \mid j = i$ then

105 Else /* else of If $\exists\ j \in m' \mid j = i$ then */
152 $last\_cut\_del(p) \leftarrow last\_cut\_del(p) \cup m$
153 $causal = 1$
154 Endif /* end of If $\exists\ j \in m' \mid j = i$ then */

Como consecuencia de la recuperación causal hacia delante, el proceso *l* puede determinar los elementos que forman el intervalo $D = \{ y_1, y_2, \ldots, y_{m+1}\}$. Finalmente, con el envío de y^+, tenemos que $b^+ = y_p$, y consecuentemente, $B = \{y_{m+2}, y_{m+3}, .., y_p\}$. En conclusión, tenemos que la segmentación de los intervalos se llevó a cabo aún en condiciones de pérdida, dando por resultado el mapeo lógico $A \rightarrow_l (C \,|||\, D) \rightarrow_l B$.

4.4.2.4 Mecanismo de recuperación cuando un mensaje *end* es perdido

Ahora, explicaremos cómo la redundancia basada en la *distancia causal* es usada para lograr la sincronización en la presencia de mensajes causales *end* perdidos. En este ejemplo, consideremos que el segmento *A* y los límites causales izquierdos de *C* y *D* ($c^- = x_{k+1}$ y $d^- = y_1$, respectivamente) están identificados (figura 4.4).

Únicamente a través de los eventos de envío (*send*) y entrega (*delivery*) del punto final derecho x^+, podemos identificar los puntos finales derechos de *C* y *D*. Con el envío de x^+ ($send(x^+ = x_n)$) (líneas 16-17) establecemos que $c^+ = x_n$, y consecuentemente, $C = \{x_{k+1}, x_{k+2}, \ldots, x_n\}$. En este ejemplo, el mensaje causal $x^+ = x_n$ es perdido durante su transmisión hacia el proceso *j,* y consecuentemente, no podemos identificar el punto final derecho del intervalo *D* ni el inicio del intervalo *B*. Siguiendo con el desarrollo del escenario, sobre la recepción de x^+ en el proceso *l* el algoritmo envía un mensaje *cut* (Líneas 145-147), el cual

lleva unida información de control acerca de los mensajes x^+ y y^- (Líneas 16-17) con distancias causales *dist* = 1 y *dist* = 2, respectivamente.

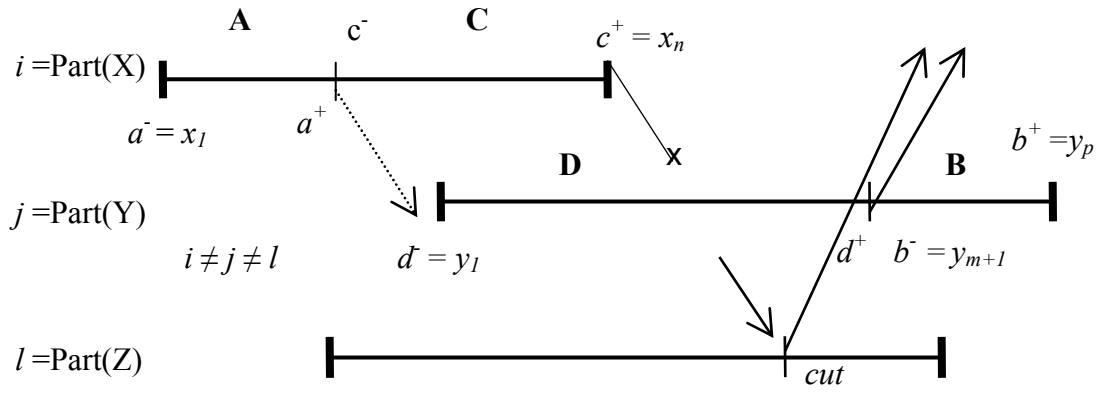

Figura 4.4. Recuperación hacia adelante sobre el proceso *j* con d=2 ($A \rightarrow_I (C \,|||\, D) \rightarrow_I B$)

Usando la información de control unida al mensaje *cut*, el proceso *j* es capaz de detectar que el mensaje x^+ ha sido perdido, la detección y recuperación es llevada a cabo en el algoritmo de la siguiente forma: en la línea 62 se analiza si el mensaje causal *cut* cumple el orden causal. En este caso, el mensaje *cut* no cumple el orden causal debido a que el mensaje *end* x^+ fue perdido. Ahora, en la línea 63 se detecta la pérdida del mensaje x^+ con la información de control unida al mensaje *cut*. Dado que el mensaje x^+ es un punto final (mensaje *end*), el proceso *j* procede a enviar un mensaje *cut* (Línea 66, ver figura 4.4) el cual establece el fin del intervalo *D* ($cut=d^+= y_m$) y el inicio del intervalo *B* ($b^-= y_{m+1}$).

```
62   If not ( t' ≤ VT(p)[l]) ∀ ( l, t' ) ∈ H(m)     then          /* causal delivery condition*/
63       If (t' > VT(p)[l]) ∀ ( l, t' ) ∈ H(m)  /*Detection of lost message*/
64         VT(p)[l] = t'      /*update the vectors*/
65         If (TP = = cut ) then
66           Send ( cut )     /*Sending a cut message by the lost end message */
67         Endif
68       Endif
69   Endif        /* endif  of  If not ( t' ≤ VT(p)[l]) ∀ ( l, t' ) ∈ H(m) */
```

Como resultado, tenemos el conjunto de elementos que forman $D=\{ y_1, y_2,\ldots, y_m\}$. Finalmente, con el evento de envío de y^+, conocemos que $b^+= y_p$, y consecuentemente, $B=\{ y_{m+1}, y_{m+2},\ldots, y_p\}$. En conclusión, la segmentación para los intervalos *X* y *Y* sobre el proceso *j* es lograda: $\boldsymbol{A \rightarrow_I (C \,|||\, D) \rightarrow_I B.}$

4.4.3 Especificación del algoritmo de sincronización de flujos continuos con recuperación causal hacia adelante

El Mecanismo presentado en el algoritmo 3 lleva a cabo la sincronización de flujos continuos, en canales de comunicación no fiables, mediante un mecanismo de recuperación causal hacia adelante de mensajes perdidos.

Algoritmo 3. Mecanismo de sincronización de flujos continuos tolerante a fallas

1.	**Initially**
2.	*VT*(*p*)[*j*] = 0 ∀ *j*: 1…n /* vector timeclock */
3.	*VTI*(*p*)[*j*] = 0 ∀ *j*: 1…n /* states vector used to detect the lost of a begin or cut message */
4.	*CI*(*p*)←∅ /* message that is not ensured by participant *p* of being delivered in a causal order */
5.	*Act*=0 /* indicate that process p is inactive */
6.	*last_fifo*(*p*)←∅ /* last FIFO message delivered */
7.	*copies_of_cut_message* = *num_cop* /* copy numbers of message cut */
8.	*con*=0 /* account of copies of begin message */
9.	*causal=0* /* if causal is equal 1,a fifo_p or cut copy message is used as causal */
10.	*last_cop_cut* ←∅ /* last cop_cut or cut message received */
	For each *m* message diffused by *p* with process identifier *i*
11.	***Send*** (*Input*: *TP* = *begin* | *end* | *cut* | *fifo_p* | *cop_cut*))
12.	*VT*(*p*)[*i*] ←*VT*(*p*)[*i*] +1 /* account of send messages by process *p* */
13.	*con* = (*TP* = *fifo_p*) ? *con* +1 : *con*
14.	If not (*TP* = *fifo_p* and *con* > *num_cop*) then
15.	If not (*TP* = *begin* or *TP* = *fifo_p* or *TP* != *cop_cut*) Then
16.	H(m)_Construction (*CI*(*p*), increment) /*Construction of the *H*(*m*) for *end* and *cut* message*/
17.	*copies_of_cut_message* = (*TP* = *cut*) ? 0 : *copies_of_cut_message* /* init the account of copies */
18.	Else
19.	If not (*TP* = *fifo_p*) then /*Construction of the *H*(*m*) begin message*/
20.	*CI*(*p*) ← *CI*(*p*) ∪ *last_fifo*(*p*) /*Adding *fifo_p* messages to CI(p)*/
21.	H(m)_Construction (*CI*(*p*), increment)
22.	*reg*(*p*) ← *last_fifo*(*p*)
23.	*last_fifo*(*p*) ← ∅
24.	Else /* else If not (*TP* = *fifo_p*) then */
25.	*reg*(*p*) ← *last_fifo*(*p*)
26.	c*op_CI*(*p*)←*CI*(*p*) /*construction of copies of *begin* message*/
27.	If not (*last_fifo*(*p*) = ∅) then
28.	∀(*x*, *l*) ∈ *reg*(*p*)
29.	If ∃ (*s*, *t*, *d*) ∈ *cop_CI*(*p*) | *x* = *s* and *l* ≥ *t* then /*construction of the H(m) for copies of begin
30.	*cop_CI*(*p*) ← *cop_CI*(*p*) / (*x*, *l*, *d*)
31.	Endif
32.	*cop_CI*(*p*) = *cop_CI*(*p*) ∪ *last_fifo*(*p*)
33.	Endif

34.	H(m)_Construction(*cop_CI*(*p*), copy) /*Construction of the *H*(*m*) for copies of begin message*/
35.	Endif /* endif of If not (*TP* = *fifo_p*) then */
36.	Endif /* If not (*TP* = *begin* or *TP* = *fifo_p* or *TP* != *cop_cut*) Then
37.	Else /*construction of copies of *cut* message*/
38.	If (*copies_of_cut_message* < *num_cop*) then
39.	H(m)_Construction(*CI*(*p*),copy) /*the first element of H(m) in a cop_cut message is information
40.	*copies_of_cut_message* = *copies_of_cut_message* + 1 control end message.*/
41.	*TP* = *cop_cut*
42.	Else
43.	*H*(*m*)← ∅ /* *H*(*m*) for FIFO messages */
44.	Endif /* endif of If (*copies_of_cut_message* < *num_cop*) then */
45.	Endif /* endif of If not (*TP* = *fifo_p* and *con* > *num_cop*) then */
46.	If (*TP* = *end*) then /*Determine if process *p* is sending or not an interval*/
47.	Act = 0
48.	con = 0
49.	Else
50.	Act = 1
51.	Endif
52.	*m* = (*i*, *t* = *VT*(*p*)[*i*], *TP* , *H*(*m*),*data*) /*construction of sent messages*/
53.	*Sending*(*m*)
54.	*H*(*m*) ←∅
55.	If ∃ (*k*, *t*, *d*) ∈ *CI*(*p*) | *d* = *dist_def* then /*delete the messages with causal distance equal to dist_def */
56.	*CI*(*p*) ← *CI*(*p*) / (*k*, *t*, *d*)
57.	Endif
58.	**Endfor**
	For each message received by *p* with process identifier *j*
59.	*Receive*(*m*) in *p* with *i* ≠ *j* and *m*= (*i*, *t* = *VT*(*p*)[*i*], *TP* , *H*(*m*),*data*)
60.	If *t* = *VT*(*p*)[*i*] +1 Then /*FIFO deliver condition*/
61.	If (*TP* ! = *fifo_p* and TP != *cop_cut*) then
62.	If not (*t'* ≤ *VT*(*p*)[*l*]) ∀ (*l*, *t'*) ∈ *H*(*m*) then /* causal delivery condition*/
63.	If (*t'* > *VT*(*p*)[*l*]) ∀ (*l*, *t'*) ∈ *H*(*m*) /*Detection of lost message*/
64.	*VT*(*p*)[*l*] = *t'* /*update of vectors*/
65.	If (TP = = *cut*) then
66.	Send (*cut*) /*Detection of lost *end* message */
67.	Endif /* end of If (TP = = *cut*) then */
68.	Endif /* endif of If (*t'* > *VT*(*p*)[*l*]) ∀ (*l*, *t'*) ∈ *H*(*m*) */
69.	Endif /* endif of If not (*t'* ≤ *VT*(*p*)[*l*]) ∀ (*l*, *t'*) ∈ *H*(*m*) */
70.	Deliver(m)
71.	*VT*(*p*)[*i*] = *VT*(*p*)[*i*] + 1
72.	Update_CI_and_Last_fifo (m)
73.	If (TP = = *cut*) then
74.	∀ *m*' ∈ *last_cut_del*(*p*)

```
75.                         If ∃ i' ∈ m' | i' = i then
76.                             Last_cut_del(p) ← last_cut_del(p) / m'
77.                         Endif
78.                     last_cut_del ← last_cut_del ∪ m
79.                 Endif   /* endif of If ( TP = = cut ) then */
80.                 If (TP = = end ) then
81.                     VTI(p)[i] = 0
82.                   Else  /* else of If ( TP = = end ) then */
83.                     VTI(p)[i] = VTI(p)[i] + 1
84.                 Endif     /* endif of If (TP = = end ) then   */
85.             Else          /* else of If ( TP ! = fifo_p and TP != cop_cut ) then , deliver of FIFO message  */
86.                 Deliver(m)
87.                 VT(p)[i] = VT(p)[i] + 1
88.                 Update_CI_and_Last_fifo ( m )
89.         Endif                   /* endif of   If ( TP ! = fifo_p and TP != cop_cut ) then */
90.     Else            /*   else of   If t = VT(p)[i] +1 Then    */
91.       If  ( t > VT(p)[ i ] ) then
92.           If ( TP = fifo_p and VTI(p)[i] = 0 ) Then       /* fifo message used as message causal */
93.               VTI(p)[ i ] = VTI(p)[ i ] + 1
94.               Causal = 1
95.             Else          /* else of  If ( TP = fifo_p and VTI(p)[i] = 0 ) Then */
96.               If ( TP = cop_cut  or TP = cut ) then     /* cop_cut message used as a message causal */
97.                       ∀ m' ∈ last_cut_del(p)
98.                         If ∃ j ∈ m' | j = i then
99.                             If ( H( last_cut_del ) ! = H(m) ) then      /*Detection of lost cut message*/
100.                                VTI(p)[i] =   VTI(p)[i] +1
101.                                last_cut_del(p) ← last_cut_del(p) / m'
102.                                last_cut_del(p) ← last_cut_del(p)  ∪ m
103.                                causal = 1
104.                            Endif     /* endif of If ( H( last_cut_del ) ! = H(m) ) then */
105.                        Else  /* else of  If ∃ i' ∈ m' | i' = i then */
106.                            last_cut_del(p) ← last_cut_del(p)  ∪ m
107.                            causal = 1
108.                        Endif  /* end of If ∃ i' ∈ m' | i' = i then  */
109.                Endif         /* endif of  If ( TP = cop_cut  or TP = cut ) then  */
110.         Endif         /*  endif of if  If ( TP = fifo_p and VTI(p)[i] = 0 ) Then  */
111.         If not ( t' ≤ VT(p)[l]) ∀ ( l, t' ) ∈ H(m)     then          /* causal delivery condition*/
112.             If (t' > VT(p)[l]) ∀ ( l, t' ) ∈ H(m)  /*Detection of lost message*/
113.                 VT(p)[l] = t'      /*update of vectors*/
114.                 If  ( TP = = cut or TP = cop_cut ) then
115.                     Send ( cut )     /*Detection of lost end message */
116.                 Endif     /* end of If  ( TP = = cut ) then */
117.             Endif
```

```
118.            Endif
119.            Deliver(m)
120.            VT(p)[ i ] = t
121.            If (TP = = end ) then
122.                 VTI(p)[i] =  0
123.            Endif   /* else of  If  ( TP = = end ) then */
124.            Update_CI_and_last_fifo ( m )
125.          Else     /* else of  if ( t > VT(p)[i] )  condition */
126.             Discard (m)
127.       Endif        /* end of  if ( t > VT(p)[i] )  condition */
128.    Endif         /* end of  if ( t =  VT(p)[i] +1) FIFO deliver condition */
129.  EndFor      /* end of function receive */
130.
131.  Update_CI_last_fifo ( m ) {    /*updating the CI and last FIFO with the message delivered */
132.   If  not ( (TP =  fifo_p or TP = cop_cut )  and causal = 0  )    Then   /* update the CI and last_fifo */
133.      CI(m) ← CI(m) ∪ { ( i, t, d = 0) }
134.      ∀(l,  t' )  ∈  H(m)                                  /*Updating  CI(p)with  a  most  recent  message*/
135.         If ∃ (s, t, d ) ∈ CI(p) | l = s and  t' = t  then
136.             (s, t, d ) ← (s, t, d+1)
137.         Endif
138.         If ∃ (s, t, d ) ∈ last_fifo(p) | l = s and t' = t  then     /* incrementing the causal distance of last FIFO*/
139.            (s, t, d ) ← (s, t, d +1)
140.         Endif
141.         If  ∃ (s,  t,  d)  ∈  CI(p)  |  d =  dist_def   then   /*delete the element with causal distance equal defined
142.             CI(p) ← CI(p) /  ( s, t, d )
143.         Endif
144.      VTI(p)[ i ] = ( TP = end ) ? 0 : VTI(p)[ i ]  /* init the VTI  when a end message is delivered */
145.      If ( Act = 1 and  not( TP = cut) and  not ( TP = begin)  then
146.         Send( cut )     /*sending a cut message by the deliver of a end message*/
147.      Endif
148.      If ( TP = cut ) then
149.          Last_cop_cut = H(m)
150.         VTI(p)[ i ] = VTI(p)[ i ] + 1
151.      Endif
152.    Else
153.      If  ∃  (x, l, d ) ∈ last_fifo(p) | x = i  then    /*Updating last_fifo(p) with a most recent message*/
154.          Last_fifo(p) ←  last_fifo(p) / ( x, l, d )
155.      Endif
156.      last_fifo(p) ← last_fifo(p) ∪ ( i, t, d=0 )   /* adding the last fifo*/
157.      causal = 0
158.  Endif   /* endif of If  not ( (TP =  fifo_p or TP = cop_cut )  and causal = 0  ) */
159.   If ∃ (s, t, d ) ∈ last_fifo(p)  | d = dist_def  then /*delete a element in last fifo */
160.       last_fifo(p) ←  last_fifo(p) / (s, t, d )
```

```
161.  Endif
162. Endif            /* end of function actualizacion_CI_last_fifo */
163. }
164.
165. H(m)_Construction (  CI(p), bandera  ) {
166.   For each (s,t,d) ∈ CI(p)
167.          (s,t,d) ← (bandera = increment) ? (s,t,d+1) : (s,t,d)
168.          H(m)  ← H(m) ∪ (s, t )
169.       Endfor
170. }
```

4.5 Algoritmo de sincronización de flujos continuos con restricciones de entrega en tiempo real

En la sección anterior, presentamos un algoritmo de sincronización de flujos continuos tolerante a la pérdida de mensajes, el cual realiza la sincronización en canales de comunicación no fiables. Sin embargo, no considera restricciones de entrega en tiempo real entre mensajes. Al contemplar restricciones de entrega en tiempo real, consideramos que cada mensaje transmitido tiene un *tiempo de vida*, después del cual sus datos son considerados no útiles. El *tiempo de vida* es el periodo de tiempo que una aplicación considera útil a los datos transportados por un mensaje. Un mensaje que sufre un retraso mayor a su *tiempo de vida* debe ser descartado. Para considerar las restricciones de entrega en tiempo real, extendemos el algoritmo de sincronización de flujos continuos presentado en [MOR05], incorporando un mecanismo (*método del cálculo de tiempo de vida distribuido*) que mantiene las restricciones de entrega en tiempo real entre mensajes, el cual fue desarrollado en el algoritmo causal mostrado en el capítulo 3, sección 3.3.

En nuestro caso, las restricciones de entrega en tiempo real deben ser preservadas entre mensajes de un mismo flujo (intra-flujo), y entre mensajes de diferentes flujos (Inter-flujo). En las siguientes secciones describimos a detalle cómo se mantienen las restricciones de entrega en tiempo real entre mensajes de un mismo flujo y de diferentes flujos.

4.5.1 Estructuras de datos

Para explicar el funcionamiento del mecanismo que mantiene las restricciones de entrega en tiempo real a nivel intra-flujo e inter-flujo (algoritmo 4, sección 4.5.4), presentamos a continuación las principales estructuras de datos empleadas en el mecanismo.

- El $VT(p)$ es el *vector de tiempo.* Para cada proceso p existe un elemento $VT(p)[j]$ donde j es un identificador de proceso. El tamaño de VT es igual al número de procesos en el grupo. $VT(p)$ contiene la vista local que el proceso p tiene de la historia causal de los mensajes difundidos del sistema. En particular, el elemento $VT(p)[j]$ representa el número

de mensajes recibidos del identificador *j*. Tiene una vista total si, en un instante *t*, contiene la información del último mensaje transmitido por cada proceso. Tiene una vista parcial si contiene información de un subconjunto de los procesos. Es por medio de la estructura *VT*(*p*) que se garantiza la entrega causal.

- La estructura de la información de control *CI*(*p*) es un conjunto de entradas (*k, t, d*). Cada elemento en *CI*(*p*) denota un mensaje que puede no ser entregado en orden causal por el participante *p*. La entrada (*k, t, d*) representa la difusión de un mensaje por parte del participante *k* en un tiempo local lógico $t = VT(p)[k]$, y *d* representa potencialmente la distancia causal.

- La estructura de un mensaje *m* es una dupla $m = (i, t, message, H(m))$, donde:

  - *i* es el identificador del participante,

  - $t=VT(p)[i]$ es el tiempo local lógico sobre el participante *i,*

  - *message* es el mensaje en cuestión,

  - *H*(*m*) contiene un conjunto de entradas (*k*,*t*), las cuales representan mensajes que tienen una distancia causal menor o igual a la definida en el algoritmo.

- El *VTIME*(*p*) es el vector de tiempo físico. Para cada proceso *p* existe un elemento *VTIME*(*p*)[*j*] donde *j* es un identificador de proceso. El tamaño de *VTIME* es igual al número de procesos en el grupo. *VTIME*(*p*) contiene el *deadline* del siguiente mensaje a recibir por el participante *p*.

- *current_time* es el tiempo local físico del participante *p*

- La estructura *queue* es un conjunto de mensajes. Cada elemento en *queue* tiene asignado un determinado *tiempo de espera*. Cuando el *tiempo de espera* de un mensaje expira, el mensaje debe ser entregado a la aplicación.

4.5.2 Mecanismo para mantener las restricciones de entrega en tiempo real a nivel intra-flujo

Para describir a detalle cómo se preservan las relaciones temporales entre mensajes de un mismo flujo presentamos el siguiente escenario (figura 4.5). El mecanismo empleado para mantener las restricciones de entrega en tiempo real utiliza los principios del mecanismo desarrollado en el capítulo 3, sección 3.3. Cada participante utiliza su reloj local físico, de manera independiente, para determinar si un mensaje ha sufrido un retraso mayor a su *tiempo de vida*. En este ejemplo, el inicio del intervalo *X* es determinado por el mensaje x_1. La entrega del mensaje x_1 sobre los procesos *l* y *j* es realizada en un tiempo *T* denominado

tiempo de entrega ($T_i \neq T_l$), el procedimiento en el algoritmo es realizado en la línea 58, (Algoritmo 4).

58 $VTIME(p)[i] = current\_time + \Delta$

El tiempo de entrega de x_l se obtiene a través de la variable *current_time*, la cual representa el tiempo actual del reloj local físico del participante en cuestión. El valor expresado por la suma de *current_time* y el *tiempo de vida* (Δ) es el *deadline* del siguiente mensaje a recibir. El *deadline* es el retraso máximo que un mensaje puede sufrir durante la transmisión hacia su destino. La estructura *VTIME* (*p*) mantiene las *referencias de tiempo* (*deadline*) de cada participante en el sistema que está transmitiendo un flujo.
Sobre este punto, el participante *l* tiene en su estructura *VTIME* (*p*) [*i*] el tiempo límite (*deadline*) para el siguiente mensaje FIFO a recibir, en este caso es el mensaje x_{k+1}.

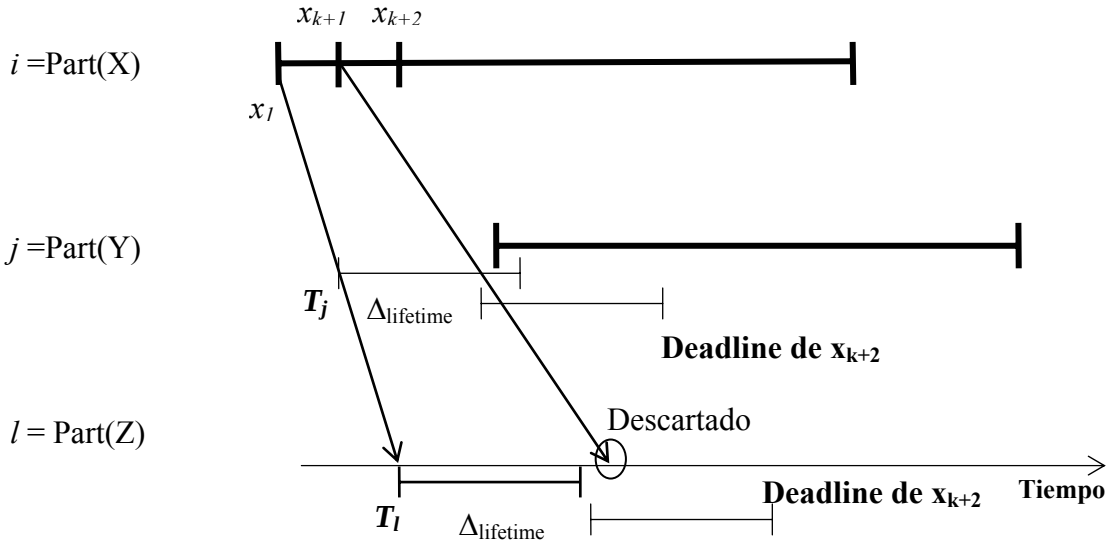

Figura 4.5. Determinación del *deadline* de los mensajes de un mismo flujo

Cada vez que un mensaje es recibido, se verifica si el mensaje ha llegado antes de la expiración de su *deadline*, si el mensaje es recibido antes de su *deadline* es entregado, en caso contrario es descartado. La condición expresada en la línea 38 determina si un mensaje debe ser aceptado o descartado.

38 If (*reception_time_message* > $VTIME(p)[i]$ and *TP*!=*Begin*) then

Continuando con la ejecución del escenario, sobre el participante *j* el mensaje x_{k+1} es recibido antes de la expiración de su *deadline*, y es entregado (Líneas 74-77). Ahora, el proceso *j* determina el *deadline* (tiempo de entrega de x_{k+1} + Δ de x_{k+2}) para el siguiente mensaje FIFO a recibir, el cual es x_{k+2}. Lo anterior es llevado a cabo en el algoritmo en la línea 77.

77 VTIME(p)[i] = VTIME(p)[i] + Δ

Cuando un mensaje es recibido después de su *deadline* se considera no útil a la aplicación y es descartado. Sobre el participante *l*, el mensaje x_{k+1} es recibido después de su *deadline* y es descartado, el procedimiento en el algoritmo es como sigue: la líneas 38 determina que el mensaje x_{k+1} es recibido después de su *deadline*, y consecuentemente, el mensaje es descartado. Actualizamos la información del participante *i* con información de control de x_{k+1}, línea 40.

38 If (*reception_time_message* > *VTIME*(*p*)[*i*] and *TP*!=*Begin*) then
39 Discard(*m*)
40 *VT*(*p*)[*i*] = *t*
41 *VTIME*(*p*)[*i*] = *VTIME*(*p*)[*i*] + Δ
42 Else

Al ser descartado el mensaje x_{k+1}, el *deadline* para el mensaje x_{k+2} es igual a la suma del *deadline* del mensaje x_{k+1} y el tiempo de vida del mensaje x_{k+2}, línea 41. Hacemos notar que los participantes mantienen las restricciones de entrega en tiempo real usando de manera independiente su reloj local físico. Cada participante, de acuerdo a la vista que posee del sistema, realiza sus acciones de forma autónoma y distribuida

4.5.3 Mecanismo para mantener las restricciones de entrega en tiempo real a nivel inter-flujo

La sección anterior describe el mecanismo que mantiene las relaciones temporales en tiempo real entre mensajes de un mismo flujo. Sin embargo, existe otro tipo de restricciones en tiempo real que se deben preservar, las cuales corresponden a las restricciones entre mensajes que poseen una relación inter-flujo. En nuestro caso, el mensaje causal *begin* debe mantener las restricciones de entrega en tiempo real únicamente a nivel inter-flujo, los mensajes causales *end* y *cut* deben preservar sus restricciones de entrega en tiempo real a nivel intra-flujo e inter-flujo. A continuación describimos cómo funciona el mecanismo para mantener las relaciones temporales a nivel inter-flujo, en los casos de mensajes causales *begin*,*end* y *cut*.

4.5.3.1 Restricciones de entrega en tiempo real en mensajes causales *begin*

Con la finalidad de explicar cómo el mecanismo mantiene las restricciones de entrega en tiempo real a nivel inter-flujo en el caso de mensajes causales *begin*, presentamos el siguiente escenario (ver figura 4.6). En nuestro algoritmo, el mensaje causal *begin* lleva unida información de control acerca del último mensaje recibido de cada participante. En el ejemplo, el mensaje causal *begin* w_1 lleva unida información de control acerca de los mensajes x_k y y_n transmitidos por los participantes *i* y *j* respectivamente (Líneas 19-25, Algoritmo 4). Por lo tanto, el mensaje causal *begin* w_1 debe preservar las restricciones de

entrega en tiempo real a nivel inter-flujo con los mensajes x_k y y_n. El mensaje y_n es entregado antes que el mensaje x_k sobre el participante *l*.

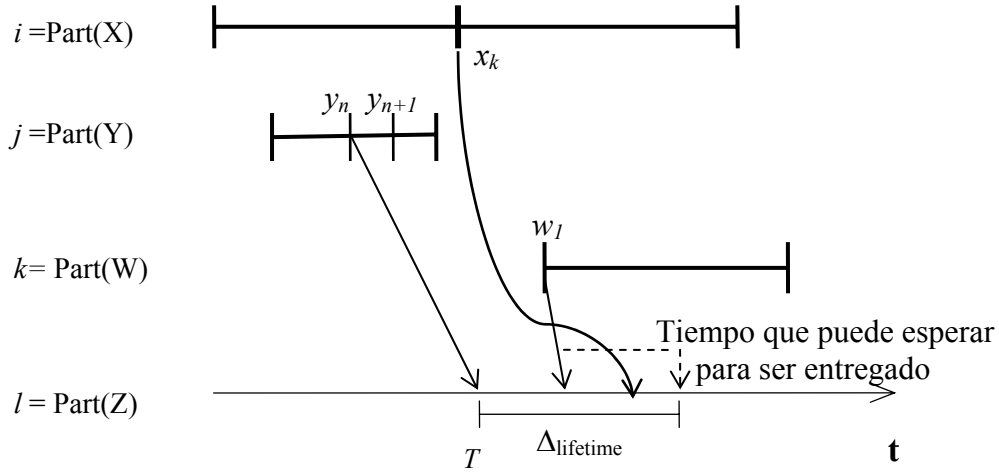

Figura 4.6. Mecanismo para preservar las restricciones de entrega en tiempo real a nivel inter-flujo en mensajes causales *begin*

En este caso, el *deadline* del mensaje w_1 es igual a la suma del tiempo de entrega *T* del mensaje y_n y el tiempo de vida del mensaje causal *begin* a nivel inter-flujo, el procedimiento es realizado por el algoritmo mediante la llamada a la función *Begin_deadline_measure*(*m*) (línea 45), la cual calcula el *deadline* del mensaje *begin* de la siguiente forma: En la línea 108, el valor contenido en $\Delta_{inter\text{-}st}$ representa la diferencia del tiempo de vida a nivel inter-flujo e intra-flujo del mensaje *begin*. El tiempo de vida para mensajes que tienen una relación inter-flujo es diferente del tiempo de vida para mensajes con una relación intra-flujo. El primero es un periodo de tiempo mayor, el cual puede tolerar la aplicación sin degradar la calidad de servicio. Siguiendo con el cálculo del *deadline* del mensaje *begin*, las líneas 110-115 determinan a través de la información de control unida al mensaje *begin* el *deadline* que expira primero a nivel intra-flujo, en este caso es el *deadline* del mensaje y_n (figura 4.6), al cual se le suma el valor expresado en $\Delta_{inter\text{-}st}$ (línea 116) obteniendo de esta forma el *deadline* del mensaje begin a nivel inter-flujo.

```
108   Begin_deadline_measure( m ) {
109      Δinter-st = Δinter - Δintra
110      ∀(l, t' )∈ H(m)
111         If ( t' ≤ VT(p)[l]) then    /* causal delivery condition*/
112            If (begin_deadline>VTIME(p)[l] ) then
113               begin_deadline = VTIME(p)[l]
114            Endif
115         Endif
116      begin_deadline = begin_deadline + Δinter-st
```

Siguiendo con la ejecución del escenario, el mensaje x_k sufre un retraso mayor que el mensaje causal w_1 durante su transmisión hacia el proceso *l*. El participante *l* recibe primero

el mensaje causal w_1 y lo pone en espera (Líneas 52) debido a que el mensaje x_k lo precede causalmente ($x_k \rightarrow w_1$), ver figura 4.6, el participante l retrasa la entrega de w_1 hasta el vencimiento de su *deadline* con la finalidad de recibir en este tiempo el mensaje x_k. En este caso, el mensaje x_k es recibido antes del vencimiento del *deadline* de w_1 y es entregado (Líneas 74-77). Ahora, el mensaje w_1 puede ser entregado conservando el orden causal. Hacemos notar que las restricciones de entrega en tiempo real a nivel inter-flujo son conservadas para el caso de mensajes causales *begin*.

4.5.3.2 Restricciones de entrega en tiempo real en mensajes *cut*

Los mensajes causales *cut* y *end* deben satisfacer el orden FIFO y el orden causal para ser entregados, consecuentemente, deben de preservar las restricciones de entrega a nivel intra-flujo e inter-flujo. Con el objetivo de explicar cómo preservamos las restricciones de entrega en tiempo real en el caso de mensajes causales *cut*, presentamos el siguiente escenario (figura 4.7).

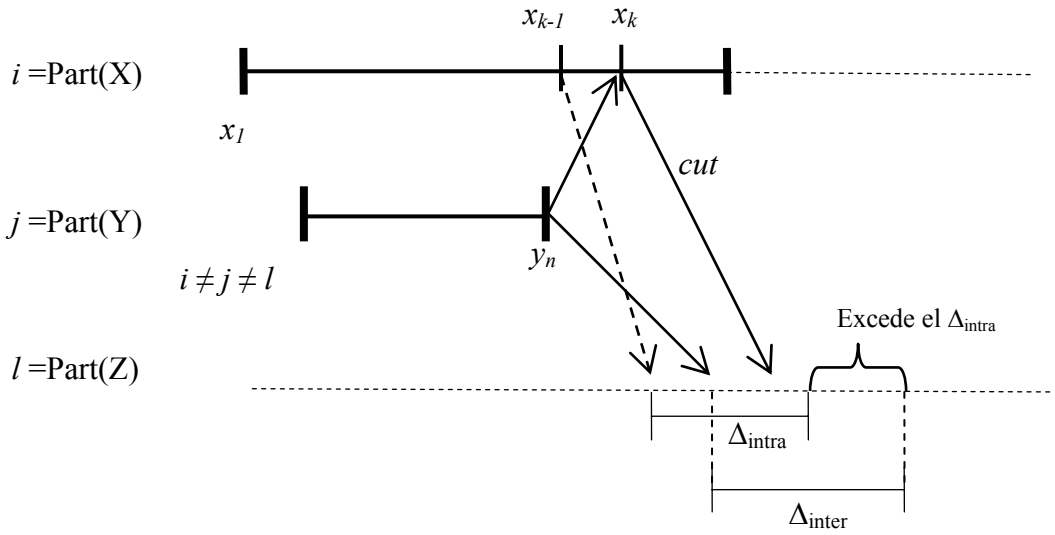

Figura 4.7. Restricciones de entrega en tiempo real en mensajes causales *cut*

El mensaje causal *cut* (x_k) lleva unida información de control acerca del mensaje y_n (Líneas 14-17), por lo tanto, tiene que preservar las restricciones de entrega en tiempo real a nivel inter-flujo con el mensaje y_n. Sin embargo, el mensaje x_k tiene otra restricción de entrega en tiempo real a nivel intra-flujo, la cual es con respecto a su predecesor inmediato x_{k-1}. Con la finalidad de preservar las restricciones de entrega en tiempo real a nivel intra-flujo e inter-flujo, se calcula el *deadline* del mensaje x_k con base en su predecesor inmediato x_{k-1} (líneas 74-77). Dando preferencia a las relaciones temporales a nivel intra-flujo. Si el *deadline* de x_k se calculara tomando como base el punto de referencia establecido por la entrega del mensaje y_n (figura 4.7), el mensaje x_k podría ser entregado en un tiempo mayor a su tiempo de vida a nivel intra-flujo, degradando la calidad de servicio de la aplicación, ver figura 4.7.

El último escenario por analizar es el formado por los mensajes causales *end*. En nuestro caso, el mensaje causal *end* debe preservar las restricciones de entrega a nivel intra-flujo siguiendo el procedimiento establecido para los mensajes FIFO, el cual hemos descrito en la sección 4.5.1 de este capítulo.

4.5.4 Especificación del algoritmo

Algoritmo 4. Especificación del mecanismo de sincronización de flujos continuos con restricciones de entrega en tiempo real.

1.	**Initially**
2.	$VT(p)[j] \leftarrow \varnothing \ \ \forall j$: 1…n
3.	*VTIME(p)[j]= big_value* $\forall j$: 1…n /* time vector used to save the deadline each message */
4.	$CI \leftarrow \varnothing$
5.	$Act=0$
6.	$last\_fifo \leftarrow \varnothing$
7.	*begin_deadline= big_value* /* deadline for begin messages initialized with a value big */
8.	*Time_by_default_added= 2* $*\Delta$
9.	**For each message** *m* **diffused by** *p* **with the process identifier** *i*
10.	**Send (** Input: TP =*begin* | *end* | *cut* | *fifo_p* **)**
11.	$VT(p)[i] \leftarrow VT(p)[i] + 1$
12.	If not (TP = = *fifo_p*) Then
13.	If not (TP = = *begin*) Then /* construction of the $H(m)$ for end and cut messages */
14.	$H(m) \leftarrow CI(p)$
15.	If (TP = = *end*) then
16.	$Act = 0$ /* indicate that the process *p* is inactive*/
17.	Endif
18.	Else /* construction of the $H(m)$ for *begin* messages */
19.	$Act = 1$ /* indicate that the process *p* is active*/
20.	$\forall (s,r) \in CI(p)$
21.	If $\exists (x,l) \in last\_fifo \mid s = x$ and $l > r$ then
22.	$CI(p) \leftarrow CI(p) / (s,r)$
23.	Endif
24.	$CI(p) \leftarrow CI(p) \cup \max\{ (x,l), (s,r) \}$
25.	$H(m) \leftarrow CI(p)$
26.	$last\_fifo \leftarrow \varnothing$
27.	Endif
28.	$CI(p) \leftarrow \varnothing$
29.	Else
30.	$H(m) \leftarrow \varnothing$
31.	Endif
32.	$m = (i, t = VT(p)[i], TP, H(m), data)$
33.	Sending(m)

34.	**Endfor**
35.	**For each message *m* received by *p* with process identifier *j***
36.	*Receive*(*m*) in *p* with $i \neq j$ and *m*= (*i*, *t* = *VT*(*p*)[*i*], *TP* , *H*(*m*),*data*)
37.	If *t* = *VT*(*p*)[*i*] +1 Then
38.	If (*reception_time_message*>*VTIME*(*p*)[*i*] and *TP*!=*Begin*)then /*messages out his deadline */
39.	Discard(*m*) /* discarded message */
40.	*VT*(*p*)[*i*] = *t*
41.	*VTIME*(*p*)[*i*] = *VTIME*(*p*)[*i*] + Δ
42.	Else /* else del If (*reception_time_message* > *VTIME*(*p*)[*i*] and *TP*!=*Begin*) then */
43.	If (*TP* ! = *fifo_p*) then
44.	If (TP = begin) then
45.	*Begin_deadline_measure*(*m*)
46.	If (*begin_message_reception_time* > *begin_deadline*) then
47.	*Discard*(*m*) /* discarded message */
48.	*VT*(*p*)[*i*] = *VT*(*p*)[*i*] + 1
49.	VTIME(p)[i] = begin_deadline + Δ_{life_time}
50.	Else /* check the causal dependencies of the begin message */
51.	If not ($t' \leq VT(p)[l]) \forall (l, t') \in H(m)$ then /* causal delivery condition*/
52.	*Put_message_in_queue*(*m*,*begin_deadline*) /*queue message */
53.	Else
54.	*Deliver*(*m*)
55.	*VTI*(*p*)[*i*] = 1
56.	*VT*(*p*)[*i*] = *VT*(*p*)[*i*] + 1
57.	*Update_CI_and_Last_fifo*(*m*)
58.	VTIME(p)[i] = Current_time + Δ
59.	Endif /* endif de If not ($t' \leq VT(p)[l]) \forall (l, t') \in H(m)$ */
60.	*Discard_message_out_its_deadline*(*m*)
61.	Endif /* endif of If (begin_message_reception_time > begin_deadline) */
62.	Else /*else of If (TP=begin) check the causal dependencies for *cut* and *end* */
63.	If not ($t' \leq VT(p)[l]) \forall (l, t') \in H(m)$ then /* causal delivery condition*/
64.	Put_message_in_queue(m,*VTIME*(*p*)[*i*]) /*queue message */
65.	Else
66.	Deliver(m)
67.	*VT*(*p*)[*i*] = *VT*(*p*)[*i*] + 1
68.	Update_CI_and_Last_fifo (m)
69.	VTIME(p)[i] = VTIME(p)[i] + Δ
70.	Endif /* endif of If not ($t' \leq VT(p)[l]) \forall (l, t') \in H(m)$ */
71.	Discard_message_out_its_deadline(m)
72.	Endif /* endif of If (TP = begin) then */
73.	Else /* else of If (*TP* ! = *fifo_p*) then */
74.	Deliver(m)
75.	*VT*(*p*)[*i*] = *VT*(*p*)[*i*] + 1

```
76.                 Update_CI_and_Last_fifo ( m )
77.                 VTIME(p)[i] = VTIME(p)[i] + Δ
78.             Endif           /* endif of   If ( TP ! = fifo_p ) then */
79.         Endif       /* endif of  If (reception_time_message > VTIME(p)[i] and TP!=Begin) then */
80.       Else  /* else of If  t =  VT(p)[i] +1 Then  */
81.         received_message_wait_time = VTIME(p)[i] + (( t – VT(p)[i]) * Δ – time_defect_added )
82.         If ( current_time > received_message_wait_time ) then
83.             If (TP! = fifo_p ) then
84.                 If not ( t' ≤ VT(p)[l])∀( l, t' ) ∈ H(m) then  /*causal delivery condition*/
85.                     VT(p)[l] = t'      /*update of vectors*/
86.                 Endif       /* endif  de If not ( t' ≤ VT(p)[l]) ∀ ( l, t' ) ∈ H(m)  */
87.             Endif   /* end of If (TP! = fifo_p ) then */
88.             Deliver(m)
89.             VT(p)[ i ] = t
90.             VTIME(p)[i] = received_message_wait_time + Δ
91.             Update_CI_and_Last_fifo(m)
92.          Else
93.             Put_message_in_queue( m, received_message_wait_time)
94.         Endif  /* end of If ( current_time > received_message_wait_time ) then */
95.      Endif   /* end of If  t =  VT(p)[i] +1 Then  */
96.     Endfor

97.     Discard_message_out_deadline( m )  {    /* message received  m = (i, t = VT(p)[i], TP , H(m),data) */
98.     ∀(l, t' )∈ H(m)
99.         If not ( t' ≤ VT(p)[l]) then    /* causal delivery condition*/
100.                msg_res = t' – VT(p)[l]
101.                msg_val = (VTIME(p)[l] - current_time) / play_time
102.                If (msg_res > msg_val )  then /*Detection of lost message*/
103.                        msg_dis = msg_res – msg_val
104.                        VT(p)[l] = VT(p)[l] + msg_dis      /*update of vectors*/
105.                Endif
106.        Endif
107.    }
108.    Begin_deadline_measure( m ) {
109.        Δinter-st = Δinter -  Δintra
110.        ∀(l, t' )∈ H(m)
111.            If  ( t' ≤ VT(p)[l]) then    /* causal delivery condition*/
112.                If (begin_deadline>VTIME(p)[l] ) then
113.                    begin_deadline = VTIME(p)[l]
114.                Endif
115.            Endif
116.        begin_deadline = begin_deadline +  Δinter-st
117.    }
```

```
118. Update_CI_last_fifo ( m ) {    /*updating the CI and last FIFO with the message delivered */
119.  If  not (TP =  fifo_p )  Then   /* update the CI and last_fifo */
120.       If ∃ (s,r) ∈ CI(p) | i = s   then
121.          CI(p) ← CI(p) \ (s,r)
122.       Endif
123.       CI(m) ← CI(m) ∪ { ( i, t) }
124.       ∀(l, t' ) ∈ H(m)                /*Updating CI(p)with a most recent message*/
125.          If ∃ (s,r) ∈ CI(p) | l =s and r ≤ t'  then
126.             CI(p) ← CI(p)  \ ( s, r )
127.          Endif
128.          If ∃ (x,l) ∈ last_fifo(p) | x = l  and l ≤ t'  then
129.             last_fifo(p) ← last_fifo(p) \ (x,l)
130.          Endif
131.       If ( Act = 1 and  not( TP = cut) and  not ( TP = begin)  then
132.          Send( cut )     /*sending a cut message by the deliver of a end message*/
133.       Endif
134.    Else   /* else of  If  not (TP =  fifo_p ) */
135.       If  ∃  (x, l ) ∈ last_fifo(p) | x = i  then    /*Updating last_fifo(p) with a most recent message*/
136.          last_fifo(p) ←  last_fifo(p) / ( x, l)
137.       Endif
138.       last_fifo(p) ← last_fifo(p) ∪ ( i, t)   /* adding the last fifo*/
139.  Endif   /* If  not (TP =  fifo_p )  Then */
140. }
141. Put_message_in_queue ( m , wait_time_max ) { /*message received  m=(i,t=VT(p)[i],TP,H(m),data) */
142.      m' = m ∪ ( wait_time_max )   /* adding to message the wait time in the queue*/
143.      queue = queue ∪ m`     /* put the message in the queue*/
144. }
145.  Check_timer {    /* message structure  m` = (i, t = VT(p)[i], TP , H(m),data, time) */
146.   ∀  m` ∈ queue
147.     If  ∃ time ∈ m` | time > current_time  then /*delivering a message with a deadline bigger that current time */
148.           ∀(l, t' ) ∈ H(m)
149.              If ( t` > VT(p)[l] ) then
150.                 VT(p)[l] = t`
151.              Endif
152.           Delivery( m` )
153.           VT(p)[i] = t
154.           VTIME(p)[i] = VTIME(p)[i] + Δ
155.           queue = queue / m`
156.           update_CI_last_fifo(m)
157.      Endif
158. }
```

Capítulo 5

Emulación y Resultados

En este capítulo, describimos la emulación del mecanismo de sincronización de flujos continuos tolerante a la pérdida de mensajes con restricciones de entrega en tiempo real, el cual es la unión del mecanismo de sincronización de flujos continuos desarrollado en [MOR05] con el mecanismo de recuperación hacia adelante y el método para el cálculo de tiempo de vida distribuido, descritos en los capítulos 3 y 4. El algoritmo completo se encuentra en el Apéndice 1. El escenario utilizado para realizar la ejecución del mecanismo representa un sistema distribuido formado por tres participantes. Los participantes utilizan el algoritmo presentado en el apéndice 1 para lograr la sincronización de flujos continuos en canales de comunicación no fiables y asíncronos. Emulamos condiciones de una red WAN usando el emulador de red NIST Net. Esta herramienta nos permite experimentar con varios parámetros de red, tales como retraso de paquetes, *jitter*, mensajes perdidos, duplicación de mensajes, entre otros. Nosotros establecemos a través de NistNet diferentes condiciones de red (pérdida y retraso de mensajes) para cada canal de comunicación entre los participantes.

La principal contribución de la emulación del algoritmo desarrollado, es la comprobación de que el mecanismo de recuperación causal hacia delante, presentado en el capítulo 4, lleva a cabo la sincronización de flujos continuos en *canales de comunicación no fiables* con un error de sincronización tolerable para la aplicación en cuestión. Los resultados obtenidos muestran cómo el mecanismo de recuperación causal hacia adelante, realiza la sincronización de los flujos continuos en presencia de mensajes causales perdidos (*begin*, *end*, *cut*), con un error de sincronización (entre los flujos continuos de audio y video) aceptable para la aplicación e imperceptible para la vista humana.

Por otra parte, nosotros consideramos restricciones de entrega en tiempo real (a nivel intra-flujo e inter-flujo) utilizando el método distribuido para el cálculo de tiempo de vida aplicado a flujos continuos, descrito en el capítulo 4, sección 4.5. Cada participante determina de manera independiente, si un mensaje ha sido recibido después de la expiración de su tiempo de vida. Cuando un mensaje sufre un retraso mayor a su tiempo de vida es descartado. De acuerdo a diversos estudios realizados en [WIJ96], nosotros consideramos un *tiempo de vida* de 70ms a nivel intra-flujo y a nivel inter-flujo contemplamos un máximo de 120ms. Ambos parámetros proporcionan un desempeño aceptable de la aplicación.

El capítulo está estructurado de la siguiente forma: En la sección 5.1, describimos de forma breve la herramienta utilizada para emular una red WAN. La sección 5.2, describe el escenario de prueba, las condiciones de los canales de comunicación y finalmente las características de los flujos enviados. Los resultados de la ejecución del mecanismo de flujos continuos son presentados en la sección 5.3.

5.1 Emulador de red NistNet

El análisis del correcto funcionamiento de protocolos de red y aplicaciones distribuidas es una tarea difícil y compleja. Hoy en día, existen tres herramientas para la prueba y experimentación de algoritmos y protocolos distribuidos, continuación describiremos de manera breve tales herramientas:

- **Simuladores de red:** esta herramienta proporciona un ambiente sintético de una red. Usualmente los simuladores requieren representar de forma lógica los componentes de una red (nodos) y convertir el código de la aplicación para que pueda ser ejecutado en un ambiente de simulación. Los simuladores de red permiten fácilmente representar una topología de red compleja. Sin embargo, las condiciones de los canales de la red están basadas en algunas suposiciones de tráfico y carga de datos, en consecuencia, ejemplifica una red real de forma pobre. Otra desventaja de los simuladores de red es que el tiempo de simulación aumentará considerablemente conforme la complejidad de la red simulada incremente.

- **Emuladores de red:** los emuladores permiten evaluar el funcionamiento de una aplicación distribuida en un ambiente de red real. Usualmente una red LAN es configurada para imitar una red WAN. Los emuladores de red proporcionan las ventajas de la simulación (un ambiente controlable y reproducible) junto con las ventajas de *testing live* (código real en un ambiente real).

- **Testing live**: en esta herramienta las pruebas de protocolos o aplicaciones distribuidas se realizan sobre redes físicas, tales como Internet. Este tipo de pruebas, resulta costoso en términos de tiempo y recursos. Además, no se pueden generar de forma fácil diversas condiciones de red y tráfico dinámico.

Para la realización de nuestro escenario de prueba utilizamos un emulador de red, llamado NistNet.

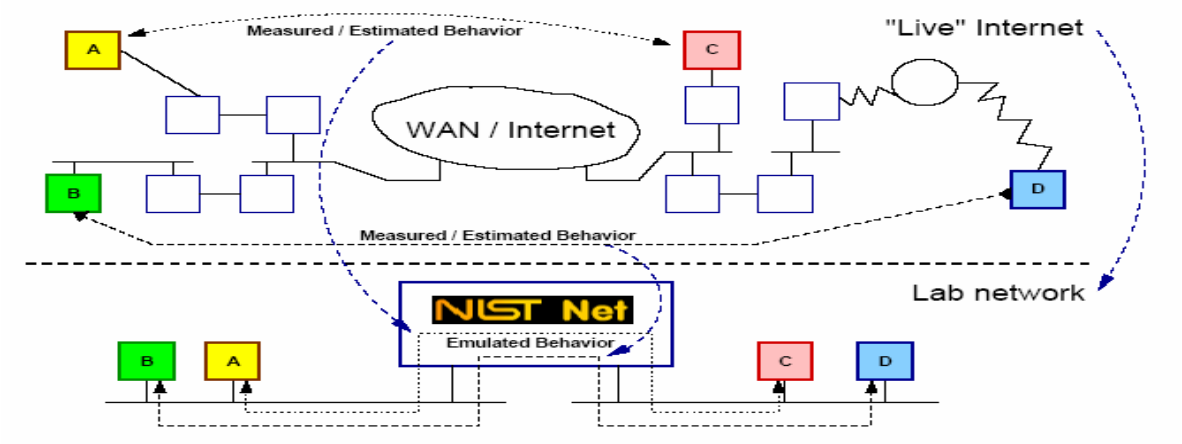

Figura 5.1. NisNet como una red WAN

NistNet nos permite configurar una PC como un *router* para emular una red WAN de un solo salto (figura 5.1). Esta herramienta nos proporciona la facilidad de experimentar con varios parámetros de red, tales como retraso de paquetes, *jitter*, ancho de banda, congestión, mensajes perdidos, duplicación de mensajes, entre otros. Mediante Nist Net establecemos para cada canal de comunicación, diferentes condiciones de red, principalmente distintas características de pérdida y retraso de mensajes. NisNet trata a cada flujo de datos de forma independiente. Un número considerable de flujos puede ser cargado a la vez, cada uno con distintas condiciones de red (pérdida y retraso de mensajes). Los flujos de datos pueden ser agregados y cambiados de forma manual durante la ejecución de la aplicación. Para agregar un flujo de datos en NistNet, se debe de especificar su origen y destino indicando el tipo de protocolo del paquete (UDP, TCP, ICMP, IGMP, IPIP, entre otros) que será afectado por los distintos parámetros de red configurados por el usuario.

La pérdida de mensajes en NistNet es emulada a tráves de una versión parametrizada del método de pérdida aleatoria derivado (*Derivative Random Drop*, DRD). DRD elimina paquetes o mensajes con una probabilidad que, después de que un umbral mínimo es alcanzado, incrementa linealmente con el crecimiento del buffer de mensajes encolados. NistNet establece inicialmente por default los valores para el máximo y mínimo umbral para DRD. Un buffer con una longitud de mensajes encolados es asociado a cada flujos de datos a transmitir, para mas detalles de cómo funciona DRD consultar [CAR03].

A continuación describimos a detalle el escenario utilizado para la emulación del mecanismo de sincronización de flujos continuos en tiempo real tolerante a fallas.

5.2 Descripción del escenario de prueba

Se realizó la emulación del algoritmo de sincronización de flujos continuos presentado en el apéndice 1, usando el escenario mostrado en la figura 5.2. El escenario representa un sistema distribuido formado por 3 participantes dispersos geográficamente. Cada participante ejecuta el algoritmo de sincronización de flujos continuos tolerante a fallas, desarrollado en este trabajo, para enviar y recibir datos.

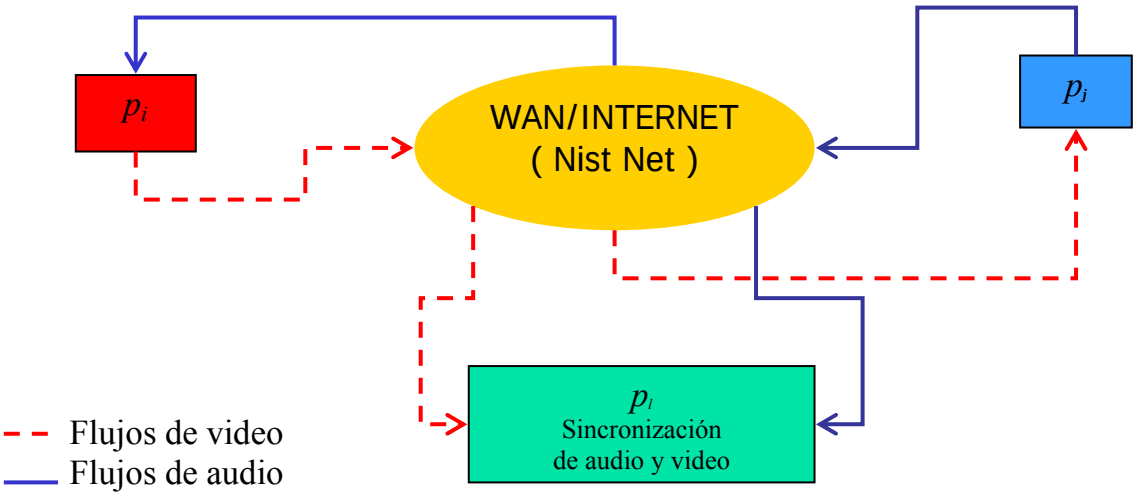

Figura 5.2. Escenario de prueba

Emulamos condiciones de una red WAN, como Internet, sobre nuestra LAN usando el emulador de red NIST Net, el cual fue descrito en la sección anterior. Esta herramienta nos permite evaluar nuestro algoritmo en canales de comunicación con diversas características propias de una red real, tales como retraso de paquetes, jitter y mensajes perdidos, entre otros.

A través de Nist Net establecemos para cada canal de comunicación, entre los participantes del escenario de prueba (figura 5.2), diferentes condiciones de red descritas a continuación en la tabla 5.1:

Source	Dest	Retraso (ms)	Pérdida %
P_i	P_l	80 ± 40	5
P_j	P_l	300 ± 50	5-15

Tabla 5.1. Condiciones de red en los canales de comunicación

El canal de comunicación entre los participantes p_i y p_l tiene un retraso aleatorio que varia de 40ms a 120ms con una tasa de pérdida del 5%. El retraso sufrido en el canal de comunicación entre p_j y p_l es de 250ms a 350ms con una tasa de pérdida entre el 5% y 15%.

El canal de comunicación entre p_i y p_j es fiable (no hay pérdida de mensajes) y el retraso de los mensajes es mínimo.

Los flujos transmitidos por los participantes tienen una duración de aproximadamente 15 segundos con características descritas en la tabla 5.2:

Flujo	Fuente	Mensajes	Tamaño (bytes)	Rate (Kbs)
Video	P_i	500	10,000	300
Audio	P_j	500	8,000	300

Tabla 5.2. Características de los flujos transmitidos

El participante p_i transmite un flujo de video (con formato MJPEG) representado por un solo intervalo, el cual está formado por paquetes (mensajes) con un tamaño promedio de 10kb. La carga de datos transmitida por un mensaje equivale a un frame del video. La transmisión de los mensajes es a razón de 25 mensajes por segundo, lo que equivale a enviar 25 frames por segundo. El participante p_j difunde un flujo de audio formado por paquetes con un tamaño aproximado de 8kb. El flujo de audio transmitido es dividido en pequeños intervalos, el tamaño de los intervalos es determinado de forma aleatoria. El motivo de transmitir el flujo de audio en varios flujos o intervalos es tener distintos cortes (mensajes *cut*) en el intervalo de video provocados por la entrega de mensajes causales *end*. Los mensajes *cut* son mensajes de control utilizados por el mecanismo para informar acerca de la segmentación de los intervalos. A continuación, presentamos de forma detallada cómo los mensajes causales proporcionan información para calcular y corregir el error de sincronización sufrido durante la ejecución del algoritmo de sincronización de flujos continuos.

5.3 Aspectos evaluados

Los principales aspectos a estudiar en la emulación del algoritmo de sincronización de flujos continuos en tiempo real son:

- Verificar que el mecanismo de detección y recuperación de mensajes perdidos permita llevar a cabo la sincronización de flujos continuos aún en presencia de mensajes perdidos.

- Evaluar el error de sincronización experimentado durante la ejecución de la aplicación. En nuestro caso, el error de sincronización permisible es igual al retraso máximo que puede tolerar un mensaje causal a nivel inter-flujo. Por ejemplo, para

flujos de audio y video en una videoconferencia, el retraso máximo que puede tolerar un mensaje de video a nivel inter-flujo es de 100ms [RAV93].

- Comprobar que el método distribuido para el cálculo de tiempo vida, descrito en el capítulo 4 (sección 4.5), mantiene (en forma independiente a cada proceso) las restricciones de entrega en tiempo real a nivel intra-flujo e inter-flujo.

Para describir como calculamos el error de sincronización, presentamos el siguiente escenario (figura 5.3), el cual representa a nivel de intervalos la ejecución del escenario de prueba mostrado en la figura 2.

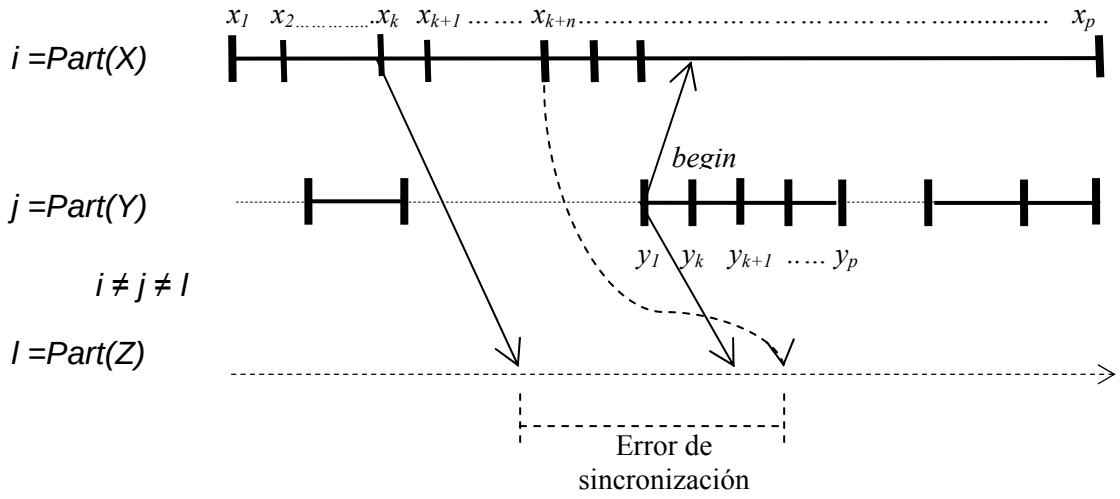

Figura 5.3. Escenario de prueba representado en intervalos

En este ejemplo (figura 5.3), sobre el participante *l* el mensaje causal y_1 es puesto en espera debido a que es recibido antes del mensaje x_{k+n}, el cual lo precede causalmente. Con la información de control unida al mensaje y_1 acerca de x_{k+n} podemos calcular el error de sincronización que existe entre los flujos de audio y video en ese instante de tiempo.

En nuestro caso, el error de sincronización es igual a:

$$\text{Error de sincronización} = (x_{k+n} - x_k) * \textit{tiempo de reproducción}$$

Donde:

- x_{k+n} representa el último mensaje *FIFO* recibido antes del envío del mensaje causal *begin* (y_1).

- x_k determina el último mensaje recibido del flujo de video.

- *tiempo de reproducción* expresa el periodo de tiempo que tarda en reproducirse un mensaje, el cual es aproximadamente 40ms.

A continuación presentamos los resultados obtenidos de las ejecuciones realizadas con el escenario de prueba descrito previamente.

5.4 Resultados

Utilizando el escenario presentado en la figura 1 realizamos los siguientes experimentos. Los experimentos consistieron en enviar y recibir los flujos de datos continuos (audio y video), utilizando el mecanismo de sincronización tolerante a fallas con restricciones de entrega en tiempo real, propuesto en esta tesis. Las pruebas realizadas consideran inicialmente canales de comunicación con un porcentaje de pérdida del 5%, posteriormente 10% y finalmente con un porcentaje de hasta 15% de mensajes perdidos.

La figura 5.4 muestra el error de sincronización experimentado por el participante p_l antes de aplicar el mecanismo de sincronización (línea punteada) y después de aplicar el mecanismo de sincronización tolerante a la pérdida de mensajes (línea continua), considerando canales de comunicación con 5% de pérdida. El error de sincronización fue calculado sobre la recepción de los mensajes causales *begin* como se describió en la sección 5.2 de este capítulo, al detectar la violación del orden causal, el mecanismo retrasa la entrega del mensaje hasta la expiración de su tiempo de vida. De esta forma, el mecanismo nivela el error de sincronización experimentado en el proceso p_l. Como podemos ver en la gráfica (figura 5.4), el error de sincronización sufrido, sin aplicar el algoritmo de sincronización (línea punteada), en diversos puntos es de 160 ms percibiendo un error de sincronización máximo de 240ms. Utilizando el mecanismo de sincronización tolerante a fallas, desarrollado en esta tesis, el error de sincronización máximo (línea continua) percibido por el participante p_l es de 80ms.
Los resultados obtenidos considerando un canal de comunicación con 10% de pérdida de mensajes son mostrados en la figura 5.5. En este caso, el error de sincronización máximo sin aplicar el mecanismo de sincronización es de 160ms. Utilizando el mecanismo de sincronización de flujos continuos tolerante a fallas, el error de sincronización máximo disminuye hasta los 120ms.

El último experimento realizado considera un canal de comunicación con 15% de pérdida (figura 5.6). El error de sincronización máximo, experimentado por el participante p_l, sin aplicar el mecanismo de sincronización en diversos puntos es mayor a 160ms, sufriendo un máximo de 440ms. Aplicando el algoritmo de sincronización tolerante a fallas, el error de sincronización en la mayoría de los puntos es menor a 120ms.

En nuestro caso, el error de sincronización máximo que puede tolerar la aplicación es de 120ms [PER03]. Por lo tanto, el error sincronización sufrido por el participante p_l sin utilizar el mecanismo de sincronización tolerante a fallas, representa una considerable degradación a la aplicación en la sincronización de los flujos de audio y video, siendo perceptible para el sentido de la vista.

Utilizando el mecanismo de sincronización de flujos continuos en tiempo real tolerante a fallas, con una *distancia causal* de 3 y una *redundancia en el tipo de paquete* de 5. El error de sincronización experimentado es menor o igual a 120, el cual es aceptable para la aplicación e imperceptible para el sentido de la vista. Los resultados obtenidos en las distintas pruebas muestran como el algoritmo de sincronización de flujos continuos tolerante a fallas, corrige el error de sincronización entre los flujos de audio y video provocado por las condiciones de retraso y pérdida de los canales de comunicación.

La tabla 5.3 muestra el número de mensajes que son descartados por el mecanismo de sintonización con la finalidad de corregir el error de sincronización. Así como el error de sincronización promedio, experimentado por el participante p_l, utilizando el algoritmo de sincronización no causal (SNC) y el error de sincronización promedio aplicando el algoritmo de sincronización causal (SC).

Tabla 5.3. Número de mensajes descartados y error de sincronización promedio en la ejecución de los algoritmos de sincronización no causal (SNC) y causal (SC)

% pérdida	**5 %**		**10 %**		**15 %**	
	SNC	**SC**	**SNC**	**SC**	**SNC**	**SC**
Paquetes descartados (%)	---	4.2	---	5.2	---	6.5
Error de sincronización promedio (ms)	48	22	71	26	93	37

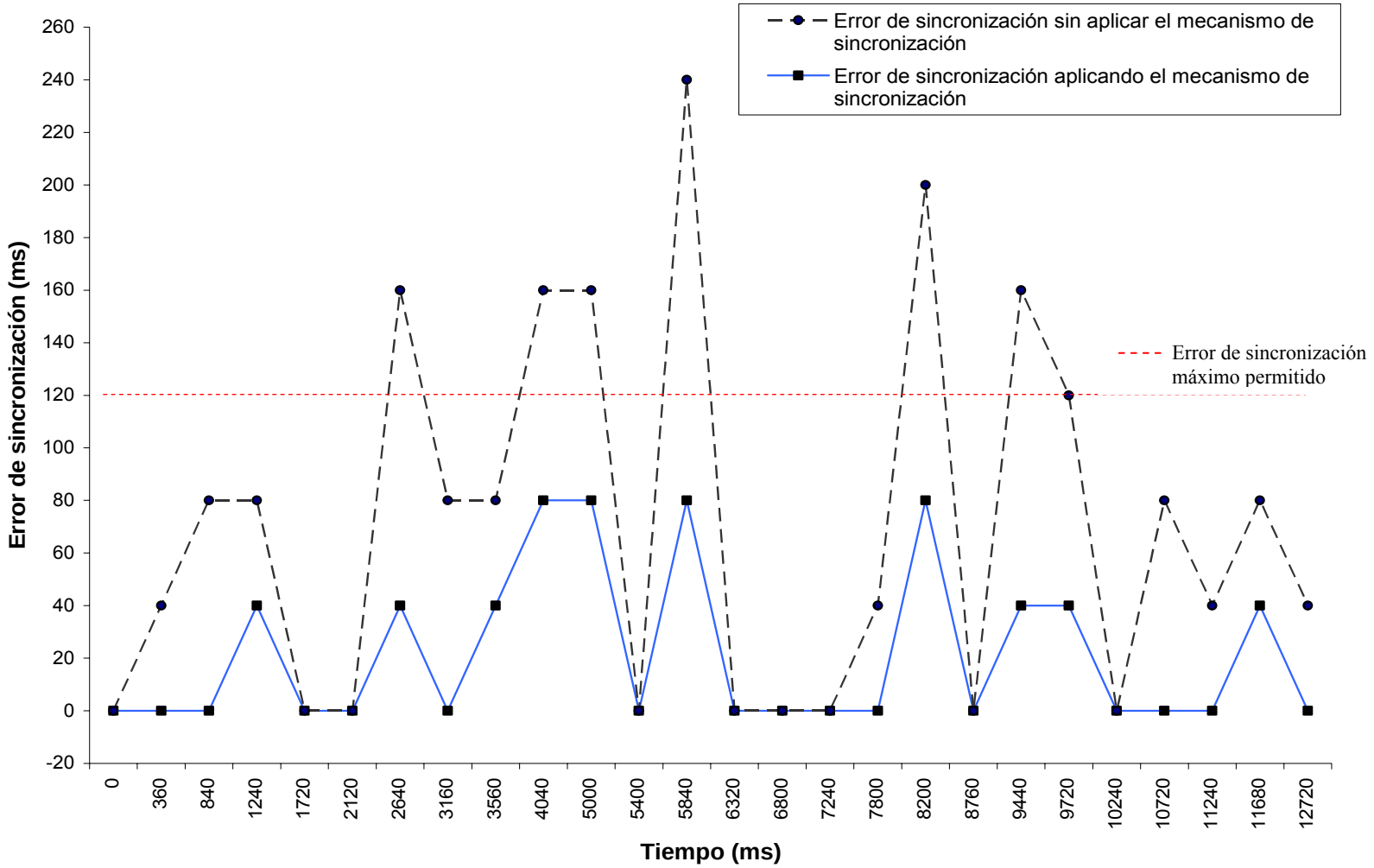

Figura 5.4. Error de sincronización inter-flujo antes y después de aplicar el algoritmo de sincronizaron causal tolerante a la pérdida de mensajes (5% pérdida)

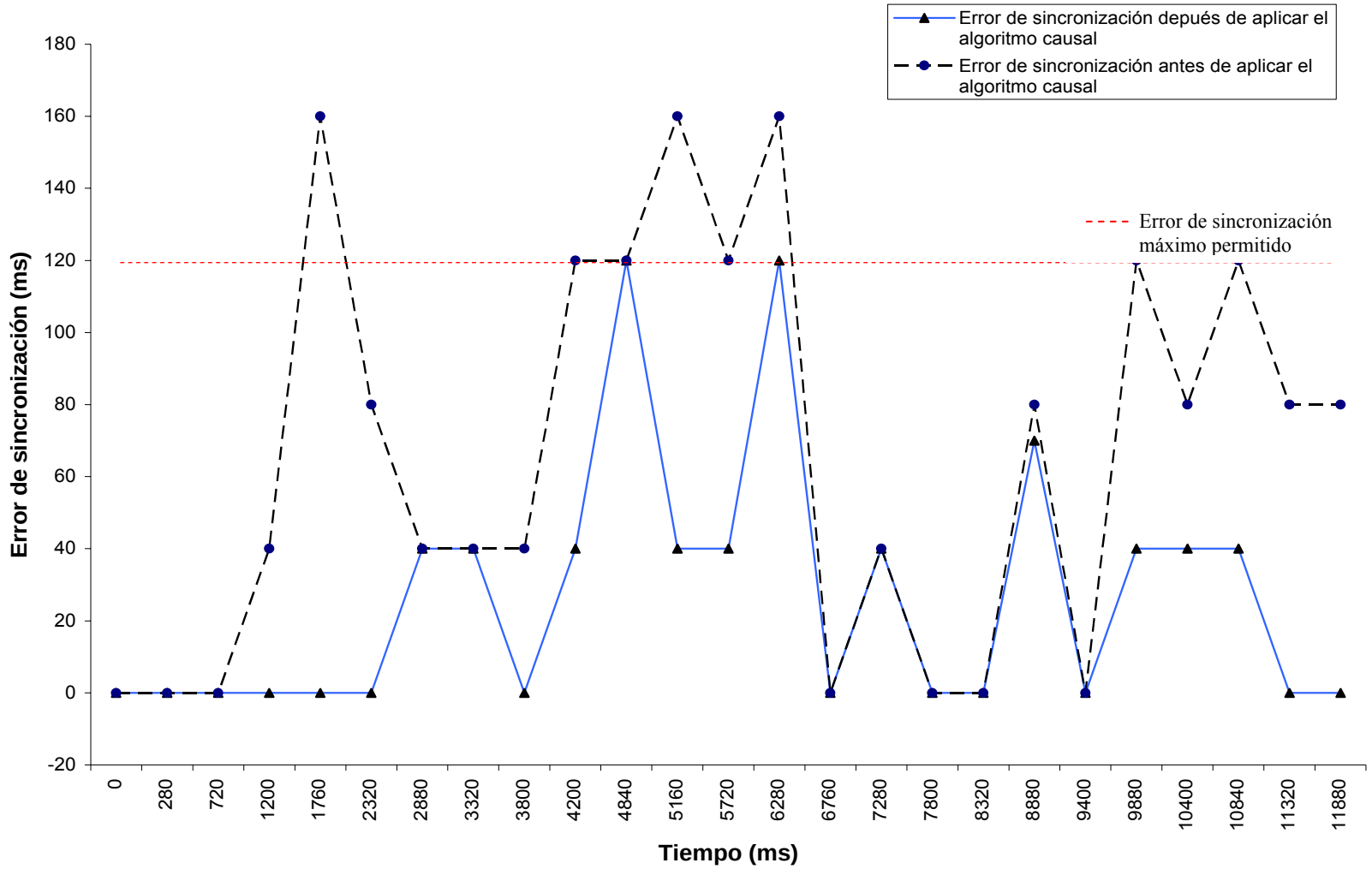

Figura 5.5. Error de sincronización inter-flujos antes y después de aplicar el algoritmo de sincronización causal tolerante a la pérdida de mensajes (10 % pérdida)

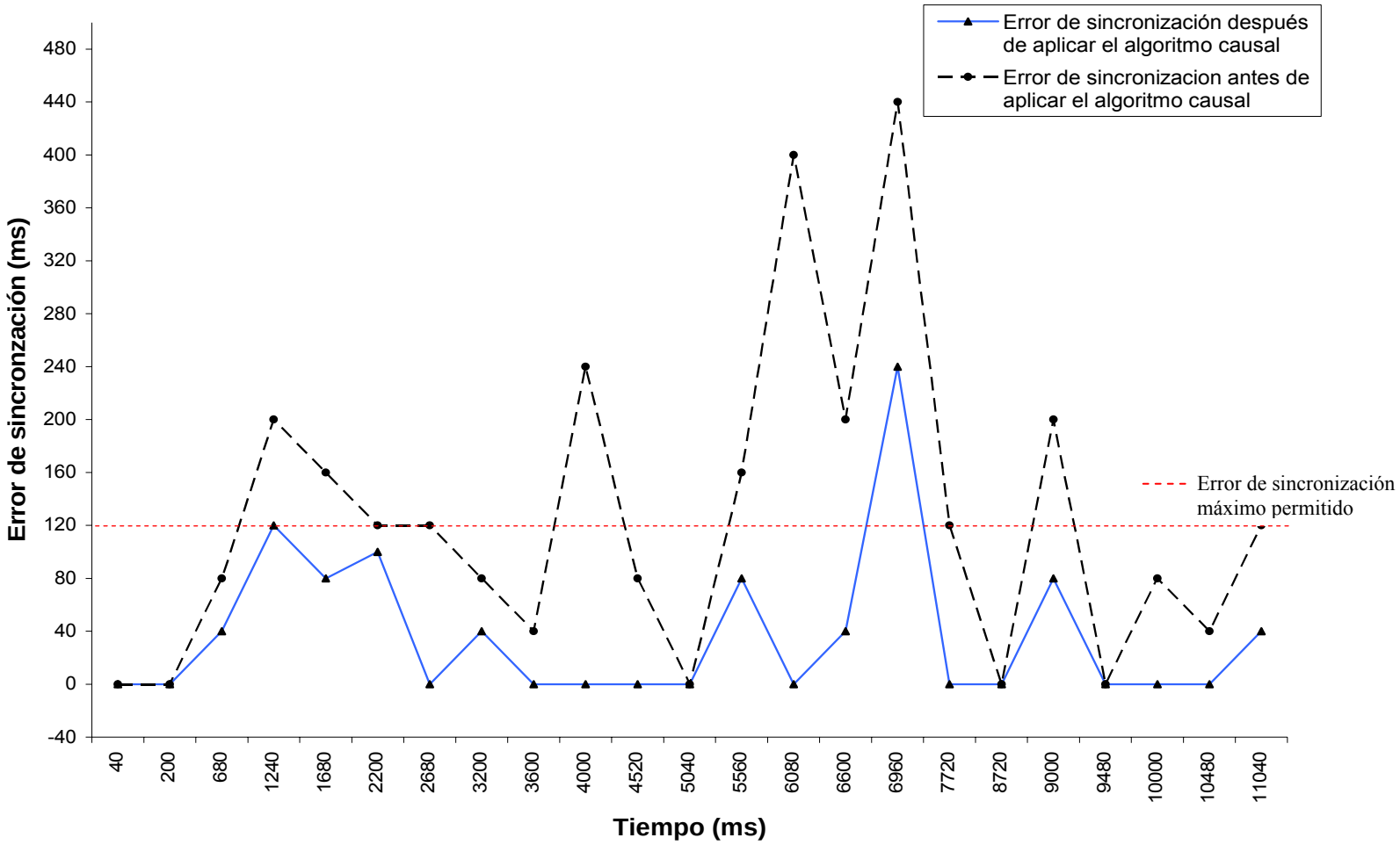

Figura 5.6. Error de sincronización inter-flujo antes y después de aplicar el mecanismo de sincronización causal tolerante a la pérdida de mensajes (15 % pérdida)

Capítulo 6

Conclusiones

En esta tesis fue presentado un mecanismo para la sincronización de flujos continuos sobre canales de comunicación no fiables, considerando restricciones de entrega en tiempo real. Para lograr la sincronización de flujos continuos en presencia de mensajes perdidos, desarrollamos un mecanismo de recuperación causal hacia adelante, inspirado en el concepto de redundancia, el cual evita la retransmisión de los mensajes perdidos. Dos tipos de redundancia fueron utilizados para llevar a cabo la recuperación causal hacia adelante: *redundancia sobre el tipo de paquete* y *redundancia sobre la información de control enviada*. La *redundancia sobre la información de control enviada* unida a cada mensaje es adaptable dinámicamente al comportamiento del sistema; es decir, sólo es agregada información de control a un mensaje cuando el comportamiento del sistema lo requiere capítulo 3. Nosotros mostramos que utilizando los dos tipos de redundancia es posible recuperar hacia adelante los mensajes perdidos, y consecuentemente, lograr la sincronización de los flujos continuos evitando la retransmisión, capítulos 4 y 5. De acuerdo al estado del arte revisado, nuestro trabajo es el primero en proponer un mecanismo de detección y recuperación hacia delante de mensajes perdidos con control de causalidad capítulo 3.

Por otra parte, al contemplar las restricciones de entrega en tiempo real se propuso un *método distribuido para el cálculo de tiempo de vida* de los mensajes transmitidos a nivel intra-flujo e interflujo, capítulo 3. El método propuesto determina si un mensaje ha sufrido un retraso mayor a su tiempo de vida. Cuando un mensaje sufre un retraso mayor a su tiempo de vida es considerado no útil a la aplicación y es descartado. Hacemos notar que el cálculo del tiempo de vida de cada mensaje es realizado sin utilizar una referencia global (*wall clock*). Cada participante determina, utilizando su reloj local físico, de manera independiente y de acuerdo a la vista parcial que posee del sistema, si un mensaje es útil a la aplicación en cuestión o si debe ser descartado.

El mecanismo de sincronización de flujos continuos tolerante a la pérdida de información con restricciones de entrega desarrollado en esta tesis, es adecuado para sistemas distribuidos en tiempo real debido a que posee las siguientes características:

- Realiza la detección y recuperación hacia adelante de los mensajes perdidos en forma distribuida, evitando la retransmisión.

- Evita el uso de una referencia global para determinar si un mensaje ha sufrido un retraso mayor a su tiempo de vida.

- No necesita previo conocimiento de la duración los datos a transmitir ni un pre-etiquetado de la información.

- La sincronización de los flujos continuos, recuperación de los mensajes perdidos, y el cálculo del tiempo de vida de los mensajes transmitidos a nivel intra-flujo e inter-lujo, es hecho en línea y de forma distribuida (local e independiente en cada participante).

A continuación describiremos las principales aportaciones realizadas en el desarrollo del tema de tesis.

6.1 Aportaciones

Las principales contribuciones de esta investigación se resumen a continuación:

- La primera se centra en la aportación de un mecanismo de recuperación causal hacia adelante, el cual nos permite hacer la sincronización de los flujos continuos en canales de comunicación no fiables. Dentro del desarrollo del mecanismo de recuperación causal hacia adelante se introdujo el concepto de *distancia causal*. La *distancia causal* establece el número de veces que la información de control es enviada en el sistema. El mecanismo de recuperación utiliza el concepto de *distancia causal* para llevar a cabo la recuperación hacia delante de los mensajes perdidos. El algoritmo de recuperación causal hacia adelante es adecuado para sistemas en tiempo real, debido a que tiene la característica de recuperar los mensajes perdidos hacia delante, evitando la retransmisión y en forma distribuida.

- La segunda contribución realizada en esta investigación fue el desarrollo de un método distribuido para el cálculo de tiempo de vida de los mensajes transmitidos sin utilizar un reloj global. El método distribuido para el cálculo de tiempo de vida permite contemplar restricciones de entrega en tiempo real a nivel intra-flujo e interflujo, de manera local e independiente, en cada participante del sistema.

6.2 Trabajo futuro

El trabajo presentado puede ser extendido en algunas direcciones. A continuación mencionamos algunas direcciones a seguir:

- Un aspecto a desarrollar se ubica en la forma de calcular el tiempo de vida que es necesario contemplar para establecer el *deadline* de un mensaje. El método desarrollado podría calcular de forma dinámica, en función del retraso sufrido por

los mensajes recibidos, el tiempo de vida necesario para determinar el deadline del siguiente mensaje a recibir. El calcular el tiempo de vida de acuerdo a las condiciones de la red puede mejorar la sincronización de los flujos.

- Algunas aplicaciones multimedia como teleconferencias, telefonia IP, entre otras requieren sincronizar flujos de datos heterogéneos (texto, audio, video, imagenes), por lo tanto, los mecanismo de recuperación hacia adelante y el método para el cáculo de tiempo de vida distribuido pueden ser extendidos para considerar la sincronización de datos discretos como texto, animaciones, imágenes, entre otros sobre canales de comunicación no fiables.

Actualmente, se encuentra en investigación y desarrollo la sincronización de datos heterogéneos (datos discretos y datos continuos) sobre canales de comunicación fiables, en la tesis de maestría titulada: “Algoritmo para la sincronización de flujos multimedia”.

Apéndice 1

Algoritmo 5 de sincronización de flujos continuos tolerantes a fallas con restricciones de entrega en tiempo real.

1.	**Initially**
2.	*VT*(*p*)[*j*] = 0 ∀ *j*: 1…n /* vector timeclock */
3.	*VTI*(*p*)[*j*] = 0 ∀ *j*: 1…n /* states vector used to detect the lost of a begin or cut message */
4.	*VTIME(p)[j]= big_value* ∀ *j*: 1…n /* time vector used to save the deadline each message */
5.	*CI*(*p*)←∅ /* message that is not ensured by participant *p* of being delivered in a causal order */
6.	*Act*=0 /* indicate that process *p* is inactive */
7.	*Last_fifo*(*p*)←∅ /* last FIFO message delivered */
8.	*copies_of_cut_message = num_cop* /* copy numbers of message cut */
9.	*con*=0 /* account of copies of begin message */
10.	*causal=0* /* if causal is equal 1,a fifo_p or cut copy message is used as causal */
11.	*begin_deadline= big_value* /* deadline for begin messages initialized with a value big */
12.	*last_cop_cut* ←∅ /* last cop_cut or cut message received */
13.	*time_default_added= 2* * Δ
	For each *m* message diffused by *p* with process identifier *i*
14.	***Send*** (*Input*: *TP = begin* | *end* | *cut* | *fifo_p* | *cop_cut*))
15.	*VT*(*p*)[*i*] ←*VT*(*p*)[*i*] +1 /* account of send messages by process *p* */
16.	*con* = (*TP = fifo_p*) ? *con* +1 : *con*
17.	If not (*TP = fifo_p* and *con* > *num_cop*) Then
18.	If not (*TP = begin* or *TP = fifo_p* or *TP* != *cop_cut*) Then
19.	H(m)_Construction (*CI*(*p*), increment) /*Construction of the *H*(*m*) for *end* and *cut* message*/
20.	*copies_of_cut_message* = (*TP = cut*) ? 0 : *copies_of_cut_message* /* init the account of copies */
21.	Else
22.	If not (*TP = fifo_p*) then /*Construction of the *H*(*m*) begin message*/
23.	*CI*(*p*) ← *CI*(*p*) ∪ *last_fifo*(*p*) /*Adding *fifo_p* messages to CI(p)*/
24.	H(m)_Construction (*CI*(*p*), increment)
25.	*reg*(*p*) ← *last_fifo*(*p*)
26.	*Last_fifo*(*p*) ← ∅

27.	Else
28.	$reg(p) \leftarrow last\_fifo(p)$
29.	$Cop\_CI(p) \leftarrow CI(p)$ /*construction of copies of *begin* message*/
30.	If not ($last\_fifo(p) = \varnothing$) then
31.	$\forall (x, l) \in reg(p)$
32.	If $\exists\ (s, t, d) \in cop\_CI(p) \mid x = s$ and $l \geq t$ then /*construction of the H(m) for copies of begin message */
33.	$cop\_CI(p) \leftarrow cop\_CI(p) / (x, l, d)$
34.	Endif
35.	$cop\_CI(p) = cop\_CI(p) \cup last\_fifo(p)$
36.	Endif
37.	H(m)_Construction($cop\_CI(p)$, copy) /*Construction of the $H(m)$ for copies of begin message*/
38.	Endif
39.	Endif
40.	Else /*construction of copies of *cut* message*/
41.	If ($copies\_of\_cut\_message < num\_cop$) then
42.	H(m)_Construction($CI(p)$,copy) /*the first element of H(m) in a cop_cut message is information
43.	$copies\_of\_cut\_message = copies\_of\_cut\_message + 1$ control end message.*/
44.	$TP = cop\_cut$
45.	Else
46.	$H(m) \leftarrow \varnothing$ /* $H(m)$ for FIFO messages */
47.	Endif
48.	Endif
49.	If ($TP = end$) then /*Determine if process p is sending or not an interval*/
50.	$Act = 0$
51.	$con = 0$
52.	Else
53.	$Act = 1$
54.	Endif
55.	$m \equiv (i, t = VT(p)[i], TP, H(m), data)$ /*construction of sent messages*/
56.	$Sending(m)$
57.	$H(m) \leftarrow \varnothing$
58.	If $\exists\ (k, t, d) \in CI(p) \mid d = dist\_def$ then /*delete the messages with causal distance equal to dist_def */

59.	$CI(p) \leftarrow CI(p) / (k, t, d)$
60.	Endif
	For each message received by *p* with process identifier *j*
61.	*Receive*(*m*) in *p* with $i \neq j$ and $m \equiv (i, t = VT(p)[i], TP, H(m), data)$
62.	If $t = VT(p)[i] + 1$ then /*FIFO deliver condition*/
63.	If (*reception_time_message* > *VTIME*(*p*)[*i*] and *TP*!=*Begin*) then /* messages out his deadline */
64.	Discard(*m*) /* discarded message */
65.	$VT(p)[i] = t$
66.	$VTIME(p)[i] = VTIME(p)[i] + \Delta_{life\_time}$
67.	Else /* else del If (*reception_time_message* > *VTIME*(*p*)[*i*] and *TP*!=*Begin*) then */
68.	If (*TP* ! = *fifo_p* and TP != *cop_cut*) then
69.	If (TP = *begin*) then
70.	*Begin_deadline_measure*(*m*)
71.	If (begin_message_reception_time > begin_deadline) then
72.	Discard(*m*) /* discarded message */
73.	$VT(p)[i] = VT(p)[i] + 1$
74.	$VTIME(p)[i] = begin\_deadline + \Delta$
75.	Else /* check the causal dependencies of the begin message */
76.	If not ($t' \leq VT(p)[l]) \forall (l, t') \in H(m)$ then /* causal delivery condition*/
77.	Put_message_in_queue(*m*,*begin_deadline*)/*queue message because causal dependencies */
78.	Else
79.	Deliver(m)
80.	$VTI(p)[i] = 1$
81.	$VT(p)[i] = VT(p)[i] + 1$
82.	Update_CI_and_Last_fifo (m)
83.	$VTIME(p)[i] = Current\_time + \Delta$
84.	Endif /* endif de If not ($t' \leq VT(p)[l]) \forall (l, t') \in H(m)$ */
85.	Discard_message_out_its_deadline(*m*)
86.	Endif /* endif of If (begin_message_reception_time > begin_deadline) */
87.	Else /* else of If (TP = begin) then, check the causal dependencies for cut and end message */
88.	If not ($t' \leq VT(p)[l]) \forall (l, t') \in H(m)$ then /* causal delivery condition*/
89.	Put_message_in_queue(m,$VTIME(p)[i]$) /*queue message because the causal dependencies */

90.	Else
91.	Deliver(*m*)
92.	*VT*(*p*)[*i*] = *VT*(*p*)[*i*] + 1
93.	Update_CI_and_Last_fifo (m)
94.	*VTIME*(*p*)[*i*] = *VTIME*(*p*)[*i*] + Δ
95.	If (*TP* = *end*) then
96.	*VTI*(*p*)[*i*] = 0
97.	Endif
98.	If (*TP* = *cut*) then
99.	∀ *m'* ∈ *last_cut_del*(*p*)
100.	If ∃ *i'* ∈ *m'* | *i'* = *i* then
101.	*last_cut_del*(*p*) ← *last_cut_del*(*p*) ⁄ *m'*
102.	Endif
103.	*last_cut_del* ← *last_cut_del* ∪ *m*
104.	*VTI*(*p*)[*i*] = *VTI*(*p*)[*i*] +1
105.	Endif
106.	Endif /* endif of If not (*t'* ≤ *VT*(*p*)[*l*]) ∀ (*l*, *t'*) ∈ *H*(*m*) */
107.	Discard_message_out_its_deadline(m)
108.	Endif /* endif of If (TP = begin) then */
109.	Else /* else of If (*TP* ! = *fifo_p* and TP != *cop_cut*) then */
110.	Deliver(*m*)
111.	*VT*(*p*)[*i*] = *VT*(*p*)[*i*] + 1
112.	Update_CI_and_Last_fifo (*m*)
113.	*VTIME*(*p*)[*i*] = *VTIME*(*p*)[*i*] + Δ
114.	Endif /* endif of If (*TP* ! = *fifo_p* and TP != *cop_cut*) then */
115.	Endif /* endif of If (*reception_time_message* > *VTIME*(*p*)[*i*] and *TP*!=*Begin*) then */
116.	Else /* else of If *t* = *VT*(*p*)[*i*] +1 Then */
117.	If (*t* > *VT*(*p*)[*i*]) then
118.	If (*TP* = *fifo_p* and *VTI*(*p*)[*i*] = 0) Then /* *fifo* message used as message causal */
119.	Begin_deadline_measure(*m*)
120.	received_message_wait_time= begin_deadline +((*t* – *VT*(*p*)[*i*]) * Δ – time_defect_added)
121.	If (*current_time* > received_message_wait_time) then

122.	If not ($t' \leq VT(p)[l]) \forall (l, t') \in H(m)$ then /* causal delivery condition*/
124.	$VT(p)[l] = t'$ /*update of vectors*/
125.	Endif /* endif de If not ($t' \leq VT(p)[l]) \forall (l, t') \in H(m)$ */
128.	Deliver(m)
129.	$VTI(p)[i] = 1$
130.	$VT(p)[i] = t$
131.	$causal = 1$
132.	$VTIME(p)[i] = received\_message\_wait\_time + \Delta$
133.	Update_CI_and_Last_fifo (m)
134.	Else
135.	Put_message_in_queue (m, $received\_message\_wait\_time$) //components la functions
136.	Endif /* endif If ($current\_time > received\_message\_wait\_time$) then */
137.	Else /* else of If ($TP = fifo\_p$ and $VTI(p)[i] = 0$) Then */
138.	$received\_message\_wait\_time = VTIME(p)[i] + ((t – VT(p)[i]) * \Delta$ – time_defect_added)
139.	If ($current\_time > received\_message\_wait\_time$) then
140.	If ($TP = cop\_cut$ or $TP = cut$) then /* cop_cut message used as a message causal */
141.	$\forall m' \in last\_cut\_del(p)$
142.	If $\exists i' \in m' \mid i' = i$ then
143.	If ($H($ last_cut_del) ! = $H(m)$) then /* Dectection of lost cut messages */
144.	If not ($t' \leq VT(p)[l])\forall(l, t') \in H(m)$ then /*causal delivery condition*/
145.	$VT(p)[l] = t'$ /*update of vectors*/
146.	Sending (cut) /* Sending a cut message by lost end message */
147.	Endif /* endif de If not ($t' \leq VT(p)[l]) \forall (l, t') \in H(m)$ */
148.	Deliver(m)
149.	$VT(p)[i] = t$
150.	$VTI(p)[i] = VTI(p)[i] +1$
151.	$last\_cut\_del(p) \leftarrow last\_cut\_del(p) \nearrow m'$
152.	$last\_cut\_del(p) \leftarrow last\_cut\_del(p) \cup m$
153.	$causal = 1$
154.	$VTIME(p)[i] = received\_message\_wait\_time + \Delta$
155.	Update_CI_and_Last_fifo (m)
156.	Else /* else of If ($H($ last_cut_del) ! = $H(m)$) then */

157.	Deliver(m)
158.	$VT(p)[\ i\ ] = t$
159.	$VTIME(p)[i] = received\_message\_wait\_time + \Delta_{life\_time}$
160.	Update_CI_and_last_fifo (m)
161.	Endif /* endif of If ($H($ last$_cut_del$) ! = $H(m)$) then */
162.	Else /* else of If $\exists\ i' \in m' \mid i' = i$ then */
163.	If not ($t' \le VT(p)[l]) \forall\ (\ l, t'\ ) \in H(m)$ then /* causal delivery condition*/
164.	$VT(p)[l] = t'$ /*update of vectors*/
165.	Send (cut) /*Sending a cut message by the lost message end*/
166.	Endif
167.	Deliver(m)
168.	$VT(p)[\ i\ ] = t$
169.	$VTIME(p)[i] = received\_message\_wait\_time + \Delta$
170.	$last\_cut\_del(p) \leftarrow last\_cut\_del(p) \cup m$
171.	$causal = 1$
172.	Update_CI_and_Last_fifo(m)
173.	Endif /* end of If $\exists\ i' \in m' \mid i' = i$ then */
174.	Else /* else of If ($TP = cop\_cut$ or $TP = cut$) then */
175.	If ($TP = = end$) then
176.	If not ($t' \le VT(p)[l]) \forall\ (\ l, t'\ ) \in H(m)$ then /* causal delivery condition*/
177.	$VT(p)[l] = t'$ /*update of vectors*/
178.	Endif
179.	Deliver(m)
180.	$VT(p)[\ i\ ] = t$
181.	$VTIME(p)[i] = received\_message\_wait\_time + \Delta_{life\_time}$
182.	Update_CI_and_last_fifo (m)
183.	Else /* else of If (TP = = end) then, deliver of FIFO messages */
184.	Deliver(m)
185.	$VT(p)[\ i\ ] = t$
186.	$VTIME(p)[i] = received\_message\_wait\_time + \Delta$
187.	Update_CI_and_last_fifo (m)
188.	Endif /* endif of If (TP = = end) then */

```
189.                    Endif            /* endif of  If ( TP = cop_cut  or TP = cut ) then  */
190.                Else            /* else of  If ( current_time > received_message_wait_time ) then */
191.                    Put_message_in_queue ( m, received_message_wait_time)
192.            Endif        /*  endif of  If ( current_time > received_message_wait_time ) then  */
193.        Endif      /*  endif of if  If ( TP = fifo_p and VTI(p)[i] = 0 ) Then  */
194.      Else     /* else of  if ( t > VT(p)[i] )  condition */
195.        Discard (m)
196.    Endif        /* end of  if ( t > VT(p)[i] )  condition */
197.  Endif         /* end of  if ( t =  VT(p)[i] +1) FIFO deliver condition */
198. EndFor      /* end of function receive */
199.
200. Discard_message_out_deadline( m )  {    /* message received  m ≡ (i, t = VT(p)[i], TP , H(m),data) */
201. ∀(l, t' )∈ H(m)
202.    If not ( t' ≤ VT(p)[l]) then    /* causal delivery condition*/
203.          msg_res = t' – VT(p)[l]
204.          msg_val = (VTIME(p)[l] - current_time) / play_time
205.          If (msg_res > msg_val )  then /*Detection of lost message*/
206.                msg_dis = msg_res – msg_val
207.                VT(p)[l] = VT(p)[l] + msg_dis      /*update of vectors*/
208.          Endif
209.    Endif
210. }
211.
212. Begin_deadline_measure( m ) {
213.    Δinter-st = Δinter -  Δintra
214.   ∀(l, t' )∈ H(m)
215.       If  ( t' ≤ VT(p)[l]) then    /* causal delivery condition*/
216.          If (begin_deadline>VTIME(p)[l] ) then
217.             begin_deadline = VTIME(p)[l]
218.          Endif
219.       Endif
220.    begin_deadline = begin_deadline +  Δinter-st
```

```
221. }
222. Update_CI_last_fifo ( m ) {    /*updating the CI and last FIFO with the message delivered */
223.  If  not ( (TP =  fifo_p or TP = cop_cut )  and causal = 0  )    Then   /* update the CI and last_fifo */
224.  CI(m) ← CI(m) ∪ { ( i, t, d = 0) }
225.  ∀(l, t' ) ∈ H(m)                /*Updating CI(p)with a most recent message*/
226.       If ∃ (s, t, d ) ∈ CI(p) | l = s and  t' = t  then
227.          (s, t, d ) ← (s, t, d+1)
228.       Endif
229.       If ∃ (s, t, d ) ∈ last_fifo(p) | l = s and t' = t  then     /* incrementing the causal distance of last FIFO*/
230.          (s, t, d ) ← (s, t, d +1)
231.       Endif
232.       If  ∃ (s, t, d) ∈ CI(p) | d = dist_def  then  /*delete the element with causal distance equal defined causal */
233.          CI(p) ← CI(p) /  ( s, t, d )
234.       Endif
235.  VTI(p)[ i ] = ( TP = end ) ? 0 : VTI(p)[ i ]  /* init the VTI  when a end message is delivered */
236.  VTIME(p)[ I ]=( TP = end )?big_value:VTIME(p)[ i ] /* init the VTIME  when a end message is delivered */
237.  If ( Act = 1 and  not( TP = cut) and  not ( TP = begin)  then
238.     Send( cut )     /*sending a cut message by the deliver of a end message*/
239.  Endif
240.  If ( TP = cut ) then
241.     last_cop_cut = H(m)
242.     VTI(p)[ i ] = VTI(p)[ i ] + 1
243.  Endif
244.  Else
245.     If  ∃  (x, l, d ) ∈ last_fifo(p) | x = i  then    /*Updating last_fifo(p) with a most recent message*/
246.        Last_fifo(p) ←  last_fifo(p) / ( x, l, d )
247.     Endif
248.     last_fifo(p) ← last_fifo(p) ∪ ( i, t, d=0 )   /* adding the last fifo*/
249.   causal = 0
250.  If ∃ (s, t, d ) ∈ last_fifo(p)  | d = dist_def  then /*delete a element in last fifo */
251.     last_fifo(p) ←  last_fifo(p) / (s, t, d )
252.  Endif
```

253.	Endif /* end of function actualizacion_CI_last_fifo */
254.	}
255.	
256.	Put_message_in_queue (*m* , *wait_time_max*) { /* message received *m*= (*i*, *t* = *VT*(*p*)[*i*], *TP* , *H*(*m*),*data*) */
257.	*m`* = *m* ∪ (*wait_time_max*) /* adding to message the wait time in the queue*/
258.	*queue* = *queue* ∪ *m`* /* put the message in the queue*/
259.	}
260.	
261.	Check_timer { /* message structure *m`* = (*i*, *t* = *VT*(*p*)[*i*], *TP* , *H*(*m*),*data, time*) */
262.	∀ *m`* ∈ *queue*
263.	If ∃ *time* ∈ m` | *time* > *current_time* then /*delivering a message with a deadline bigger that current time */
264.	∀(*l*, *t'*) ∈ *H*(*m*)
265.	If (*t`* > *VT*(*p*)[*l*]) then
266.	*VT*(*p*)[*l*] = *t`*
267.	If (*TP* = *cut* or *TP* = *cop_cut*) then
268.	Send (*cut*)
269.	Endif
270.	Endif
271.	Deliver (*m`*)
272.	*VT*(*p*)[*i*] = *t*
273.	*VTI*(*p*)[*i*] = (*TP* = *begin*) ? 1 : *VTI*(*p*)[*i*]
274.	*VTIME*(*p*)[*i*] = *VTIME*(*p*)[*i*] + Δ
275.	*queue* = *queue* / *m`*
276.	update_CI_last_fifo (*m*)
277.	Endif
278.	}
279.	H(m)_Construction (*CI*(*p*), *bandera*) {
280.	For each (*s*,*t*,*d*) ∈ *CI*(*p*)
281.	(*s*,*t*,*d*) ← (bandera = *increment*) ? (*s*,*t*,*d*+*1*) : (*s*,*t*,*d*)
282.	*H*(*m*) ← *H*(*m*) ∪ (*s*, *t*)
283.	Endfor
284.	}

Referencias

[ABO98] A. Abouaissa, A. Benslimane, M. Naimi, "A Group Communication Model for Distributed Real-Time Causal Delivery", *Seventh International Conference on Computer Communications and Networks (ICCCN '98)*, 1998, 918.

[ABO99] A. Abouaissa, A. Benslimane, "A Multicast Synchronization Protocol for Real Time Distributed Systems", *Seventh IEEE International Conference on Networks (ICON'99)*, 1999, 21.

[ADE95] F. Adelstein, M. Singhal, "Real-time causal message ordering in multimedia systems", *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, 1995, 36-43.

[AGA96] N. Agarwal, S. Hyuk, "A Model for Specification and Synchronization of Data for Distributed Multimedia Applications", *Multimedia Tools and Applications,* 3(2), 1996, 79-104.

[AMI00] Y. Amir, C. Danilov, J. Stanton, "A low latency, loss tolerant architecture and protocol for wide area group communication", *Proceedings of International Conference on Dependable Systems and Networks*, 2000, 327-336.

[BAL94] R. Baldoni, Achour Mostefaoui, Michel Raynal, "Causal delivery of messages with real-time data in unreliable networks", *Real-Time Systems, the International Journal of Time-Critical Computing Systems*, 10(3): 1996, 245-262.

[BAL97] R. Baldoni, R. Friedman, R. van Renesse, "The hierarchical daisy architecture for causal delivery", *Proceedings of 17 International Conference on Distributed Computing Systems (ICDCS '97)*, 1997, 71-81.

[BAL00] P. Balaouras, I. Stavrakakis, L. F. Merakos, "Potential and Limitations of a Teleteaching Environment based on H.323 Audio-visual", *Communication Systems. Computer Networks*, 34(6), 2000, 945-958.

[BAN04] S. Banerjee, S. Lee, R. Braud, B. Bhattacharjee, A. Srinivasan: " Scalable Resilient Media Streaming ", *Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video (NOSSDAV '04)*, 2004, 4-9.

[BEN00] A. Benslimane, "A Multimedia Synchronization Protocol for Multicast Groups", *26th EUROMICRO Conference (EUROMICRO'00)*, 1(1), 2000, 456-463.

[CAR03] M. Carson, D. Santay, "NIST Net – A Linux-based Network Emulation Tool", *the Special Interest Group on Data Communication (SIGCOMM)*, 33(3),2003, 111-126.

[DWY98] D. Dwyer, S. Ha, J. Li, V. Bharghavan, "An Adaptive Transport Protocol for Multimedia Communication", *Proceedings of IEEE Conference on MultiMedia Computing and Systems*, 1998, 23-32.

[FEA02] N. Feamster, H. Balakrishnan, "Packet Loss Recovery for Streaming Video", *12th International Packet Video Workshop*, 2002.

[GUS89] R. Gusella, S. Zatti, “The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley Unix 4.3BSD”, *IEEE Transactions on Sofware Engineering*, 15(7), 1989, 847-853.

[HAS98] T. Hasegawa, T. Hasegawa, T. Kato, “Design and Implementation of Reliable Protocol for Video Data Produced in Real-Time Manner”, *Fifth International Conference on Real-Time Computing Systems and Applications (RTCSA'98)*, 1998, 200.

[HOF95] M. Hofmann, Torsten Braun, Georg Carle, “Multicast communication in large scale networks”, *Proceedings of Third IEEE Workshop on High Performance Communication Subsystems (HPCS)*, Mystic, Connecticut, 1995, 147-150.

[JIA95] X. Jia, “A Total Ordering Multicast Protocol Using Propagation Trees”, *IEEE Transactions on Parallel and Distributed System*, 6(6), 1995, 617-627.

[LAM78] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System”, *Communications, ACM,* 21(7), 1978, 558-565

[LOP05] E. López, J. Estudillo, J. Fanchon, S. Pomares, “A Fault-tolerant Causal Broadcast Algorithm to be Applied to Unreliable Networks”, *The 17th IASTED International Conference on Parallel and Distributed Computing and Systems*, Arizona,USA, (2005).

[LOP06] E. López, L. Morales, S. Pomares, “A Mechanism to Synchronize Distributed Continuous Media in Unreliable Networks”, *Sixth IEEE International Symposium and School on Advance Distributed Systems ISSADS 2006*, Springer- Lecture Notes in Computer Science, México, 2006.

[MIL91] D. Mills, Internet Time Synchronization, “The Network Time Protocol”, *IEEE Transactions on Communication*, 39(10), 1991, 1482-1493.

[MOL91] G. Molina, H. Spauter, “Ordered and Reliable Multicast Communication”, *ACM Transactions. Computer System*, 9(3), 1991,pp 224-271.

[MOR05] L. Morales, “Algoritmo de Sincronización de flujos continuos en tiempo real”, Tesis de Maestria en Ciencias Computacionales, INAOE. Num. XM1086. Classification: XMM-M67-2005-XM1086, Tonantzintla Puebla, México. 2005.

[NAK94] A. Nakamura, M. Takizawa, “Causally Ordering Bradcast Protocol”, *the 26th International Conference on Distributed Computing Systems* (ICDCS-14), 1994, 48-55.

[NON97] J. Nonnenmacher, E. Biersack, D. Towsley, “Parity-Based Loss Recovery for Reliable Multicast Transmission”, *Special Interest Group on Data Communications '97*, Cannes, France, 1997, 289-300.

[PAU96] J. Lin, S. Paul, “RMTP A reliable multicast transport protocol”, *Proceedings of IEEE Infocom*, 1996, 1414-1424.

[PER03] C. Perkins, “RTP Audio and Video for Internet”, Addison Wesley, USA 2003, 412.

[POM02] S. Pomares, “Services de Coordination et Algorithmes De Diffusion Causale pour Les Applications Coopératives Distribuées”, PhD dissertation at the National Polytechnic Institute of Toulouse (INPT), France, November, 2002.

[POM04] S. Pomares, J. Fanchon, K. Drira, “The Immediate Dependency Relation: An Optimal Way to Ensure Causal Group Communication”, *Annual Review of*

Scalable Computing, Editions World Scientific, Series on Scalable Computing, 6 (1), 2004, pp. 61-79.

[PLE05] C. Plesca, R. Grigoras, P. Queinnec, G. Padiou, "A Flexible Communication Toolkit for Synchronous Groupware", *Proceedings of the 2005 Systems Communications (ICW'05, ICHSN'05, ICMCS'05, SENET'05)*, 2005, 216 – 221.

[RAN96] S. Sampath, S. Rajan., "Continuity and Synchronization in MPEG", *IEEE Journal on Selected Areas in Communications*, 14(1), 1996, 52-60.

[RAV93] K. Ravindra, V. Bansal, "Delay Compensation Protocols for Synchronization of Multimedia Data Streams", *IEEE Transactions on Knowledge and Data Engineering*, 5 (1), 1993, 574-589.

[RUB98] D. Rubenstein, J. Kurose, D. Towsley, "Real-Time Reliable Multicast Using Proactive Forward Error Correction", *Proceedings of The 8th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Cambridge, England, 1998.

[TAC96a] T. Tachikawa, M. Takizawa, "Group Communication for Real-time Continuous Media", *Procceding of International Simposium on Multimedia System 1996*, 118-125.

[TAC96b] T. Tachikawa, M. Takizawa, "Communication Protocol for Wide-area Group", *The International Computer Symposium (ICS'96)*, 1996, 158-165.

[TAC97] T. Tachikawa, M. Takizawa, "Δ-Causality in Wide-Area Group Communications". *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS '97)*, 11-13 December 1997, Seoul, Korea, IEEE Computer Society 1997, 260-267.

[YAN99] Z. Yang, C. Sun, A. Sattar, Y. Yang, "A New Look At Multimedia Synchronization in Distributed Environments", *International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '99)*, 1999, 322.

[WAH94] T. Wahl, K. Rothermel, "Representing Time in Multimedia Systems", *International Conference on Multimedia Computing and Systems*, Massachusetts, USA, 1994, 538-543.

[WIJ96] D. Wijesekera, J. Srivastava, A. Nerode, M. Foresti, "Experimental Evaluation of Loss Perception in Continuous Media", *Multimedia Systems*, July 1997, 486-499.