Capítulo 8 - SENSEIGMS

SenseiGMS es un sistema de grupo de comunicaciones fiables desarrollado específicamente para este proyecto. Está basado en el modelo de sincronía virtual y sobre este sistema se han implementado los protocolos de transferencia de estado presentados en los anteriores capítulos, así como varias técnicas desarrolladas para la simplificación de sistemas fiables, descritas en los siguientes capítulos.

Su diseño se ha realizado bajo tres requisitos:

- Soportar una funcionalidad simple y básica para facilitar su implementación, pues el objetivo de este proyecto no es el desarrollo de un sistema de comunicaciones fiables de grupo, sino la implementación de los protocolos de transferencia de estado estudiados y de herramientas que simplifiquen el desarrollo de aplicaciones replicadas.
- Limitar su funcionalidad a la descrita en el modelo de sincronía virtual, permitiendo únicamente las extensiones disponibles en otros sistemas de comunicaciones ampliamente conocidos, como es el envío de mensajes punto a punto. El objetivo de este requisito es poder sustituir este sistema por esos sistemas de comunicaciones existentes, generalmente más complicados funcionalmente y con mejor rendimiento.
- Presentar una interfaz orientada a objetos y con soporte de mensajes tipados, con el objetivo de facilitar el desarrollo de las capas situadas sobre este sustrato de comunicaciones.

Este capítulo presenta el diseño de SenseiGMS y los algoritmos que lo sustentan, así como el modelo de programación que soporta. Las razones que han impuesto la creación de un sistema propio de comunicaciones en grupo se reducen básicamente a la necesidad de disponer de un sistema sobre CORBA y JavaRMI; en el momento de desarrollar la mayor parte de la tesis, no había disponibilidad de un sistema de comunicaciones en grupo sobre CORBA, a pesar de que la especificación de tolerancia a fallos de esta arquitectura estuviera ya aceptada. De la misma forma, la necesidad de desarrollar nuestro propio algoritmo para las comunicaciones multipunto fiables se justifica al observar que los algoritmos existentes se basan normalmente en el empleo de primitivas de comunicaciones multipunto no fiables (UDP), cuando la especificación de nuestro sistema incluye sólo soporte de comunicaciones fiables punto a punto.

8.1. Diseño

SenseiGMS soporta aplicaciones CORBA y JavaRMI. Su interfaz está orientada a objetos, especificada directamente en OMG/IDL y en Java, y como las aplicaciones deben definir sus mensajes de la misma forma, el soporte de mensajes tipados es directo.

La interfaz de aplicación de SenseiGMS está basada en la definida en *Isis* que, a su vez, ha sido la base para un gran número de sistemas de comunicaciones en grupo: *Horus*, *Ensemble*, *Totem*, *Transis* o *JavaGroups*.

Las comunicaciones se realizan directamente empleando CORBA o JavaRMI. La aproximación empleada en otros sistemas de comunicaciones es crear las pilas de protocolos sobre UDP o, incluso, directamente sobre IP, pilas que además resultan dinámicas en la selección de los algoritmos de ordenación de mensajes. Esta elección implica un peor rendimiento en SenseiGMS, pero simplifica su diseño. Adicionalmente, la elección de un *middleware* de alto nivel como substrato de comunicaciones permite emplear las facilidades y servicios asociados al mismo. Por ejemplo, empleando *applets* autentificados, se podrían desarrollar *applets* tolerantes a fallos más fácilmente que si el soporte se realiza directamente sobre *sockets*. De la misma manera, se pueden aprovechar otras facilidades como *http tunneling*, ya disponibles al trabajar con JavaRMI o CORBA, o la descarga dinámica de código.

Otra de las decisiones de simplificación del diseño es el soporte exclusivo de orden total causal en la secuenciación de mensajes. El algoritmo empleado para obtener el orden total causal, es mediante el paso de un testigo sobre un anillo lógico compuesto por los componentes del grupo, que puede cambiar su estructura según se incluyan nuevos miembros en el grupo o antiguos sean excluidos. La decisión de soportar exclusivamente orden total está influenciada, además, por tres motivos:

- La programación de aplicaciones replicadas se simplifica enormemente cuando los mensajes se procesan en orden total.

- El protocolo que sustenta el orden total puede soportar este orden a un coste no mayor que con empleo de orden causal o simplemente con mensajes *fifo* punto a punto fiables [Amir95].
- La especificación del servicio de tolerancia a fallos de CORBA incluye únicamente el soporte de este orden de secuenciación.

Para soportar simultáneamente ambas plataformas, SenseiGMS define su interfaz en OMG/IDL o en JavaRMI, e implementa separadamente los algoritmos, comunes en los dos casos, empleando Java. Desde el punto de vista de implementación, tanto la interfaz OMG/IDL, como la interfaz JavaRMI, comparten un mismo *package*, llamado *sensei.middleware.GMS*, aunque se localizan en distintos directorios. Los algoritmos comunes, definidos en el paquete *sensei.GMS*, se compilan sobre una de las plataformas previas, simplemente seleccionando un *classpath* diferente. El único requisito, en este caso, es que ambas interfaces sean completamente compatibles.

La interfaz que SenseiGMS presenta a la aplicación, que incluye, por ejemplo, operaciones para el envío de mensajes fiables, constituye su interfaz pública. Sin embargo, para realizar las comunicaciones internas también en CORBA o JavaRMI, como, por ejemplo, el paso del testigo entre miembros de un grupo, es necesario definir una segunda interfaz en los correspondientes lenguajes. Esta última interfaz es privada; en caso de querer sustituir SenseiGMS por otro grupo de comunicaciones, es necesario soportar exclusivamente la interfaz pública.

Finalmente, SenseiGMS no presenta ningún soporte de transferencia de estado. A pesar del énfasis de esta tesis en los algoritmos de transferencia de estado, esta decisión permite implementarlos totalmente en una capa superior, soportando así más fácilmente su migración a otros sistemas de comunicaciones en grupo.

8.1.1. Interfaz pública

Esta interfaz se basa en tres tipos básicos:

- Identidades de miembros, para lo que se emplea un simple entero, que debe ser único para cada miembro del grupo.
- Definición de mensajes. Cada aplicación puede definir sus propios mensajes, que deben pertenecer a una jerarquía de objetos definida. Estos mensajes pueden definirse en detalle obteniéndose, así, mensajes tipados.
- Definición de vistas, acorde al modelo de sincronía virtual. Cuando un grupo cambia su composición, cada miembro del grupo recibe una lista con todos los miembros. Esta lista constituye la vista que incluye, asimismo, una identidad que se define de forma numérica.

En OMG/IDL, su especificación es:

```
typedef long GroupMemberId;
typedef sequence <GroupMemberId> GroupMemberIdList;

struct View
{
   long              viewId;
   GroupMemberIdList members;
   GroupMemberIdList newMembers;
   GroupMemberIdList expulsedMembers;
};

valuetype Message {};
```

En SenseiGMS, por conveniencia, las vistas incluyen también una lista con los miembros excluidos y los nuevos miembros en el grupo. Es importante destacar que esta información no supone una funcionalidad añadida: un miembro nuevo recibe una vista en la que todos los demás miembros son listados como nuevos. Este detalle es importante para los algoritmos de transferencia de estado, que pueden simplificarse si las vistas contienen información con la que discernir cuándo un miembro es más antiguo que otro. Siguiendo el objetivo de no ofrecer mayor funcionalidad que otros sistemas de comunicaciones, esta información no se incluye.

La definición de mensaje como *valuetype* es esencial. Si se define como *interface*, el mensaje no es realmente transferido a cada miembro del grupo, que sólo recibe una referencia remota; si se define como *struct*, se transfiere a cada miembro del grupo, pero no se pueden definir nuevos mensajes por herencia.

Bajo JavaRMI, la especificación resulta similar; sin incluir los constructores para la clase, es:

```
final public class View implements java.io.Serializable
{
   public int viewId;
   public int[] members;
   public int[] newMembers;
   public int[] expulsedMembers;
}

public abstract class Message
  implements java.io.Serializable
{
}
```

Un objeto debe implementar la interfaz *GroupMember* para poder pertenecer a un grupo. Esta interfaz se especifica en JavaRMI como:

```
public interface GroupMember extends java.rmi.Remote
{
   public void processPTPMessage(int sender, Message message) throws RemoteException[2];
   public void processCastMessage(int sender, Message message) throws RemoteException;
   public void memberAccepted(int identity, GroupHandler handler, View view)
                              throws RemoteException;
   public void changingView() throws RemoteException;
   public void installView(View view) throws RemoteException;
   public void excludedFromGroup() throws RemoteException;
}
```

A través de esta interfaz, el miembro recibe todas las comunicaciones del grupo que, siguiendo el orden mostrado, es:

- Recepción de mensajes multipunto: todos los miembros ven la misma secuencia de mensajes. El miembro que envía un mensaje también lo recibe.
- Recepción de mensajes punto a punto. SenseiGMS respeta el orden total tanto entre mensajes multipunto como entre mensajes punto a punto; sin embargo, ésta no es una característica común a otros sistemas de comunicaciones fiables, donde estos mensajes pueden recibirse con antelación a mensajes multipunto que les preceden en el orden establecido. Por esta razón, ninguno de los algoritmos en Sensei es dependiente de este orden en mensajes punto a punto.
- Aceptación del miembro en el grupo, que es el primer evento que se recibe en todos los casos.
- Evento de cambios de vista. Se recibe antes de que se instale una nueva vista, e implica que el servicio de pertenencia de miembros ha detectado que es necesario un cambio de vista y está negociando la nueva composición del grupo. Este evento no se incluye en el modelo de sincronía virtual, pero es común en los sistemas de comunicaciones en grupo. Los protocolos de transferencia de estado precisan este evento, como se demostró en los capítulos anteriores.
- Instalación de una nueva vista.
- Evento de exclusión del grupo, ya sea por petición del miembro, o porque el GMS considere, erróneamente, que este miembro ha caído.

Un miembro interactúa con los demás miembros del grupo a través de la interfaz *GroupHandler*, que define la interfaz pública del servicio de pertenencia a grupos o GMS. Esta interfaz no define cómo crear el grupo o cómo incluir un miembro en un grupo ya existente, pues esta funcionalidad es específica a la implementación de SenseiGMS y se define como parte de su interfaz privada. Hay una relación biunívoca entre *GroupMember* y *GroupHandler*, es decir, cada miembro

[2] Todas las excepciones son *java.rmi.RemoteException.*

del grupo mantiene una instancia del servicio de pertenencia a grupos. Otros sistemas de comunicaciones en grupo permiten, sin embargo, que una instancia de este servicio controle varios miembros o, incluso, varios grupos, pero esta aproximación no implica una mayor o menor funcionalidad en ningún caso.

Esta interfaz se especifica en JavaRMI como:

```
public interface GroupHandler extends java.rmi.Remote
{
   public boolean castMessage(Message message) throws RemoteException;
   public boolean sendMessage(int target, Message message) throws RemoteException;
   public boolean leaveGroup() throws RemoteException;
   public int getGroupMemberId() throws RemoteException;
   public boolean isValidGroup() throws RemoteException;
}
```

Las operaciones definidas son:

- Enviar un mensaje al grupo, que es recibido por todos los miembros, incluido el que lo envía. Devuelve un valor que indica si el mensaje será procesado en la vista actual. A partir del momento en que el servicio de pertenencia al grupo bloquea una vista para negociar la composición de la siguiente, todos los nuevos mensajes que se envíen en el grupo son bloqueados, siendo sólo enviados en la siguiente vista.
- Enviar un mensaje a un miembro determinado del grupo. Para mantener una interfaz coherente, esta operación devuelve un valor *true* si el mensaje es recibido por el miembro destinatario en la misma vista.
- Abandonar el grupo. Esta operación no es inmediata, el miembro no es efectivamente expulsado hasta recibir el evento de expulsión.
- Detectar si un grupo es válido.
- Obtener la identidad de miembro asociada.

La especificación en OMG/IDL de estos tipos es paralela:

```
interface GroupMember
{
   void processPTPMessage(in GroupMemberId sender, in Message msg);
   void processCastMessage(in GroupMemberId sender, in Message msg);
   void memberAccepted(in GroupMemberId identity, in GroupHandler handler,
                       in View theView);
   void changingView();
   void installView(in View theView);
   void excludedFromGroup();
};
```

```
interface GroupHandler
{
   boolean castMessage(in Message msg);
   boolean sendMessage(in GroupMemberId target, in Message msg);
   boolean leaveGroup();
   GroupMemberId getGroupMemberId();
   boolean isValidGroup();
};
```

8.1.2. Interfaz privada

La parte privada de la interfaz define las operaciones que los miembros del GMS deben soportar para implementar su propio protocolo de comunicaciones, en este caso, basado en el paso de un testigo. La definición principal es la del miembro del GMS, denominada *SenseiGMSMember*, que es una extensión de la definición de *GroupHandler* presentada en la sección anterior. En esta sección, los tipos están especificados empleando únicamente OMG/IDL pues, como se ve en los casos anteriores, la especificación JavaRMI es inmediata.

Las operaciones definidas en esta interfaz están fuertemente ligadas a los algoritmos empleados para mantener el modelo de sincronía virtual. Dos características importantes, desarrolladas en mayor profundidad al explicar el protocolo, son:

- Cada miembro puede comunicarse con los otros cuando posee el testigo. Como el principal problema en este tipo de comunicación es que el testigo aparezca duplicado en el grupo, se define de tal forma que sea identificable sin posibilidad de duplicados. Empleamos su anglicismo, *token*, para nombrarlo:

```
struct Token
{
   GroupMemberId creator;
   long id;
};
```

- En la interfaz pública, los miembros reciben vistas con las identidades de todos los miembros del grupo. Sin embargo, para realizar las comunicaciones, es necesario que se conozcan las referencias a estos miembros, no sólo sus identidades, lo que supone el empleo de vistas internas. Adicionalmente, existe el concepto de vista temporal, como cada una de las vistas que el grupo negocia mientras se discute la composición final del grupo; por esta razón, la identidad de vista interna incluye un campo identificando la vista como temporal:

```
struct InternalViewId
{
   long id;
```

```
      long installing;
   };
   typedef sequence <SenseiGMSMember> SenseiGMSMemberList;
   struct InternalView
   {
      InternalViewId viewId;
      SenseiGMSMemberList members;
      GroupMemberIdList memberIds;
      GroupMemberIdList newMembers;
   };
```

La interfaz *SenseiGMSMember* se define como:

```
interface SenseiGMSMember : GroupHandler
{
   boolean receiveToken(in GroupMemberId sender, in Token theToken);
   boolean receiveView(in GroupMemberId sender, in InternalView view,
                       in Token viewToken);
   boolean receiveMessages(in GroupMemberId sender, in MessageList messages,
                           in MessageId msgId);
   boolean confirmMessages(in GroupMemberId sender, in MessageId msgId);
   boolean receivePTPMessages(in GroupMemberId sender, in MessageList messages,
                              in MessageId msgId);
   AllowTokenRecoveryAnswer allowTokenRecovery(in GroupMemberId sender,
                                               in InternalViewId myViewId);
   boolean recoverToken(in GroupMemberId sender);
   boolean addGroupMember(in SenseiGMSMember other);
   boolean createGroup();
   boolean joinGroup(in SenseiGMSMember group);
};
```

Estas operaciones se describen junto a su significado en la siguiente sección, al explicar el algoritmo empleado. Las tres últimas operaciones son las que permiten crear o extender grupos, tomando como argumento una referencia a un objeto del mismo tipo. Es decir, sólo un miembro que implemente esta interfaz privada puede incluirse en el grupo y, por esa razón, es preciso definir estas operaciones como privadas.

Para que una aplicación pueda crear réplicas sin necesidad de acceder a esta interfaz privada, Sensei incluye una interfaz externa denominada SenseiGMNS que contempla la funcionalidad generalmente disponible en otros sistemas de pertenencia a grupo. GMNS significa *GroupMembershipNamingService* pues, además de incluir la creación y extensión manual de grupos, permite manejarlos de forma simple, asociando un nombre a los grupos. Este servicio, totalmente ligado a la implementación de SenseiGMS, se describe en el capítulo 11.

Todas las operaciones devuelven un valor booleano; si éste es *false*, implica que el miembro destino no acepta la operación efectuada, generalmente porque considere al miembro fuente no perteneciente al grupo. La excepción es la operación *allowTokenRecovery*, que precisa más información y se define como:

```
struct AllowTokenRecoveryAnswer
{
   boolean validCommunication;
   boolean recoveryAllowed;
   MessageId lastMessage;
};
```

8.2. Algoritmo de paso de testigo

SenseiGMS se basa en el paso de un testigo que circula entre los miembros del grupo; la posesión del testigo permite realizar el ordenado de los mensajes de una forma sencilla, y el algoritmo queda sólo complicado por la necesidad de regenerar el testigo si el miembro que lo controlaba se cae, e impedir a la vez que dos testigos circulen simultáneamente en el grupo.

Casi todas las operaciones de grupo que un miembro puede realizar, tales como enviar mensajes, expulsar a miembros sospechosos de haber caído o incluir nuevos miembros, se realizan, exclusivamente, una vez que el miembro entra en posesión del testigo. La única excepción se refiere a las operaciones de recuperación del testigo.

El paso de un testigo no es la única forma de obtener orden total. En Ameba [Kaashoek91], por ejemplo, el orden total se alcanza mediante un proceso especial, denominado secuenciador, al que los demás procesos envían sus mensajes punto a punto; el secuenciador los redistribuye luego a los demás miembros con el orden adecuado. El protocolo *Psync* [Peterson89] crea un orden parcial en los mensajes que puede ser convertido en orden total.

Sin embargo, son más numerosos los sistemas basados en paso de testigo. *Totem* define un algoritmo muy eficiente de paso de testigo que admite, además, el particionado de la red. Al igual que en SenseiGMS, el miembro que posee el testigo es el único que envía mensajes aunque, en otros casos [Chang84], cualquier miembro puede enviar mensajes en cualquier momento, produciendo, sin embargo, altas latencias cuando hay numerosos mensajes en el grupo o ante caídas de miembros. Similar al protocolo de *Totem* es el protocolo *TPM* [Rajagopalan89], aunque no admite el particionado de los grupos. Estos protocolos se basan en el soporte de mensajes multicast *best-effort*, principalmente UDP, mientras que SenseiGMS se construye sobre las comunicaciones punto a punto de CORBA o RMI, lo que ha provocado la necesidad de desarrollar un algoritmo propio.

8.2.1. Paso de testigo y estructura en anillo

Los miembros pueden considerarse lógicamente dispuestos en una estructura de anillo, donde la vista define los componentes y su respectivo orden. Este anillo es dinámico, expandiéndose según nuevos miembros se insertan en el grupo o los antiguos lo abandonan o son excluidos. La implementación actual del algoritmo inserta los miembros lógicamente tras el miembro que los introduce en el grupo.

Un testigo no es, en realidad, mas que un tipo especial de mensaje transferido punto a punto. Cuando una nueva vista temporal se crea, se define un nuevo testigo con una identidad única, que el miembro que envía la vista propaga junto con esa vista. Este testigo circula, a continuación, en el grupo y los miembros deben verificar su identidad. Dos problemas pueden aparecer con este paso de testigo. El primer problema es que el miembro que posee el testigo se caiga, con lo que el testigo se pierde. Todos los miembros disponen de un temporizador que se reinicia cada vez que el miembro recibe una comunicación. Si el temporizador vence, el miembro trata de regenerar el testigo, según se detalla en el apartado posterior de recuperación del testigo. El segundo problema viene asociado a éste y consiste en que el miembro considerado caído fuera simplemente muy lento y otro genere un nuevo testigo, terminando el grupo con varios testigos activos. Este punto se soluciona con que cada miembro admita sólo un testigo como válido, con lo que no es posible que un mismo miembro transmita dos o más testigos diferentes. Esta solución conlleva que el grupo puede particionarse, pero no se compromete la integridad de sus comunicaciones.

Este paso de testigo, realizado a través de la operación *receiveToken*, implica que el grupo está realizando comunicaciones continuamente, incluso cuando la aplicación está inactiva sin enviar mensajes. El sistema puede detectar esta inactividad y retrasar el envío del testigo entre miembros en estos casos. En cualquier caso, otros sistemas no basados en paso de testigo deben realizar también comunicaciones periódicas para detectar caídas de miembros.

8.2.2. Detección de errores

La detección de errores se realiza a partir de las comunicaciones normales: envío de mensajes y vistas, paso de testigo, etc. Si un miembro no puede acceder a otro, lo considera caído. Adicionalmente, el miembro contactado debe responder en un tiempo límite, configurable según sea la calidad de la red.

Puesto que el paso del testigo se realiza continuamente, la caída del miembro se detectará de forma razonablemente rápida, salvo que el miembro que posee el testigo sea el que se haya caído. Por esta razón, todo miembro espera recibir alguna comunicación del grupo en un tiempo dado y si no se produce, inicia el protocolo de regeneración del testigo.

Problemas en la red o miembros lentos pueden suponer detecciones de errores erróneas. Sin embargo, una vez que un miembro considera a otro erróneo tal decisión es permanente, y lo comunica a los demás miembros del grupo que pasan a considerarlo erróneo igualmente. En estas condiciones, rechazan cualquier comunicación proveniente de este miembro falsamente erróneo.

A la inversa, si un miembro accede a otro y este último rechaza la comunicación, el primero le considera erróneo igualmente y se lo comunica a los demás. Esta situación puede suponer la división del grupo, pero sólo uno, o incluso ninguno, de los nuevos grupos tendrá consenso y sobrevivirá, puesto que SenseiGMS no soporta particionados de la red. Los sistemas de sincronía virtual extendida permiten que varios subgrupos permanezcan activos tras su división por problemas de comunicaciones en la red; si esta funcionalidad no se soporta, debe asegurarse que sólo uno de los grupos, como máximo, permanezca activo. Este objetivo se logra imponiendo un requisito sobre las vistas: una vista temporal sólo se acepta si contiene, al menos, la mayoría de los miembros que se incluían en la anterior vista *permanente*. Esta implementación implica que un grupo de dos miembros donde uno realmente cae, perderá el consenso y se volverá inoperativo; por esta razón, SenseiGMS permite definir grupos como pertenecientes a una red fiable, relajando, en este caso, el anterior requisito y permitiendo que el consenso se mantenga con vistas conteniendo sólo la mitad de los miembros de la anterior vista permanente. En estos casos, decimos que SenseiGMS trabaja con *consenso débil*.

8.2.3. Envío de mensajes

El modelo de sincronía virtual no prescribe el soporte de mensajes punto a punto entre los miembros de un grupo; su soporte bajo un algoritmo de paso de testigo es sin embargo sencillo, incluso respetando el orden entre mensajes punto a punto y mensajes al grupo. Es además importante: aunque bajo CORBA o JavaRMI cada componente puede realizar comunicaciones punto a punto con otros componentes, la información incluida en la vista no contiene las referencias a otros miembros, sino sus identidades, una abstracción que sólo tiene significado bajo SenseiGMS.

Un miembro encola todos los mensajes a enviar, que procesa cuando recibe el testigo. Bajo condiciones normales, cuando recibe ese testigo, itera sobre la cola de mensajes, enviándolos al miembro especificado, o a todos los miembros si es un mensaje multipunto. Si tiene varios mensajes multipunto, los puede agrupar en un sólo envío, minimizando el número de comunicaciones y delegando en el *middleware* la división del mensaje en submensajes si aquél resulta demasiado grande. Para el envío de mensajes, la interfaz *GroupHandler* incluye tres operaciones:

- *receiveMessages*: recibe conjuntos de mensajes multipunto.
- *receivePTPMessages*: la operación equivalente para recibir mensajes punto a punto.

- *confirmMessages*: completa el envío de los mensajes, que se realiza en dos pasos: el miembro envía el mensaje y, a continuación, confirma que se puede procesar. Como un miembro puede enviar varios mensajes, el envío de un segundo mensaje directamente confirma el primero y sólo es necesario realizar una confirmación de mensajes en el último paso, antes de liberar el testigo. Los miembros del grupo procesan los mensajes multipunto sólo cuando reciben esta confirmación, mientras que los mensajes punto a punto no necesitan, evidentemente, una confirmación.

Si el miembro se cae antes de enviar todos los mensajes, el algoritmo de recuperación del testigo debe verificar los mensajes procesados por los miembros activos, de tal forma que esos mensajes sean procesados por los demás miembros. Cada mensaje incluye, por lo tanto, una identidad de mensaje, con la vista en que se envía, y un número secuencial que se reinicia en cada nueva vista. La recepción de un mensaje con mayor número secuencial confirma automáticamente al anterior. Los mensajes deben, entonces, encolarse cuando se reciben y cada miembro almacena la identidad del último mensaje procesado, que es utilizado en el algoritmo de recuperación del testigo. Esta identidad de mensaje, que no se mostró con la interfaz privada, es en OMG/IDL:

```
struct MessageId
{
   long view;
   long id;
};
```

Un miembro que recibe un mensaje punto a punto de otro miembro da automáticamente por confirmados sus anteriores mensajes multipunto; sin embargo, el mensaje punto a punto no puede incrementar su identidad de mensaje (*id*, el número secuencial), ya que no lo reciben todos los miembros. Por ello, en la identidad de los mensajes punto a punto, sólo tiene significado la identidad de la vista y no el número secuencial asociado.

8.2.4. Manejo de vistas

Todos los miembros del grupo deben recibir las mismas vistas. Éstas pueden ser temporales, cuando los miembros discuten su contenido, o permanentes. La aplicación sólo recibe las permanentes y se garantiza que todos los miembros de la aplicación reciben las mismas vistas. Sin embargo, no es necesario que todos los miembros reciban las mismas vistas temporales.

Las vistas se envían a través de la operación *receiveView* en la interfaz *SenseiGMSMember*. Una vista es enviada cuando un miembro desea cambiar la composición del grupo, bien sea por la inserción de nuevos miembros, o tras sospechar que un miembro ha caído y debe ser expulsado. Cada vista viene

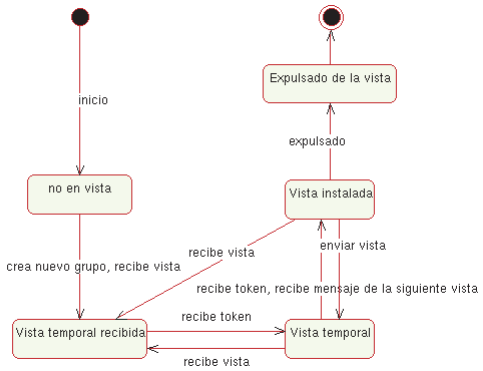

Figura 8.1. Estados de *vistas* en SenseiGMS

identificada por una identidad de vista, e incluye la lista ordenada de los demás miembros del grupo y sus identidades, así como las de los miembros que son nuevos en el grupo (identidades que no son luego propagadas con las vistas públicas). Esta lista es ordenada, pero no se puede asegurar que el primer miembro sea el más antiguo del grupo.

Siguiendo el modelo de sincronía virtual, los mensajes deben ser procesados por todos los miembros en la misma vista. Los cambios de vistas bloquean, por lo tanto, este envío. Cada miembro válido puede estar en tres estados: *vista instalada*, *vista temporal*, y *vista temporal recibida*. Cuando un miembro en estado de *vista instalada* envía una nueva vista, debe primero enviar todos los mensajes y bloquear el envío de nuevos mensajes, antes de pasar a estado de *vista temporal*. En este estado, cuando la aplicación solicita el envío de un mensaje, se le notifica que no será enviado hasta la siguiente vista.

Las vistas se instalan en dos pasos; un miembro envía su vista, que siempre presenta una identidad temporal y, a continuación, pasa el testigo al siguiente miembro de la vista. Todos los miembros recibirán este testigo y podrán cambiar el contenido de la vista; en este caso, serán nuevas vistas temporales. Si el miembro que envía la vista recibe el testigo sin que la vista haya cambiado, considerará que la vista es válida y la propagará como permanente a la aplicación.

Cuando un miembro recibe una vista, pasa a estado de *vista temporal recibida*. Cuando reciben el testigo en la siguiente vuelta, tienen una oportunidad para enviar sus mensajes y, a continuación, pasan a estado de *vista temporal*, bloqueando igualmente los mensajes. Un miembro en *vista temporal* pasa a *vista instalada*

cuando recibe el testigo en la siguiente vuelta; si, estando en *vista temporal*, recibe una vista de otro miembro, pasa a *vista temporal instalada* sin desbloquear los mensajes. El diagrama de estados en la figura 8.1 visualiza este algoritmo.

Hay un caso especial: un miembro en estado de *vista temporal* que recibe un mensaje perteneciente a una vista posterior. Este caso implica que otro miembro en el anillo ha pasado ya a estado de *vista instalada* y envía nuevos mensajes; como consecuencia, pasa, automáticamente, a estado de *vista instalada* y lo comunica a la aplicación.

8.2.5. Gestión de grupos

Un grupo se crea a partir de un único miembro, que crea un testigo inicial que se envía a sí mismo. Esta aproximación se puede optimizar para evitar el paso del testigo cuando hay un solo miembro en el grupo. Para extender el grupo, el potencial miembro debe contactar a alguno de los miembros válidos, y solicitar su inserción. Este miembro encolará esa solicitud y, cuando reciba el testigo, considerará si el estado del grupo permite la inserción de nuevos elementos. Esta consideración se tiene en cuenta a continuación, al explicar los cambios de vistas.

Los siguientes métodos de la interfaz *SenseiGMSMember* son necesarios para cubrir esta funcionalidad:

- *createGroup*.
- *addGroupMember*. Incluye eventualmente al miembro especificado como parámetro de la operación en el grupo propio.
- *joinGroup*. Es un método redundante, definido en función de *addGroupMember*, que bloquea la llamada hasta que se completa la inserción en el grupo.

Tras recibir una solicitud de inclusión en el grupo, el miembro contactado añadirá, eventualmente, a ese miembro o nuevos miembros, creando una nueva vista que envía al grupo que incluye a esos nuevos miembros; éstos, al recibir esta vista, pueden ya considerarse aceptados en el grupo. Este proceso puede fallar si el miembro contactado se cae o pierde el consenso, en cuyo caso, los nuevos miembros deberán repetir la solicitud a otro miembro. La obtención de referencias a miembros existentes no queda cubierta en la definición del algoritmo.

Para abandonar voluntariamente un grupo, un miembro debe enviar, cuando recibe el testigo, una vista que no le incluya y, a continuación, liberar el testigo. Un miembro que abandona de esta forma el grupo afecta a la forma en que se resuelve el consenso, excluyéndolo de la anterior vista permanente al contabilizar la mayoría. Por ejemplo, en condiciones normales, un grupo de dos miembros donde uno cae pierde el consenso, pero esto no ocurre si el miembro solicita su exclusión.

Este protocolo no soporta el concepto de grupos *exclusivos*: cualquier instancia del tipo *SenseiGMSMember* puede solicitar su inclusión en otro grupo si obtiene una

referencia válida y ninguno de los miembros existentes puede vetar tal inclusión. Desde el punto de vista de seguridad, implica, por ejemplo, que un miembro podría ser incluido y enviar entonces vistas erróneas diferentes a distintos miembros del grupo, provocando que éste se particione en grupos sin consenso. Esta seguridad puede, sin embargo, incluirse en una capa superior. Por ejemplo, empleando JavaRMI, es posible especificar una factoría propia de *sockets* que emplee sockets seguros o valide todo inicio de comunicaciones: sin afectar a SenseiGMS, la aplicación ha superpuesto un nivel de seguridad en las comunicaciones.

8.2.6. Protocolo de recuperación del testigo

Un miembro que no recibe ninguna comunicación tras un determinado periodo de tiempo considera que el testigo se ha perdido e inicia el protocolo de recuperación de aquél. Este protocolo se realiza en tres fases. En la primera fase, el miembro contacta a los demás miembros a través de la operación *allowTokenRecovery*, solicitándoles su permiso para recuperar el testigo. Si alguno de los miembros responde negativamente, este miembro aborta la recuperación del testigo.

Un miembro responde negativamente por dos posibles razones: porque posea el testigo, o porque tenga prioridad para recuperarlo. La segunda razón busca evitar que dos miembros recuperen simultáneamente el testigo. Un miembro considera que tiene prioridad sobre otro si su identidad de vista (temporal/permanente) es superior a la del otro miembro y, cuando es igual, si su identidad de miembro de grupo es superior a la del otro miembro; de esta forma, si un miembro ha enviado su propuesta de vista a un subconjunto del grupo y se cae, ese subconjunto tiene prioridad para instalar una nueva vista. La solicitud de recuperación del testigo incluye, por lo tanto, la identidad de vista del miembro que envía la solicitud. Al responder, los miembros incluyen, igualmente, la identidad del último mensaje procesado, de tal manera que sea posible conocer la mayor identidad de mensaje procesado por algún miembro en el grupo.

En la segunda fase, el miembro considera que puede recuperar el testigo, y envía su decisión a los demás miembros, empleando la operación *recoverToken*. Cuando estos miembros reciben esta señal, consideran que el testigo se ha perdido y que está siendo regenerado por el miembro dado. El grupo no aceptará entonces ninguna nueva comunicación del grupo (paso de testigo, o mensajes, etc.), aceptando únicamente otra comunicación *recoverToken* o un cambio de vista proveniente del miembro que está recuperando el testigo. De esta manera, se impide que el anterior testigo cause problemas si el miembro que lo poseía era muy lento o simplemente había un problema en las comunicaciones de red. Se impide también que dos miembros puedan recuperar el testigo simultáneamente, también por problemas de comunicaciones. Si fuera éste el caso, el grupo se dividiría entre los miembros que han recibido las solicitudes de recuperación de cada miembro; eventualmente se puede provocar que el grupo se vuelva inoperativo, pero impide la

inconsistencia de que dos miembros consideren que poseen el testigo y envíen simultáneamente comunicaciones contradictorias al grupo.

En la tercera fase, el miembro envía la vista a los miembros. Esta vista excluye a los miembros que no pudo contactar en la primera o segunda fase y a los miembros que respondieron negativamente al evento de recuperación del testigo. Si tras estas exclusiones el miembro ha perdido el consenso, se considera automáticamente excluido.

Un miembro que es expulsado del grupo cuando no tiene el testigo ejecutará eventualmente este protocolo, descubriendo que no tiene consenso para continuar operativo, enviando entonces el evento de exclusión de grupo a la aplicación. Si el miembro es expulsado cuando tiene el testigo, fracasará en su intento de transferirlo y, de nuevo, al intentar instalar una vista, detectando entonces su pérdida de consenso igualmente.

8.3. Uso de SenseiGMS

El siguiente capítulo, dedicado a la metodología de diseño de grupos de objetos replicados, se centra en los problemas generales al diseñar aplicaciones tolerantes a fallos y muestra métodos genéricos y las herramientas necesarias para soportar esa metodología. Esta sección se limita a mostrar un ejemplo particular sobre SenseiGMS, obviando, al menos, un problema de uso que con las herramientas mostradas en el siguiente capítulo tendría una fácil solución.

8.3.1. Diseño de un servicio replicado

La sección dedicada al diseño de SenseiGMS mostró su interfaz pública y las directrices generales para emplear replicación de objetos: definir los mensajes que se emplearán en las comunicaciones del grupo, implementar la interfaz *GroupMember* y obtener una referencia a un objeto *GroupHandler*, ya sea creando un nuevo grupo o incluyéndose en uno existente; mediante esta referencia se accede al grupo, y es preciso procesar los mensajes provenientes de éste.

El ejemplo a implementar es un caso real: un servicio de directorio de miembros de grupos, de tal forma que nuevos miembros puedan obtener fácilmente referencias a otros miembros ya existentes en un determinado grupo, que se nombra mediante una cadena de caracteres. Este es uno de los servicios existentes en Sensei, denominado SenseiGMNS, con una interfaz más complicada para ser funcional a los clientes del grupo y no sólo a sus miembros.

Para implementar este servicio, empleamos una estructura de datos denominada *GMNSdata*, que contiene una lista de todos los grupos existentes y, para cada grupo, la lista de miembros asociados. Si la máquina que contiene esta

estructura se cae, el servicio se vuelve inaccesible, por lo que la implementación debe ser tolerante a fallos: es un componente replicado.

La definición de este componente es, en OMG/IDL:

```
interface GMNSdata : GroupMember
{
   void insert (in string groupName, in GroupHandler member, in string memberName);
   void remove (in string groupName, in GroupHandler member);
   GroupHandler get (in string groupName);
};
```

Las operaciones se definen como:

- *insert*: incluye un miembro en el grupo dado, creando el grupo si es necesario. Cada miembro tiene un nombre, solamente útil para su visualización.
- *remove*: elimina un miembro del grupo y el grupo mismo si se queda vacío.
- *get*: devuelve un elemento cualquiera de un grupo.

Un miembro que desea incluirse en un determinado grupo, accede a este componente[3] mediante la operación *get* para obtener referencias a otros miembros del grupo. Si no hay referencias, crea el grupo e introduce en este componente su referencia con la operación *insert*. Si hay otros miembros, *get* devuelve uno de esos miembros, al que solicita su inclusión en el grupo, insertando finalmente en este componente su referencia. Sin embargo, hay problemas de concurrencia: puede darse el caso de dos futuros miembros de un mismo grupo, todavía no existente, que accedan a este componente, observando ambos que el grupo no existe y creando entonces dos grupos independientes, para insertar, a continuación, sus referencias bajo el mismo nombre de grupo. Para este ejemplo, obviamos este problema, que se trata en los siguientes capítulos.

Si un componente *GMNSdata* recibe una solicitud *insert* o *remove*, debe comunicárselo a las demás réplicas *GMNSdata* de tal forma que todo el grupo mantenga el mismo estado. Si la solicitud es *get*, el componente puede dar esa información sin realizar ninguna comunicación al grupo. Definimos dos mensajes:

```
valuetype GMNSdataInsertMessage :
          Message
{
   public string groupName;
   public string memberName;
   public GroupHandler member;
};
```

```
valuetype GMNSdataRemoveMessage :
    Message
{
   public string groupName;
   public GroupHandler member;
};
```

[3] Tal acceso no es directo, un componente intermedio, el GMNS, es el encargado de acceder ordenadamente al *GMNSdata*, evitando colisiones bajo múltiples solicitudes.

Cada mensaje incluye la información necesaria para poder ejecutar en cada réplica el mismo código, lo que, en general, supone incluir como atributos cada uno de los parámetros definidos en la operación asociada.

Desde un punto de vista formal, no es preciso que *GMNSdata* herede de *GroupMember*: es suficiente que se defina un componente que implemente ambas interfaces.

8.3.2. Implementación

La implementación es diferente si se emplea JavaRMI o CORBA y, en este último caso, varía según el lenguaje de implementación escogido y el adaptador de objetos empleado. En el caso de emplear Java bajo CORBA, usando el *Portable Object Adapter*, una implementación posible es crear una clase que herede de *GMNSdataPOA*, que es generada automáticamente por el compilador de OMG/IDL. Con RMI, una posibilidad es crear una clase que herede de *java.rmi.server.UnicastRemoteObject* e implemente la interfaz *GMNSdata*. En ambos casos, el algoritmo es el mismo y, puesto que en los dos casos se emplea Java, el código que implementa la funcionalidad *GMNSdata* es común para CORBA y RMI. Para no entrar en detalles innecesarios sobre este ejemplo concreto, suponemos que existe una estructura de datos interna capaz de manejar la lógica del problema (sin replicación) y con las mismas operaciones definidas en la interfaz *GMNSdata* (sin incluir las operaciones de *GroupMember*). La instancia de esta estructura la denominamos *internalGMNSData*.

La figura 8.2 muestra la idea básica para el procesado de mensajes en este ejemplo: al recibir una solicitud de servicio, la réplica no la procesa inmediatamente, sino que envía un mensaje al grupo con la información sobre la operación a realizar. El miembro que envía el mensaje, también lo recibe, en el orden adecuado, y es entonces cuando procesa la operación. La ventaja de esta aproximación es que hay un único punto de procesamiento, tanto para las operaciones directamente invocadas en esta réplica, como para los mensajes provenientes de esas mismas operaciones al ser invocadas sobre otras réplicas del grupo.

Este ejemplo es un caso simplificado pues, normalmente, el servidor debe devolver algún valor al cliente y es entonces necesario sincronizar de alguna manera la invocación de la operación con su procesamiento, como detallamos en el siguiente capítulo. Incluso en este simple ejemplo, no puede garantizarse que la siguiente porción de código se ejecute satisfactoriamente:

```
map.insert(“Grupo A”, specificGroupHandler, “Miembro 1”);
if (map.get(“Grupo A”)==null)
{
   //Error, el grupo no tiene miembros tras haber insertado uno?
}
```

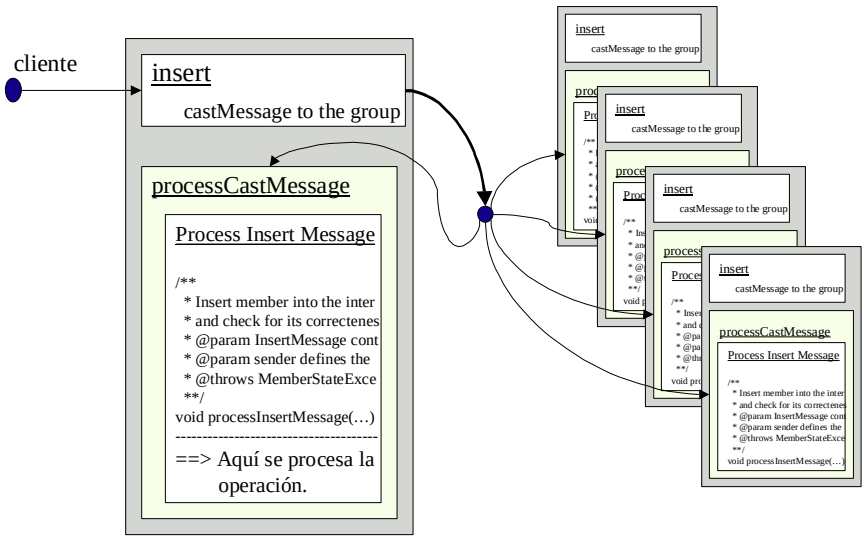

Figura 8.2. Tratamiento de mensajes empleando orden total.

El motivo de este error es que cuando la operación *insert* vuelve, no ha sido aún procesada, por lo que el valor devuelto por la operación *get* puede indicar que el grupo dado no tiene ningún miembro. Y si la operación se sincroniza, la principal desventaja es el mayor tiempo de proceso que el cliente experimenta. En la siguiente lista aparecen en cursiva los pasos que deben ser ejecutados en adición al proceso normal, si el servidor no estuviera replicado:

- Recibir la petición del cliente.
- *Crear un mensaje con la información de la operación a procesar.*
- *Enviar este mensaje al grupo.* Lo cual incluye enviar el mensaje a cada una de las réplicas garantizando que todas obtienen una copia. En el peor caso, implica enviar a cada réplica tres mensajes.
- *Recibir el mensaje.*
- Procesar la operación solicitada.
- *Sincronizar este procesamiento con la operación invocada para poder devolver al cliente los resultados de esa operación.*
- Devolver al cliente los resultados de la operación, si fuera necesario.

Estos pasos adicionales deben ejecutarse en cualquier caso en un componente replicado activamente, pero el cliente no experimenta estos mayores tiempos de procesamiento si el sistema se programa de tal forma que los mensajes sean dinámicamente no uniformes. En ese caso, cuando un cliente invoca la operación *insert*, el servidor la procesa inmediatamente y envía a continuación (o incluso

concurrentemente) el mensaje al grupo, pero el cliente observa el mismo tiempo de procesamiento que si el servidor no estuviera replicado. Obviamente, el servidor tiene ahora una mayor carga de trabajo, por lo que no será capaz de soportar el mismo número de operaciones por unidad de tiempo.

Emplear mensajes dinámicamente no uniformes implica normalmente cambiar los algoritmos de los grupos, pues no todas las réplicas procesan los mismos mensajes en el mismo orden: la réplica que envía el mensaje no lo procesa. Por ejemplo, al programar una estructura de datos que implemente un mapa replicado, estos mensajes producirán, normalmente, inconsistencias en el grupo: si dos miembros insertan concurrentemente dos valores distintos para una misma clave, las réplicas terminarán con estados diferentes. Esta restricción no se da, sin embargo, en el caso de *GMNSdata*, donde el valor de cada entrada del mapa es, a su vez, un conjunto de valores sin requerimientos de orden.

El siguiente código implementa un servidor replicado *GMNSdata* usando SenseiGMS bajo JavaRMI y empleando mensajes dinámicamente uniformes:

```
public class GMNSdataImpl extends UnicastRemoteObject implements GMNSdata
{
   public GMNSdataImpl throws RemoteException {}

   public void insert(String groupName, GroupHandler member, String memberName)
   {
      handler.castMessage(new GMNSdataInsertMessage(groupName, member, memberName));
   }

   public void remove(String groupName, GroupHandler member)
   {
      handler.castMessage(new GMNSremoveMessage(groupName, member));
   }

   public GroupHandler get(String groupName)
   {
      return internalGMNSdata.get(groupName);
   }

   public void processCastMessage(int snd, Message msg)
   {
      if (msg instanceof GMNSdataInsertMessage)
      {
         GMNSdataInsertMessage iMsg = (GMNSdataInsertMessage) msg;
        internalGMNSdata.insert(iMsg.groupName, iMsg.member, iMsg.memberName);
      }
      else if (msg instanceof GMNSdataRemoveMessage)
      {
```

```
            GMNSdataRemoveMessage rMsg = (GMNSdataRemoveMessage) msg;
            internalGMNSdata.remove(rMsg.groupName, rMsg.member);
        }
    }

    public void changingView() {}

    public void void installView(View view) {}

    public void processPTPMessage(int snd, Message msg) {}

    public void memberAccepted(int id, GroupHandler handler, View view)
    {
        this.handler = handler;
    }

     public void excludedFromGroup()
    {
        System.exit(0);
    }

    GroupHandler handler;
}
```

- Este servidor no necesita información sobre las demás réplicas, luego no procesa las vistas que recibe, y no necesita código en las operaciones *installView* o *changingView*. Cuando el miembro es aceptado en el grupo, recibe mediante *memberAccepted* una instancia del tipo *GroupHandler*, que guarda para utilizar posteriormente en sus interacciones con el grupo.
- El algoritmo no precisa de mensajes punto a punto, luego tampoco es necesario ningún código en *processPTPMessage*. Al recibir mensajes del grupo en *processCastMessage* debe primero discernir el tipo de mensaje recibido, procesándolos acordemente. Si el mensaje es *GMNSdataInsertMessage*, lee la información del mensaje e invoca esa operación sobre la estructura de datos interna, tratando de forma similar el mensaje *GMNSdataRemoveMessage*.
- Las operaciones *insert* y *remove* se tratan creando los mensajes correspondientes y enviándolos al grupo a través de la instancia *GroupHandler* recibida cuando el miembro fue aceptado.
- La operación *get* es procesada inmediatamente, empleando la estructura de datos interna.

El código mostrado no trata errores (la instancia de *GroupHandler* es remota, luego su acceso puede siempre provocar excepciones) ni realiza transferencia de estado, pero muestra las características fundamentales de SenseiGMS: orientación a

objetos, empleo de mensajes tipados, comunicaciones en grupo fiables con orden total. Es posible llevar la orientación a objetos más lejos empleando herramientas genéricas que definan los mensajes automáticamente a partir de una interfaz dada y transformen la recepción de mensajes en llamadas a procedimientos específicos según el mensaje recibido; este tema se trata también en el siguiente capítulo.

Tampoco muestra este código cómo se crea o se extiende el grupo: si la implementación se realiza totalmente en Java, la clase *ActiveGroupMember* en el paquete *sensei.GMS* implementa la interfaz *SenseiGMSMember*, necesaria para la creación o extensión de grupos. El primer miembro crea una instancia de esta clase e invoca la operación *createGroup*. El segundo y posteriores miembros deben obtener de alguna manera una referencia a este miembro, por ejemplo, accediendo a un fichero donde esa referencia se haya guardado. Estos miembros crean de la misma forma una instancia de la clase *ActiveGroupMember* e invocan la operación *joinGroup*, pasando la referencia del primer miembro del grupo. Este proceso se realiza automáticamente y sin necesidad de discernir entre 'primero' o 'posterior' miembro del grupo empleando SenseiGMNS, detallado en el capítulo 11.

8.3.3. Configuración

El algoritmo emplea varias variables que pueden cambiar el comportamiento del sistema, permitiendo así su optimización en función del entorno de trabajo. Estas propiedades son:

- GMS.*channelLiveness*: periodo en milisegundos en que un canal de comunicación espera una respuesta. Transcurrido este tiempo, el miembro correspondiente se considera caído. Por defecto son 10 segundos.
- GMS.*maxProcessMaxDelay*: periodo en milisegundos que define el retraso máximo en que la aplicación puede procesar un mensaje dado. Transcurrido este tiempo, que por defecto es de 5 segundos, Sensei considerará que la aplicación es muy lenta y excluirá al correspondiente miembro del grupo.
- GMS.*tokenStoppedPeriod*: retención del testigo en caso de inactividad en el sistema. Cuando el sistema no realiza ninguna comunicación, cada miembro retiene el testigo el tiempo dado por esta variable, 50 milisegundos por defecto.
- GMS.*tokenLostTimeout*: periodo en milisegundos que tarda un miembro que no recibe ninguna comunicación en iniciar el algoritmo de recuperación del testigo. Por defecto, es de 6 segundos.
- GMS.*weakConsensus*: si esta variable booleana se define *a true*, se emplea un modelo de consenso débil, donde es suficiente que la mitad de los miembros de una vista sobrevivan para que el grupo conserve el consenso.

Estas propiedades se definen en un fichero específico, aunque cada grupo puede definir configuraciones distintas o, simplemente, redefinir alguno de los

anteriores valores. No es necesario tampoco que todos los miembros de un mismo grupo definan la misma configuración.

8.4. VirtualNet

SenseiGMS se ha probado en profundidad, comprobando situaciones límite mediante el empleo de instancias de *SenseiGMSMember* con una implementación deliberadamente errónea del algoritmo expuesto. Estas alteraciones del algoritmo incluyen:

- Los mensajes no son enviados a todos los miembros en el grupo.
- La vista no se envía a todos los miembros.
- La confirmación de los mensajes no se envía a todos los miembros.
- Los mensajes no se envían hasta que se recibe un cambio de vista.
- Los mensajes no se envían hasta la siguiente vista.
- Se retrasa la inserción de los miembros que lo solicitan hasta que se reciba un cambio de vista.
- El paso del testigo se realiza lentamente.
- Miembros se caen en situaciones comprometidas.

Comprobar el comportamiento de Sensei en condiciones reales es una tarea más complicada, al precisar de entornos muy variados. Por ejemplo, comprobar un enlace más lento, o la caída de uno o más enlaces, es difícil de realizar, incluso cuando sea posible realizar la desconexión manual de esos enlaces físicos. Un entorno real puede involucrar ordenadores físicamente muy distantes, conectados por Internet a través de un número desconocido de nodos, donde estos nodos pueden caerse o donde algunas conexiones pueden tener grandes fluctuaciones de rendimiento. Para solucionar este problema, hemos desarrollado una aplicación independiente de Sensei denominada *VirtualNet*.

La filosofía de VirtualNet es simple: simular una red en un solo ordenador, donde se pueden definir nodos y enlaces que pueden alterarse en cualquier momento, ya sea definiendo diferentes rendimientos o desactivando temporal o permanente esos nodos o enlaces. Cada miembro debe asociarse a un nodo, y sus comunicaciones con otros miembros suponen que debe existir una ruta virtual entre sus nodos.

La versión inicial de VirtualNet sólo funciona con JavaRMI, empleando un compilador propio para generar los *stubs* y *skeletons*, que verifican esa ruta virtual antes y después de acceder al objeto remoto. La forma de asociarse a un nodo es empleando el servicio de nombrado de RMI (*java.rmi.Naming*), que se redefine para VirtualNet. Así, en lugar de emplear *java.rmi.Naming*, se emplea *vnet.remote.Host*,

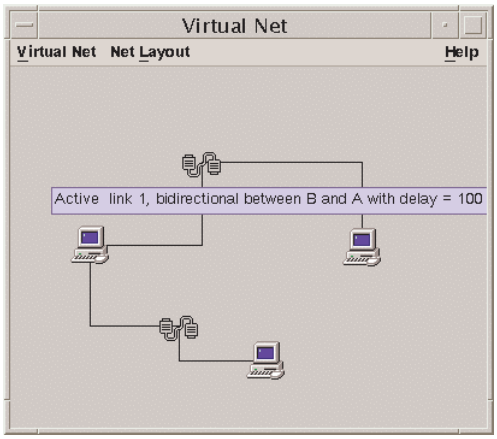

Figura 8.3. VirtualNet

que define exactamente la mismo interfaz, facilitando la prueba de las aplicaciones. Sólo debe cambiarse el nombre con que se publica la aplicación, que debe ser del tipo: `/virtualNetName/Host/Server` (nombre de la red virtual creada, máquina a emplear, nombre del servidor). Por ejemplo, `/redMadrid/MaquinaUCM/GMNS`.

Una aplicación no necesita, sin embargo, emplear el registro de RMI para operar. Este registro es sólo una forma cómoda de obtener referencias a otros objetos remotos existentes. De hecho, Sensei no emplea el registro en ningún momento. Por esta razón, se ha desarrollado una nueva versión que, por el contrario, funciona de momento sólo con CORBA. VirtualNet contiene ahora la lógica para comprobar las rutas virtuales entre dos nodos cualesquiera, nodos a los que se les puede asignar cualquier nombre. Puede soportar múltiples redes virtuales, cada una de las cuales se instala en un determinado puerto. La aplicación final debe conectarse a este puerto, indicando el nombre de la máquina en que reside y, a continuación, puede consultar indefinidamente si hay rutas virtuales a otras máquinas. VirtualNet dispone de una interfaz gráfica que permite visualizar dinámicamente el estado de la red virtual, tal como muestra la figura 8.3.

Los enlaces pueden ser monodireccionales o bidireccionales y, tanto los nodos, como los enlaces, pueden tener un retraso asociado que se procesa en cada chequeo de ruta. Es decir, al contactarse VirtualNet para chequear si una ruta es válida o no, una respuesta positiva se efectuará en el tiempo necesario para recorrer virtualmente cada uno de los nodos y enlaces en la ruta calculada. Pueden realizarse varios chequeos simultáneamente.

El protocolo de acceso a VirtualNet es muy simple y una aplicación en cualquier lenguaje puede emplearlo para comprobar las rutas. Sin embargo, es necesario que estos accesos sean lo más transparentes posible para que el empleo de esta herramienta sea práctico. Bajo CORBA es posible implementar *interceptors*, cuyo objetivo es interceptar cualquier acceso remoto. VirtualNet incluye interceptores que acceden a la red virtual en cada acceso y emiten excepciones *org.omg.CORBA.COMM_FAILURE* cuando la ruta no es válida. Para ello, añaden en cada comunicación el nombre de la máquina que realiza la comunicación y, cuando el servidor recibe esa comunicación, puede chequear la ruta. De la misma forma, cuando el servidor devuelve la respuesta al cliente, éste puede volver a comprobar la ruta.

Para que una aplicación emplee VirtualNet debe incluir la siguiente línea de código al principio del programa, antes de iniciar el ORB[4]:

```
args=vnet2.OOCInterceptor.setup(null, args, props);
```

El primer parámetro indica el receptor de los mensajes de error que, por defecto, es el dispositivo normal de salida. El segundo parámetro contiene los argumentos de la línea de comando y, el tercero, las propiedades con que se inicializará posteriormente el ORB. Los argumentos asociados a VirtualNet son eliminados para no interferir con la lógica de la aplicación. Estos argumentos requeridos son:

- *-virtualNetHost host*: indica la máquina donde se ejecuta el programa de VirtualNet que, por defecto, es "*localhost*".
- *-virtualNetPort port*: indica el puerto donde VirtualNet espera conexiones.
- *-virtualHostName name*: indica el nombre de la máquina virtual, que deberá estar definido en la red virtual asociada.

Si los dos últimos argumentos están presentes, el interceptor se instala como cliente y como servidor, y se emplea entonces activamente la red virtual.

8.5. Conclusiones

SenseiGMS es un sistema de comunicaciones fiables en grupo que implementa sobre CORBA y JavaRMI una interfaz de programación de aplicaciones similar al soportado en otros sistemas de comunicaciones en grupo. La funcionalidad implementada se reduce a la mínima común con esos sistemas de comunicaciones, de tal forma que la implementación posterior, tanto de los algoritmos de

[4] En este ejemplo se está empleando el interceptor OOC, específico para *Orbacus*.

transferencia de estado expuestos, como de las herramientas de apoyo al desarrollo de aplicaciones replicadas mostradas en los siguientes capítulos, pueda reimplementarse sin cambios en cualquier sistema de comunicaciones en grupo.

La necesidad de SenseiGMS se ha basado en la no disponibilidad de sistemas similares sobre CORBA o JavaRMI. La especificación del servicio de tolerancia a fallos de CORBA define, para la replicación activa, un grupo de comunicaciones en sincronía virtual situado lógicamente por debajo del ORB, con el objetivo de lograr un mejor rendimiento. SenseiGMS implementa el sistema de comunicaciones *sobre* el ORB, degradando su rendimiento en beneficio de una simplificación en su implementación. Cuando esté disponible una implementación compatible con la actual especificación, confiamos en la fácil migración de la interfaz de SenseiGMS sobre esa implementación, teniendo en cuenta la mínima funcionalidad soportada.

SenseiGMS define, sin embargo, dos facilidades adicionales sobre otros sistemas de comunicaciones en grupo tradicionales: orientación a objetos y soporte de mensajes tipados. Estas dos facilidades no resultan, por otro lado, difíciles de implementar sobre esos sistemas.

El objetivo de simplificar la implementación de SenseiGMS afecta, también, a su soporte de orden en los mensajes fiables, admitiendo sólo orden total. Este orden se obtiene mediante un algoritmo de paso de testigo, diseñado sobre mensajes fiables punto a punto. Las aplicaciones replicadas se programan más fácilmente empleando orden total que causal o *fifo*, y el diseño de herramientas genéricas está también basado en ese orden, lo que garantiza que esta simplificación, específica en la funcionalidad de SenseiGMS, no afecta al dominio de aplicaciones que soporta.

Probar la corrección de SenseiGMS, así como de las aplicaciones que lo usan, implica la misma problemática asociada a sistemas distribuidos, donde deben probarse procesos ejecutándose sobre múltiples máquinas. Aquí se añade el problema de probar los enlaces mismos, pues los algoritmos empleados son muy dependientes de cualquier retraso o alteración en las comunicaciones. Con el objetivo de cubrir esta necesidad, VirtualNet permite simular una red de cualquier complejidad sobre una única máquina, ofreciendo facilidades para alterar su topología y la calidad de las comunicaciones en cualquier momento. Además, su empleo no implica importantes cambios en la aplicación a probar, limitándose su impacto a la adición de una simple línea de código al principio del programa.