Capítulo 10 - SENSEIDOMAINS

SenseiDomains es la implementación del soporte de la metodología desarrollada en el anterior capítulo, para el desarrollo de aplicaciones tolerantes a fallos mediante el empleo de componentes replicados.

La idea básica es que todas las comunicaciones deben realizarse a través de SenseiDomains, el cual las delega a su vez al sistema de comunicaciones en grupo subyacente. Los principales requisitos sobre dicho sistema de comunicaciones en grupo son el empleo de un modelo de sincronía virtual y el soporte de comunicaciones con orden total causal. Aunque el sistema de comunicaciones subyacente pueda incluir otros órdenes menos restrictivos en los mensajes, en la versión actual de SenseiDomains no se han considerado. La implementación actual se realiza sobre SenseiGMS, pero su dependencia con este sistema de comunicaciones se limita a su interfaz pública. No hay ningún detalle de implementación de SenseiGMS sobre el que se base SenseiDomains, lo que asociado a la generalidad de la interfaz de SenseiGMS, implica una fácil migración a otros sistemas de comunicaciones de grupo.

En lo referente a la transferencia de estado, SenseiDomains implementa la interfaz de aplicación mostrada en el capítulo 7, con pequeñas variaciones al no tratarse de una implementación del servicio de tolerancia a fallos de CORBA. Las variaciones introducidas no restan, sin embargo, flexibilidad a la transferencia, que presenta todas las características que se detallaron en aquel capítulo. Esta transferencia de estado se implementa totalmente en SenseiDomains, sin precisar que el sistema de comunicaciones preste un soporte específico de transferencia. El

capítulo 4 se centró en los sistemas de comunicaciones en grupo existentes, mostrando el soporte que presentan a la transferencia de estado, soporte que es, cuando existente, inferior al que precisa SenseiDomains.

Además de la limitación al empleo de comunicaciones con orden total, SenseiDomains no soporta el particionado del grupo. Esa falta de soporte no viene dada por SenseiGMS, que tampoco lo soporta, sino porque, tanto la metodología expuesta en el capítulo anterior, como el modelo de transferencia de estado presentado en los capítulos 5 al 7, son insuficientes para aplicaciones que permiten particiones del grupo.

La metodología expuesta en el capítulo anterior se centró en su aplicabilidad bajo CORBA, pero es igualmente válida para JavaRMI. Al igual que SenseiGMS, SenseiDomains define una interfaz común para JavaRMI y CORBA, compartiendo los algoritmos de implementación. En este capítulo explicamos exclusivamente la interfaz OMG/IDL; esta interfaz es considerablemente más extensa que la necesaria para SenseiGMS, al igual que la implementación resulta también mucho más compleja, por lo que incluir también la descripción de las clases e interfaces Java precisas para trabajar con JavaRMI supondría una extensión innecesaria, ya que sería repetitivo.

La interfaz CORBA se reparte físicamente en ocho partes que agrupan de forma lógica las distintas definiciones; a cada uno de los ficheros le corresponde una sección en este capítulo, para mantener esa agrupación lógica. Todas las definiciones se incluyen en el paquete Java *sensei.middleware.domains*, lo que implica que están definidas en el módulo *domains*, incluido en el módulo *middleware*, definido a su vez en el módulo *sensei*.

El nombre SenseiDomains viene dado por el elemento principal de la metodología, el dominio, que permite agrupar y gestionar los componentes replicados: todas las características de la metodología se definen en torno a este concepto de dominio. La interfaz definida para el manejo de dominios es *DomainGroupHandler*; sin embargo, esta interfaz depende [figura 10.1] en parte de las demás entidades de SenseiDomains, por lo que no se define hasta las últimas secciones.

10.1. Excepciones

En `DomainExceptions.idl` se define una única excepción, correspondiente a las operaciones efectuadas por el dominio bajo un estado incorrecto. Son posibles otras excepciones, pero asociadas a características específicas del dominio, definidas junto a esas características en las secciones siguientes.

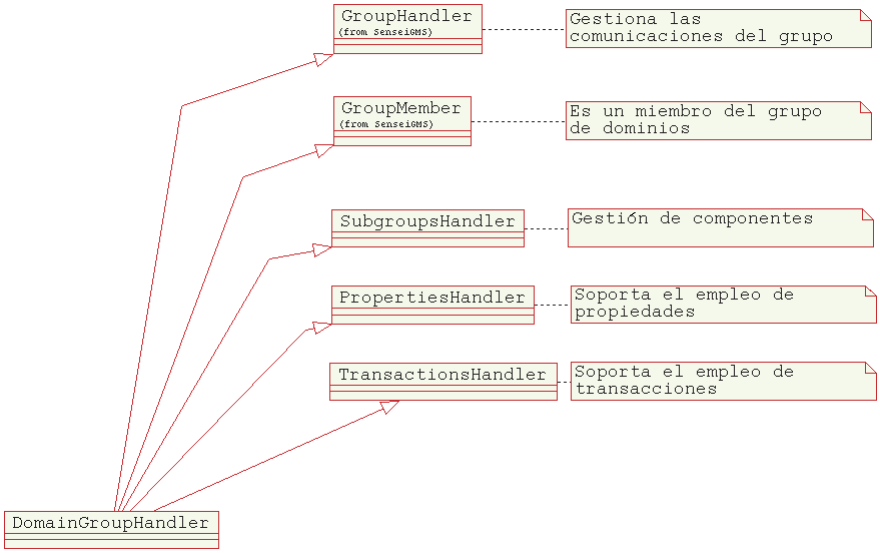

Figura 10.1. Jerarquía de herencia en *DomainGroupHandler*

La mayoría de las operaciones en el dominio son susceptibles de recibir una excepción *MemberStateException*, definida como:

```
enum MemberStateExceptionReason
{
        MemberNotJoined,
        MemberJoined,
        MemberWithoutState,
        MemberExcluded
};

exception MemberStateException
{
        MemberStateExceptionReason reason;
};
```

10.2. Transferencia de estado

La definición de la funcionalidad de transferencia de estado se incluye en `StateTransfer.idl`. El capítulo 7 desarrolló la interfaz de aplicación de un servicio tolerante a fallos CORBA con una transferencia de estado flexible; SenseiDomains no es una implementación de este servicio CORBA, por lo que la interfaz soportada es ligeramente diferente, con el objetivo de integrarse con el modelo genérico de grupos

que emplea SenseiGMS. En cualquier caso, la flexibilidad es la misma, así que la explicación de las siguientes definiciones no es exhaustiva, para no duplicar la información ya dada en el capítulo 7.

Esta definición incluye tipos directamente definidos en la especificación del servicio de tolerancia a fallos de CORBA, pero redefine algunos, bien para adaptarse a la interfaz de SenseiGMS, bien para añadir alguna funcionalidad. Estos tipos son:

```
abstract valuetype State {};

typedef GroupMemberId Location;
typedef sequence <Location> Locations;

interface Checkpointable : GroupMember
{
        State getState();
        void setState (in State s);
};

typedef string Name;
typedef string Value;
struct Property
{
        Name nam;
        Value val;
};
typedef sequence <Property> Properties;
```

- *State*: es el estado a transferir entre réplicas, definido en CORBA como un tipo *any*. SenseiDomains emplea un *valuetype* de tal forma que las aplicaciones puedan definir su propio tipo y no prescindan del soporte de tipado del lenguaje.
- *Location*: define la identidad de cada miembro del grupo, que viene dada en CORBA por una cadena de caracteres (`CosNaming::Name`). SenseiGMS define estas identidades con *GroupMemberId*, y empleamos este mismo tipo para *SenseiDomains*. De la misma forma, *Locations* define un conjunto de miembros.
- *Checkpointable*: define la interfaz que todos los miembros de un grupo con soporte de transferencia de estado deben implementar, correspondiente al soporte más simple en SenseiDomains. En CORBA, las operaciones emplean nombres separados, como *get_state* en lugar de *getState*, y la interfaz no extiende, obviamente, la definición SenseiGMS de *GroupMember*.
- *Property*: las propiedades asociadas al grupo, o a un miembro dado, se definen como pares {nombre, valor}. Bajo SenseiDomains, tanto el nombre como el valor de la propiedad se asocian a una cadena de caracteres para simplificar su tratamiento, mientras que en CORBA los valores son tipos *any*.

La transferencia de estado planteada en los capítulos anteriores soporta las transferencias en varias etapas. Como se explicó en la sección 7.2.1, la aplicación debe definir una entidad que contenga la información necesaria para sincronizar las diferentes etapas. SenseiDomains sólo emplea ese tipo para decidir cuándo una transferencia ha finalizado, predefiniendo, por lo tanto, una parte de esa entidad:

```
valuetype PhaseCoordination
{
        public boolean transferFinished;
};
```

La aplicación define su propia coordinación de fase, extendiendo generalmente este tipo previo. En cada etapa, debe asignar el valor correspondiente al campo *transferFinished* de la fase, donde un valor *true* marca el final de la transferencia.

Un componente incluido en el dominio define el soporte de transferencia de estado que precisa a partir de la interfaz que ese componente implementa. Las interfaces de transferencia de estado que puede implementar son:

- *StateHandler*: completo soporte de transferencias en varias etapas, admitiendo la interrupción de las transferencias por cambios de vistas.
- *BasicStateHandler*: soporte de transferencias en varias etapas, pero los cambios de vistas suponen la reiniciación completa de la transferencia.
- *ExtendedCheckpointable*: soporte básico, similar al definido en CORBA.
- *Checkpointable*: soporte básico, tal como lo define CORBA.
- Sin implementar ninguna de las anteriores interfaces: sin soporte de transferencia.

Estas interfaces se especifican a continuación, sin incluir *Checkpointable*, que ya se ha definido anteriormente. La única adición con respecto a la información del capítulo 7 es la interfaz *ExtendedCheckpointable*, que sólo se diferencia de *Checkpointable* en la operación *assumeState*. Esta operación permite que un componente pueda decidir cuál es su estado si no hay ninguna otra réplica que se lo pueda transferir.

```
interface ExtendedCheckpointable : Checkpointable
{
    void assumeState ();
};

interface BasicStateHandler : GroupMember
{
    void assumeState ();
    void startTransfer (in Locations joiningMembers, inout PhaseCoordination phase);
    State getState (inout PhaseCoordination phase);
```

```
    void setState (in State s, in PhaseCoordination phase);
    void stopTransfer (in Locations joiningMembers, in boolean transferFinished);
};

interface StateHandler : BasicStateHandler
{
    void syncTransfer (in Location coordinator, inout PhaseCoordination phase);
    void interruptTransfer (inout PhaseCoordination phase);
    void continueTransfer (in Locations joiningMembers,
                           inout PhaseCoordination coordinator,
                           in PhaseCoordination joining);
};
```

Cada componente de un mismo dominio puede implementar una interfaz diferente, no es preciso que todos los componentes requieran el mismo soporte de transferencia de estado. Además, esta interfaz soporta las transferencias concurrentes: por ejemplo, cuando una transferencia comienza, la operación *startTransfer* incluye el conjunto de miembros que recibirán el estado. Sin embargo, la implementación actual se limita a transferencias en serie: no es posible que un miembro transfiera el estado a varios miembros concurrentemente.

SenseiDomains permite la elección del coordinador de cada transferencia mediante las siguientes definiciones:

```
struct CoordinatorInformation
{
    Location statusMember;
    Locations currentCoordinations;
};
typedef sequence <CoordinatorInformation> CoordinatorInformationList;

interface CoordinatorElector
{
    Location getCoordinator (in Location loc, in CoordinatorInformationList info);
};
```

Cada componente del dominio puede requerir un soporte de transferencia de estado diferente pues, en caso contrario, se limitaría la reusabilidad de los componentes al poder emplear conjuntamente sólo aquellos que definieran el mismo soporte de transferencia. Sin embargo, no es posible que cada componente decida el coordinador para su transferencia: la aplicación debe registrar un único objeto que implemente la interfaz *CoordinatorElector*, con la que elige un coordinador cada vez que se precisa una transferencia. Para realizar esta elección, la aplicación recibe una lista con todos los miembros con estado y, para cada uno de esos miembros, las transferencias que están realizando en ese momento dado, de tal forma que pueda

realizar balanza de carga (esta posibilidad no se incluye en la interfaz dada en el capítulo 7).

No es necesario que la aplicación implemente esta interfaz: en caso contrario, la elección la realiza el dominio mismo, como se explica más adelante.

El último tipo definido especifica el comportamiento de la aplicación ante cambios de estados:

```
enum BehaviourOnViewChanges
{
        MembersOnTransferExcludedFromGroup,
        StatelessMembersDoNotBelongToGroup,
        StatelessMembersBelongToGroup
};
```

Durante la transferencia, SenseiDomains bloquea los mensajes a los miembros que la realizan, salvo que esos mensajes se definan como de no bloqueo. Ante un cambio de vista, *BehaviourOnViewChanges* define las acciones a tomar con esos mensajes:

- *MembersOnTransferExcludedFromGroup*: los miembros en la transferencia se consideran excluidos del grupo, por lo que no se ven afectados por los cambios de vista. Los mensajes quedan encolados y sólo se envían a la aplicación cuando la transferencia ha concluido. Permite implementar las transferencias más sencillas, pero no respeta completamente el modelo de sincronía virtual. Puesto que la transferencia no se interrumpe, no resulta lógico implementar la complicada interfaz *StateHandler* cuando se define este comportamiento.
- *StatelessMembersDoNotBelongToGroup*: considera que los miembros sin estado no pertenecen al grupo. Ante un cambio de vista, la transferencia se interrumpe y el miembro que la coordina recibe los mensajes que se hubieran bloqueado, que debe procesar antes de instalar la nueva vista. El miembro sin estado, sin embargo, no recibe esos mensajes, que son descartados.
- *StatelessMembersBelongToGroup*: todos los miembros se consideran incluidos en el grupo, siendo ésta la única opción que permite una solución que se adhiere completamente al modelo de sincronía virtual. Se comporta como en el anterior caso, pero el nuevo miembro recibe también los mensajes bloqueados antes de procesar la nueva vista.

Puesto que SenseiDomains soporta transacciones que, como mostró el capítulo anterior no se adhieren estrictamente al modelo de sincronía virtual, el comportamiento por defecto ante vistas es el que soporta una transferencia más simple: *MembersOnTransferExcludedFromGroup*.

Los protocolos que soportan la interfaz de transferencia de estado fueron definidos como *push* o *pull*, según el miembro sin estado recibiera automáticamente desde otro miembro el estado o tuviera que solicitarlo explícitamente. La interfaz

pública en ambos casos es la misma, luego la elección del protocolo afecta únicamente al rendimiento. Se demostró que, excepto en casos concretos y con soporte multipunto del medio empleado, el protocolo *push* ofrece mejores rendimientos, por lo que la versión actual de SenseiDomains implementa únicamente este protocolo.

10.3. Propiedades

Al estudiar la transferencia de estado se realizó una distinción entre el estado de un miembro y sus propiedades, y se definió la interfaz necesaria para soportar éstas. Esa interfaz estaba muy ligada al servicio de tolerancia a fallos CORBA, empleando entidades básicas de ese servicio, como el *PropertyManager* o las *GenericFactory* (sección 7.4). La solución aportada pretendía integrarse en la especificación hecha del servicio CORBA, introduciendo los mínimos cambios posibles para obtener la flexibilidad deseada. Esta integración no es precisa en SenseiDomains, e introducir las entidades necesarias supondría una complejidad adicional sin aportar ninguna ventaja.

Por esta razón, `Properties.idl` define las interfaces necesarias para soportar la funcionalidad de propiedades descrita en el capítulo 7, pero sin emplear las entidades CORBA. Este cambio permite, además de incrementar la funcionalidad, un manejo de las propiedades más flexible que en la descripción inicial. Posteriormente a esta primera descripción hemos introducido el concepto de dominio, agrupando componentes que podrían definir sus propias propiedades. Sin embargo, las propiedades deben emplearse a nivel de dominio, no a nivel de componente, y asociándose, por tanto, al servidor replicado y no a los componentes que lo constituyen.

La estructura *MemberProperties* asocia a un miembro dado un conjunto de propiedades:

```
struct MemberProperties
{
        GroupMemberId member;
        Properties props;
};
typedef sequence <MemberProperties> MemberPropertiesList;
```

De forma genérica, cuando una entidad debe usarse en listas se define un tipo específico, identificado para *MemberProperties* como *MemberPropertiesList*.

El capítulo 6 desarrolló los protocolos de bajo nivel que soportan la transferencia de estado y de propiedades entre miembros. El empleo de propiedades afecta a esos protocolos, que deben emplear más mensajes y con más información, incluso si un grupo específico no hace uso de esas propiedades. Para optimizar este

caso e impedir que el rendimiento de un grupo se resienta por una facilidad que no emplea, es posible especificar que el dominio no utiliza propiedades, en cuyo caso no se permite emplear las operaciones de manejo de esas propiedades. Si se emplean, se obtiene un error, asociado a la siguiente excepción:

```
exception PropertiesDisabledException
{
};
```

Todos los miembros del dominio deben utilizar la misma política de empleo de propiedades: o todos los miembros soportan propiedades o ninguno lo hace. Si se emplean propiedades, un miembro del dominio puede recibir notificaciones de cambios en las propiedades de otros miembros. Esta notificación no incluye los cambios, sólo su ámbito:

```
interface PropertiesListener
{
        void propertiesUpdated (in Location loc);
};
```

Las propiedades se modifican y se consultan empleando la interfaz *PropertiesHandler*:

```
interface PropertiesHandler
{
        void enableProperties() raises (MemberStateException);
        boolean arePropertiesEnabled();
        void setPropertiesListener(in PropertiesListener listener)
                raises (PropertiesDisabledException);
        MemberPropertiesList getAllProperties ()
                raises (MemberStateException, PropertiesDisabledException);
        MemberPropertiesList getPropertyForAllMembers (in Name nam)
                raises (MemberStateException, PropertiesDisabledException);
        Properties getMemberProperties (in Location loc)
                raises (MemberStateException, PropertiesDisabledException);
        Value getMemberProperty (in Location loc, in Name n)
                raises (MemberStateException, PropertiesDisabledException);
        Properties getProperties() raises (PropertiesDisabledException);
        Value getProperty(in Name n) raises (PropertiesDisabledException);
        void setProperties (in Properties props)
                raises (MemberStateException, PropertiesDisabledException);
        void addProperties (in Properties props)
                raises (MemberStateException, PropertiesDisabledException);
        void removeProperties (in Properties props)
                raises (MemberStateException, PropertiesDisabledException);
};
```

Estas operaciones se dividen en cuatro tipos. El primer tipo agrupa las operaciones que afectan al manejo global de propiedades:

- *enableProperties*: por defecto, el dominio no soporta propiedades, que deben habilitarse explícitamente, siendo imposible deshabilitarlas posteriormente. Además, esta operación debe efectuarse antes de incluir al miembro en un grupo, momento en que se inician los protocolos entre miembros, o se recibe una excepción *MemberStateException*.
- *arePropertiesEnabled*: método de consulta, permite conocer si las propiedades han sido habilitadas.
- *setPropertiesListener*: registra la instancia *PropertiesListener* que recibe las notificaciones de cambios de propiedades. Esta instancia puede modificarse en cualquier momento, salvo que no se hayan habilitado las propiedades. El argumento puede ser nulo para dejar de recibir notificaciones.

El resto de operaciones sólo pueden ser invocadas si las propiedades han sido habilitadas. El segundo grupo de operaciones permite modificar las propiedades de un miembro del grupo. Debe notarse que las propiedades de un miembro específico sólo pueden ser modificadas por ese miembro y siempre que no haya sido expulsado del grupo, en cuyo caso se lanzaría una excepción *MemberStateException*:

- *setProperties*: fija el conjunto de propiedades, eliminando toda propiedad previamente definida.
- *addProperties*: añade las propiedades dadas, sobrescribiendo las ya existentes con el mismo nombre.
- *removeProperties*: elimina las propiedades especificadas. No se considera un error eliminar una propiedad no definida.

Las operaciones que permiten observar las propiedades del miembro propio conforman el tercer grupo:

- *getProperties*: devuelve el conjunto de propiedades definidas para el propio miembro.
- *getProperty*: devuelve una propiedad específica del propio miembro; si no se ha definido tal propiedad, se devuelve una referencia nula.

El último grupo incluye a las operaciones que devuelven las propiedades de otros miembros del grupo. Como sólo es posible obtener información de otros miembros si se ha completado la transferencia de estado, es un error invocar a las siguientes operaciones sobre miembros sin estado:

- *getAllProperties*: devuelve las propiedades de todos los miembros.
- *getPropertyForAllMembers*: devuelve el valor de una propiedad específica en cada uno de los miembros del grupo.
- *getMemberProperties*: devuelve las propiedades de un miembro específico.

- *getMemberProperty*: devuelve una propiedad específica de un miembro determinado.

Gran parte de la funcionalidad proporcionada puede parecer superflua, pues los tres últimos grupos de operaciones podrían sintetizarse en dos operaciones: *setProperties, getProperties*. Como contrapartida, se obtiene una mayor facilidad de uso y un mejor rendimiento en casos específicos, como por ejemplo, al emplear *getPropertyForAllMembers*.

La interfaz *DomainGroupHandler* implementa la interfaz *PropertiesHandler*, es decir, todo el manejo de propiedades se realiza a través de la definición de dominio.

10.4. Componentes

En `SubgroupsHandler.idl` se define la gestión de componentes propuesta en el capítulo anterior, que emplea indistintamente los términos subgrupo y componente. Esta propuesta distinguía entre dos tipos de componentes, estáticos y dinámicos, según fuera su ciclo de vida. Ambos tipos son identificados en el dominio empleando una identidad, pero los primeros deben suministrar esa identidad mientras que es el dominio el que asigna la identidad de los segundos.

Las identidades de los componentes se definen como enteros en SenseiDomains, que incluye, además, una limitación adicional, al diferenciar los rangos de identidades que pueden emplearse para componentes estáticos y dinámicos. Se define, también, una identidad de componente universal, aplicable a todos los subgrupos del dominio:

```
typedef long SubgroupId;
typedef sequence<SubgroupId> SubgroupIdsList;
const long EVERY_SUBGROUP = 0;
const long MAX_STATIC_SUBGROUP_ID = 65535;
```

Al limitar los rangos de identidades, intentar registrar un componente con una identidad inválida o ya usada, provoca un error, para lo que define una excepción específica *SubgroupsHandlerException*, usada también al emplear componentes dinámicos erróneamente:

```
enum SubgroupsHandlerExceptionReason
{
        SubgroupIdAlreadyInUse,
        InvalidStaticSubgroupId,
        InvalidDynamicSubgroupId,
        DynamicBehaviourNotRegistered
};
exception SubgroupsHandlerException
{
```

```
        SubgroupsHandlerExceptionReason reason;
};
```

Si el dominio emplea componentes dinámicos, un miembro lo crea tras haberse incorporado al grupo y los demás miembros deben crear ese mismo componente en su espacio de memoria. Para que estos miembros conozcan el componente que deben crear, el primero debe enviar cierta información que le permita a la aplicación seleccionar el tipo de componente adecuado. Esta información se define a partir del tipo *DynamicSubgroupInfo*, definido simplemente como:

```
valuetype DynamicSubgroupInfo
{
};
```

La aplicación debe definir un tipo concreto, a partir del anterior, que incluya esa información, que es totalmente opaca para SenseiDomains. Como facilidad adicional, se incluye un tipo concreto para el caso en que la información pueda darse mediante una cadena de caracteres:

```
valuetype DynamicSubgroupInfoAsString : DynamicSubgroupInfo
{
        public string info;
};
```

El empleo de componentes dinámicos implica que la aplicación debe suministrar funcionalidad adicional que permita la creación de componentes en demanda, así como su eliminación. Esta funcionalidad se proporciona al crear y registrar un objeto que implemente la interfaz *DynamicSubgroupsUser*:

```
interface DynamicSubgroupsUser
{
    GroupMember acceptSubgroup (in SubgroupId id,
                                in DynamicSubgroupInfo info);
    GroupMember subgroupCreated(in GroupMemberId creator,
                                in SubgroupId id,
                                in DynamicSubgroupInfo info);
    void subgroupRemoved       (in GroupMemberId remover,
                                in SubgroupId id,
                                in DynamicSubgroupInfo info);
};
```

- *subgroupRemoved*: comunica a la aplicación que un miembro del grupo ha eliminado un componente dado. La razón de esa eliminación, si es necesaria, se incluye mediante un parámetro de tipo *DynamicSubgroupInfo*.
- *acceptSubgroup* y *subgroupCreated*: ambas operaciones se emplean para solicitar a la aplicación que cree un componente dado. La primera operación se invoca

cuando el miembro se une al grupo y recibe información de los componentes dinámicos ya existentes, mientras que la segunda se invoca cuando el miembro ya pertenece al grupo. La diferencia es que sólo en el primer caso el componente recibe una transferencia de estado.

La distinción entre *acceptSubgroup* y *subgroupCreated* se comprende mejor con un ejemplo. Un componente que implementa un árbol binario replicado se crea vacío, sin elementos, por lo que no es necesario que se transfiera información a las demás réplicas cuando se crea dinámicamente. En este caso, esas réplicas reciben una llamada *subgroupCreated* con la que crean un componente *vacío*. Si en algún momento se incluye un nuevo miembro en el grupo, recibe la solicitud *acceptSubgroup* para crear ese componente pero, en este caso, su estado vacío no es consistente con el grupo y debe recibir el estado desde otra réplica.

Esta aproximación es obviamente una generalización, pues hay componentes que pueden necesitar para su inicialización de una información dada. En ese caso, esta información puede incluirse en el tipo *DynamicSubgroupInfo*, siempre y cuando su tamaño no sea considerado *grande* pues, en caso contrario, el componente debería definir sus propios métodos de inicialización de estado. Por ejemplo, en un sistema replicado de ficheros, el componente *fichero* debería necesitar para su construcción el nombre asignado. Si se pretende crearlo directamente con contenido, el contenido de un fichero podría fácilmente exceder los megabytes, en cuyo caso es difícilmente defendible la inclusión de esa información en un objeto de tipo *DynamicSubgroupInfo* y su replicación en un único paso: el componente debe definir sus propias operaciones para soportar esta inicialización.

La gestión de componentes se realiza a través de la interfaz *SubgroupsHandler*:

```
interface SubgroupsHandler
{
        void setDynamicSubgroupsUser(in DynamicSubgroupsUser user)
                raises (MemberStateException);
        void registerSubgroup(in SubgroupId uniqueSubgroupId, in GroupMember subgroup)
                raises (MemberStateException, SubgroupsHandlerException);
        void castSubgroupCreation(in DynamicSubgroupInfo info)
                raises (MemberStateException, SubgroupsHandlerException);
        SubgroupId createSubgroup(in DynamicSubgroupInfo info, in GroupMember subgroup)
                raises (MemberStateException, SubgroupsHandlerException);
        boolean removeSubgroup(in SubgroupId id, in DynamicSubgroupInfo info)
                raises (MemberStateException, SubgroupsHandlerException);
        SubgroupIdsList getSubgroups();
        GroupMember getSubgroup(in SubgroupId id);
        SubgroupId getSubgroupId(in GroupMember subgroup);
};
```

- *setDynamicSubgroupsUser*: registra la instancia que recibe las notificaciones de creación y eliminación de componentes dinámicos. Si el miembro no define

ninguna instancia, no soportará componentes dinámicos. Tal como ocurre con las propiedades, todos o ninguno de los miembros del grupo deben soportar componentes dinámicos.

- *registerSubgroup*: registra un componente estático en el dominio, especificando su identidad. Los componentes estáticos deben registrarse antes de incorporar el dominio a su grupo.
- *castSubgroupCreation*: comunica al grupo que se va a crear un nuevo componente dinámico. En esta operación no se incluye el componente que se crea, por lo que deberá crearlo posteriormente en demanda, como las demás réplicas, al recibir la notificación *subgroupCreated*.
- *createSubgroup*: creación de un componente dinámico, como en el anterior caso. Sin embargo, el dominio obtiene ahora ese componente y, por tanto, la aplicación no recibirá posteriormente la notificación *subgroupCreated*. Esta operación se bloquea hasta que cada réplica del grupo recibe la notificación de creación del nuevo componente.
- *removeSubgroup*: elimina el componente dado. Sólo es posible eliminar los componentes dinámicos.
- *getSubgroups*: permite obtener la lista de componentes definidos, tanto estáticos como dinámicos.
- *getSubgroup*: devuelve el componente asignado a la identidad especificada, o una referencia nula, si no existe tal componente.
- *getSubgroupId*: devuelve la identidad del componente dado. Si el componente no pertenece al dominio, el valor devuelto es *EVERY_SUBGROUP*.

Los componentes no deben emplear ninguna nueva interfaz, implementan la interfaz *GroupMember* definida en SenseiGMS, pues la abstracción que da el dominio es la de agrupar componentes que de otra forma deberían definir su propio grupo independiente de réplicas, implementando igualmente la interfaz *GroupMember*. La interfaz *DomainGroupHandler* implementa la interfaz *SubgroupsHandler* y gestiona, por lo tanto, los componentes del dominio.

10.5. Mensajes

SenseiGMS es el sistema de comunicaciones empleado para enviar fiablemente mensajes entre réplicas. Sin embargo, los mensajes se definen a nivel de aplicación y su contenido es totalmente opaco para SenseiGMS.

SenseiDomains sí necesita enviar información adicional en cada mensaje como, por ejemplo, la identidad del componente al que afecta. Por esta razón, se define en `DomainMessage.idl` un mensaje con toda la información necesaria; este

mensaje debe ser extendido por la aplicación para que incluya su información específica, tal como ocurría bajo SenseiGMS. La definición base es:

```
typedef long MessageId;

valuetype DomainMessage : Message
{
        public SubgroupId subgroup;

        public boolean unqueuedOnST;
        public boolean unTransactionable;

        public boolean waitReception;
        public MessageId  id;
};
```

DomainMessage extiende la definición de *Message*, necesario para su envío por SenseiGMS. Define cinco atributos, de los que sólo tres son asignados por la aplicación.

- *subgroup*: identidad del componente al que afecta. Un valor *EVERY_SUBGROUP* se emplea para enviar el mensaje a todos los componentes.
- *unqueuedOnST*: si se define este atributo como *true*, el mensaje no es bloqueado durante las transferencias de estado. Un ejemplo de este tipo de mensajes son los definidos por el propio dominio para implementar la transferencia de estado. Por defecto se inicializa a *false*, todos los mensajes son bloqueados.
- *unTransactionable*: al definir este atributo como *true*, el mensaje no es bloqueado durante transacciones. El atributo previo bloquea los mensajes en recepción, en tanto este atributo los bloquea en envío. Por defecto se inicializa también a *false*.
- *waitReception* y *id*: usados internamente por el dominio.

El capítulo anterior mostró la restricción de que todo mensaje no transaccionable tampoco puede ser bloqueado durante transferencias. Por lo tanto, el valor del atributo *unTransactionable* es irrelevante si *unqueuedOnST* contiene el valor por defecto.

10.6. Control de concurrencia

La concurrencia sobre componentes replicados se controla mediante monitores y transacciones, ambos definidos en `Monitor.idl`.

La interfaz que debe soportar un monitor replicado se detalló en la sección 9.6. Su explicación incluyó la posibilidad de un empleo incorrecto de estos monitores

como, por ejemplo, liberarlos cuando no han sido previamente adquiridos, lo que supone la necesidad de una excepción específica, *MonitorException*:

```
exception MonitorException
{
};
```

El empleo de un monitor replicado puede provocar errores adicionales relacionados con el estado del miembro en el grupo, que se identifican con la excepción *MemberStateException*, ya definida. La interfaz *Monitor* se define como:

```
interface Monitor
{
        void lock() raises (MemberStateException);
        void unlock() raises (MonitorException, MemberStateException);
};
```

Donde la operación *lock* adquiere el monitor y *unlock* lo libera. La semántica concreta se explica detalladamente en el capítulo anterior; específicamente, los monitores deben ser reentrantes, es decir, un monitor puede ser readquirido repetidas veces.

La interfaz *Monitor* no define, sin embargo, un componente específico "monitor replicado", sino las operaciones que todo monitor debe soportar. La distinción es importante; en Java, por ejemplo, todos los objetos tienen asociado un monitor, por lo que puede considerarse que extienden una interfaz *Monitor* (con la semántica propia de monitores del lenguaje). De la misma forma, es posible definir un componente replicado cualquiera que incluya su propio acceso a recursos e implemente su interfaz en términos de esta interfaz *Monitor*. En el caso de una *lista*, por ejemplo, sería posible definirla como:

```
interface List : Monitor
{
 . . .
};
```

Este componente permitiría un acceso sincronizado sin necesidad de otro componente adicional que implementara el monitor replicado.

SenseiDomains define e implementa, además de esta interfaz, un componente replicado específico con esta funcionalidad, denominado *GroupMonitor*:

```
interface GroupMonitor : Monitor, ExtendedCheckpointable
{
};
```

Este componente define simplemente la misma interfaz *Monitor* y el soporte de transferencia de estado que precisa: una aplicación puede crear un componente

GroupMonitor para realizar el sincronizado sobre accesos concurrentes. Es el único componente replicado definido en implementado en SenseiDomains, por su necesidad para el empleo de transacciones.

Una aplicación puede realizar sus propias implementaciones de monitores, siempre y cuando extiendan la interfaz *Monitor*. Debe tenerse en cuenta que el mensaje asociado a la adquisición del monitor es no transaccionable, luego no se bloquea durante las transacciones. Por esta razón, es necesario considerar la posibilidad de reconstruir el estado del monitor a partir de mensajes de adquisición y liberación que pudieran llegar en orden incorrecto. La implementación en SenseiDomains soporta directamente este escenario.

Las transacciones son mecanismos necesarios para el empleo de componentes, pues permiten conservar la consistencia en el grupo aun cuando la réplica caiga sin haber completado una actualización que afecte a múltiples componentes. Sin embargo, puesto que su uso viola el modelo de sincronía virtual, una aplicación estrictamente adherida al modelo debe poder evitarlas. En este caso, el empleo de transacciones provoca una excepción:

```
exception TransactionException
{
};
```

Como detalle de implementación, en SenseiDomains no es posible especificar explícitamente cuándo las transacciones son permisibles o no. Este permiso se determina indirectamente a partir del valor dado a *BehaviourOnViewChanges*, el comportamiento ante cambios de vista. Este valor se emplea para definir el tratamiento de mensajes durante transferencias ante cambios de vistas, definiendo tres modos de los que sólo uno es totalmente compatible con el modelo de sincronía virtual. Esta compatibilidad implica una mayor complejidad en la implementación de los algoritmos de transferencia, por lo que su empleo implica una adherencia al modelo y la imposibilidad, por tanto, de usar transacciones. Si un dominio no se define con valor *MembersOnTransferExcludedFromGroup*, que es el valor por defecto, se considera que no soporta transacciones.

Las transacciones se manejan mediante la interfaz *TransactionHandler*:

```
interface TransactionsHandler
{
        void startTransaction(in Monitor locker)
                raises (MonitorException, TransactionException, MemberStateException);
        void endTransaction()
                raises (MonitorException, TransactionException, MemberStateException);
};
```

La transacción se realiza en torno al monitor especificado al iniciarla, no siendo necesario especificarlo de nuevo para completarla, aunque se aniden las

transacciones. Los cambios realizados en los componentes del dominio durante la transacción no se reflejan en las demás réplicas si la réplica que los realiza se cae o es expulsada del grupo sin haber completado todas sus transacciones en curso. La implementación de SenseiDomains implica que realizar la modificación de dos o más componentes en una transacción ofrece mejores resultados que si no se emplea esa transacción.

La interfaz *TransactionHandler* es extendida por *DomainGroupHandler*, por lo que las transacciones se manejan a través del concepto de dominio, como se propuso en el capítulo anterior.

10.7. Dominios

Como se adelantó en la introducción, el concepto de dominio lo define la interfaz *DomainGroupHandler*, especificada en `DomainGroupHandler.idl`.

Un dominio es un grupo que aglutina lógicamente una serie de componentes, que de otra forma hubieran constituido otros tantos grupos independientes, permitiendo así su gestión conjunta. Al implementar una funcionalidad adicional, la interacción con el dominio es más complicada que la que había con el grupo en SenseiGMS. La expulsión de un miembro de su grupo de comunicaciones se debe, en todos los casos, a problemas, reales o virtuales, en sus enlaces de comunicaciones. La expulsión de un miembro de un grupo gestionado mediante un dominio puede, sin embargo, deberse a otras razones, generalmente violaciones del modelo de dominios. Por este motivo se ha definido un tipo que enumera estas razones:

```
enum DomainExpulsionReason
{
        GMSLeaveEvent,
        WrongPropertyAllowance
        WrongStaticSubgroupsComposition,
        WrongCoordinatorElectionPolicy,
        WrongBehaviourMode,
        WrongDynamicPolicy,
        WrongSubgroupsTypes,
        SubgroupError
};
```

- *GMSLeaveEvent*: la expulsión no la ha originado el dominio, sino el mismo substrato de comunicaciones, sea por petición de la aplicación o por un error de las comunicaciones en el grupo.
- *WrongPropertyAllowance*: el miembro que se añade al grupo no presenta los mismos permisos de propiedades que los miembros ya existentes. Todos los miembros del grupo deben definir la misma política de empleo de propiedades.

- *WrongStaticSubgroupsComposition*: el miembro que se añade al grupo no ha definido los mismos componentes e identidades que los miembros ya existentes.
- *WrongCoordinatorElectionPolicy*: es también obligatorio que o todos o ninguno de los miembros del grupo sean capaces de elegir al coordinador de una transferencia.
- *WrongBehaviourMode*: el miembro que se incluye en el grupo ha definido un valor diferente para *BehaviourOnViewChanges* que los miembros ya existentes.
- *WrongDynamicPolicy*: no se ha cumplido el requisito asociado a componentes dinámicos, por el que todos los miembros del grupo deben emplear la misma aproximación en cuanto a permitirlos o no.
- *WrongSubgroupsTypes*: cada componente se instancia en cada miembro del dominio y, para que la transferencia de estado sea posible, es necesario que esos componentes implementen las mismas interfaces de soporte de transferencia en cada una de las réplicas. En caso contrario, se obtiene este motivo en la expulsión.
- *SubgroupError*: el dominio ha recibido una excepción al acceder a uno de los componentes, lo que supone su autoexclusión del grupo.

El dominio genera también eventos asociados al grupo de componentes en su conjunto, que la aplicación puede recibir si crea y registra una instancia de la interfaz *DomainGroupUser*:

```
interface DomainGroupUser
{
        void domainAccepted(in GroupMemberId id);
        void domainExpulsed(in DomainExpulsionReason reason);
        void stateObtained(in boolean assumed);
        void offendingSubgroup(in SubgroupId subgroup, in string reason);
};
```

- *domainAccepted*: evento de aceptación en el grupo, incluye la identidad de grupo asignada. Por otra parte, todos los componentes del dominio reciben el evento *memberAccepted* definido en *GroupMember*, que incluye igualmente la identidad de grupo, siendo obviamente la misma para todos los componentes del dominio.
- *domainExcluded*: expulsión del grupo, incluye la razón aducida por el dominio para tal expulsión. Los componentes reciben el evento *expulsedFromGroup*, aunque no el motivo de la expulsión.
- *stateObtained*: obtención de estado, evento obtenido una vez que el dominio se considera con estado, bien porque lo haya recibido o bien porque se le considere el miembro primario de un grupo sin miembros con estado. En este segundo caso, se considera que el dominio *asume* su estado.

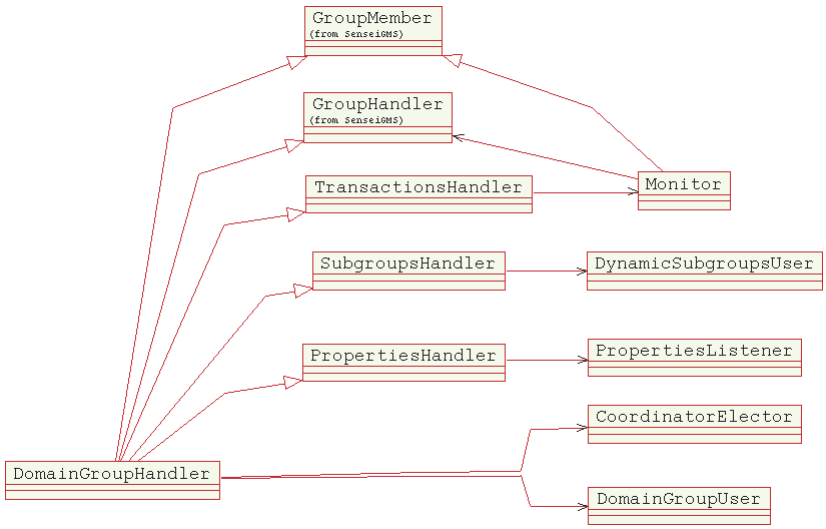

Figura 10.2. Principales dependencias de *DomainGroupHandler*

- *offendingSubgroup*: evento producido por un problema en un componente del dominio. A este evento le sigue eventualmente el de expulsión del grupo.

Finalmente, la interfaz *DomainGroupHandler* de define como:

```
interface DomainGroupHandler : GroupHandler, GroupMember,
        PropertiesHandler, SubgroupsHandler, TransactionsHandler
{
        void setBehaviourMode(in BehaviourOnViewChanges mode)
                raises (MemberStateException);
        void setCoordinatorElector(in CoordinatorElector elector)
                raises (MemberStateException);
        void setDomainGroupUser(in DomainGroupUser user)
                raises (MemberStateException);
        GroupMemberIdList getStatefulMembers() raises (MemberStateException);
        boolean syncCastDomainMessage(in DomainMessage message,
                                      in boolean normalProcessing)
                raises (MemberStateException);
        void syncCastDomainMessageProcessed()
                raises(MemberStateException, SyncCastDomainMessageProcessedException);
};
```

La figura 10.2 muestra las principales dependencias de esta definición:

- *DomainGroupHandler* se comporta como *GroupHandler*, que es la interfaz empleada en SenseiGMS para interaccionar con el grupo. Al emplear dominios,

toda la interacción con el grupo se realiza a partir de *DomainGroupHandler*, que implementa, por lo tanto, la funcionalidad de *GroupHandler* como, por ejemplo, la exclusión del grupo o el envío de mensajes. La interfaz *GroupMember*, que implementan los componentes, define el evento *memberAccepted*, por el que los componentes reciben la instancia *GroupHandler* cuando son aceptados en el grupo; al emplearlos en un dominio, la instancia que reciben en ese evento es la del dominio mismo.

- A su vez, *DomainGroupHandler* es el miembro de un grupo de réplicas, donde los otros miembros son instancias equivalentes del mismo dominio. Por esta razón debe implementar la interfaz *GroupMember*, según la define SenseiGMS.
- *DomainGroupHandler* es la entidad que gestiona las propiedades del dominio, implementando la interfaz *PropertiesHandler*. Admite el registro de una instancia *PropertiesListener* que recibe las notificaciones de cambios de propiedades en el grupo.
- *DomainGroupHandler* es el gestor de componentes, que implementa la interfaz *SubgroupsHandler*. Es capaz de soportar componentes dinámicos si se registra una instancia *DynamicSubgroupsUser* en el dominio.
- Por último, *DomainGroupHandler* extiende la interfaz *TransactionsHandler*, por lo que soporta el uso de transacciones sobre los componentes que gestiona.

Además del comportamiento definido por herencia, soporta la siguiente funcionalidad:

- *setBehaviourMode*; permite fijar el comportamiento de la transferencia de estado ante cambios de vistas. El comportamiento por defecto es el correspondiente a *MembersOnTransferExcludedFromGroup*; en caso de cambiarlo, el nuevo valor debe fijarse antes de incluir al dominio en el grupo.
- *setDomainGroupUser*: registra la instancia *DomainGroupUser* que recibe los eventos del dominio.
- *getStatefulMembers*: devuelve la lista de miembros del grupo con estado. La lista de todos los miembros, con o sin estado, es una información que ya se recibe a través de las vistas.
- *setCoordinatorElector*: registra la instancia *CoordinatorElector* empleada para la elección del coordinador. Si no se define ninguna instancia, el algoritmo por defecto en SenseiDomains distribuye las transferencias entre los miembros con estado, realizando una balanza de carga efectiva.

Una característica especial de SenseiDomains es que soporta una balanza de carga con pesos. Es posible asignar a cada miembro del dominio la propiedad "*weight*", cuyo valor determina su peso en esta balanza de carga. Por ejemplo, un miembro en el que esta propiedad tiene el valor "*2.0*" soporta el doble número de transferencias que otro con valor "*1.0*", mientras que uno con valor "*0.0*" sólo intervendrá en la transferencia si no hay otros miembros con estado con un valor

superior. De esta forma, es posible fijar el papel de cada miembro en la transferencia sin necesidad de crear un componente *CoordinatorElector*. Por ejemplo, un grupo que destina un miembro específico para las transferencias, de tal forma que los demás miembros no pierdan disponibilidad, puede asignar a ese miembro un peso grande, por ejemplo "*1000*", dejando el valor por defecto en los otros miembros.

Quedan dos operaciones de la interfaz *DomainGroupHandler* por explicar, relacionadas con el patrón de sincronización de mensajes que se desarrolló en la primera sección del capítulo anterior. Este patrón permite sincronizar la solicitud de un servicio en una réplica con el proceso asíncrono de obtener una respuesta tras comunicarse con las otras réplicas. Además, soporta dos variantes: la elaboración de la respuesta se hace en el mismo *thread* que realiza la comunicación con el grupo, o en el *thread* que recibe las comunicaciones del grupo. Si la solicitud de servicio requiere devolver algún valor al cliente, debe emplearse, preferiblemente, la primera variante; si no devuelve ningún valor, ambas variantes son adecuadas. Las operaciones son:

- *syncCastDomainMessage*: envía el mensaje al dominio y se bloquea este *thread* hasta que ese mismo mensaje se reciba. El parámetro *normalProcessing* indica dónde se procesa ese mensaje; si su valor es *true*, se procesa en la recepción del mensaje, y el bloqueo sólo termina cuando se ha completado su proceso. Si es *false*, el bloqueo finaliza al recibir el mensaje, lo que permite al proceso que inició la llamada *syncCastDomainMessage* despertarse y procesar el mensaje que envió. En este caso, el componente no observa la recepción del mensaje, y es necesario que, tras procesar el servicio, invoque la operación *syncCastDomainMessageProcessed*, o el dominio entero queda bloqueado.
- *syncCastDomainMessageProcessed:* completa la sincronización de un servicio que se procesa en envío.

Empleamos un ejemplo para clarificar su uso, basado en el que se utilizó en el capítulo anterior para exponer este patrón de sincronización. La operación *getNumber* devuelve un número secuencial, incrementado cada vez que se invoca sobre cualquier réplica del grupo, de tal forma que nunca se devuelvan dos números iguales. La operación *getNumber* se traduce en un mensaje *GetNumberMessage* que no incluye ninguna información adicional.

El siguiente listado muestra el código asociado a la primera alternativa:

```
public class NumberGenerator : . . .
{
        DomainGroupHandler domain;
        int lastGeneratedNumber;

        . . .

        public int getNumber()
```

```
        {
                domain.syncCastDomainMessage(new GetNumberMessage(), false);
                int result = ++lastGeneratedNumber;
                domain.syncCastDomainMessageProcessed();
                return lastGeneratedNumber;
        }

        public void processCastMessage(in GroupMemberId sender, in Message msg)
        {
                if (msg instanceof GetNumberMessage) {
                        ++lastGeneratedNumber;
                }
        }
}
```

- Al solicitarse un número, se envía el mensaje al grupo, comunicándole al dominio que no se emplea el procesado normal. En el procesado normal, tal como lo define SenseiGMS, todos los mensajes se reciben a través de *processCastMessage*. Si no se emplea procesado normal, el mensaje que una réplica envía no lo recibe.
- Cuando la operación de envío del mensaje vuelve, puede procesarse el servicio, incrementando simplemente el valor de la variable *lastGeneratedNumber* que almacena el último número dado.
- Tras procesarlo, y haber almacenado el valor a devolver al cliente, se comunica al dominio que el mensaje ha sido procesado. Con esta comunicación se indica al dominio que se puede procesar el siguiente mensaje del grupo, si lo hubiera.

El siguiente listado muestra el código asociado a la segunda alternativa.

```
public class NumberGenerator : . . .
{
        DomainGroupHandler domain;
        int lastGeneratedNumber;

        . . .

        public int getNumber()
        {
                domain.syncCastDomainMessage(new GetNumberMessage(), true);
                return lastGeneratedNumber;
        }

        public void processCastMessage(in GroupMemberId sender, in Message msg)
        {
                if (msg instanceof GetNumberMessage) {
```

```
                    ++lastGeneratedNumber;
            }
        }
}
```

- Al solicitarse un número en *getNumber*, se envía el mensaje al grupo, comunicándole al dominio que se emplea el procesado normal.
- El mensaje se recibe entonces a través del procedimiento normal, en *processCastMessage*. No hay diferencia entre la recepción de un mensaje propio o de otra réplica, se incrementa igualmente la variable *lastGeneratedNumber*.
- Cuando *processCastMessage* finaliza, se completa el bloqueo de *syncCastDomainMessage* en *getNumber*. El valor que devuelve al cliente es el que tiene en ese momento *lastGeneratedNumber*.

La segunda alternativa requiere menos código, aunque esa reducción es muy pequeña. Sin embargo, esta segunda solución es incorrecta, pues el manejo de *threads* impide saber si el desbloqueo sobre *syncCastDomainMessage* en *getNumber* se produce inmediatamente. Puede darse el caso de que se procese aún un segundo mensaje *GetNumberMessage* que modificaría el valor de *lastGeneratedNumber*, produciéndose un resultado erróneo. Es posible que la clase defina valores temporales para corregir este error pero, en ese caso, resulta más fácil emplear la primera alternativa.

Puesto que la invocación de *syncCastDomainMessageProcessed* implica que el dominio está bloqueado esperando a que la aplicación procese un mensaje, es un error invocarla en caso contrario, lo que se traduce en la excepción *SyncCastDomainMessageProcessedException*, definida simplemente como:

```
exception SyncCastDomainMessageProcessedException
{
};
```

Debe notarse que, puesto que *DomainGroupHandler* implementa la interfaz *GroupHandler*, también soporta la operación básica de envío de mensajes sin bloqueo: *`boolean castMessage(in Message msg)`*.

10.8. Implementación

La interfaz pública de SenseiDomains que se ha definido en las secciones anteriores, cubre las siguientes especificaciones:

- `DomainExceptions.idl`: excepciones genéricas del dominio.
- `DomainMessage.idl`: definición básica de mensaje de dominio.
- `Properties.idl`: soporte de propiedades.

- `DomainGroupHandler.idl`: definición de dominio.
- `Monitor.idl`: soporte de concurrencia, mediante réplicas y transacciones.
- `StateTransfer.idl`: soporte de transferencia de estado.
- `SubgroupsHandler.idl`: gestión de componentes.

Además, `InternalDomainMessages.idl` incluye todas las definiciones empleadas internamente por SenseiDomains pero que deben ser definidas a nivel de interfaz. Estas definiciones incluyen los mensajes internos y estructuras empleadas para la transferencia de estado.

SenseiDomains soporta todos los algoritmos de transferencia de estado desarrollados en los capítulos anteriores, a pesar de que el uso de transacciones es sólo compatible con el caso más simple. De la complejidad de estos algoritmos, sumada al amplio conjunto de facilidades asociadas al concepto de dominio, se deduce la dificultad de la implementación de SenseiDomains: tanto en complejidad como en tamaño (o el de las clases que lo implementan), es varias veces[8] superior a SenseiGMS.

El capítulo anterior definió la metodología que SenseiDomains implementa; sin embargo, hay detalles de esta implementación no directamente especificados en la metodología y aspectos de la metodología no recogidos en la implementación. Por ejemplo, la balanza de carga en la elección de coordinadores de transferencia empleando pesos definidos por la aplicación es una facilidad añadida.

Por otra parte, en SenseiDomains la posesión de un monitor no afecta a la transferencia de estado. Como se comprobó, una réplica que realiza una transacción no puede transferir su estado hasta concluir esa transacción. Aunque esta imposibilidad se podría eliminar complicando los algoritmos de tranferencia de estado, se trata de una propiedad conveniente: puesto que la transacción supone la adquisición de un monitor que puede bloquear a otras réplicas, es conveniente que termine esa transacción y ese bloqueo cuanto antes, lo que se facilita al no asignársele la tarea adicional de coordinar una transferencia de estado. El diseño de SenseiDomains permite inhibir la transferencia sobre miembros que realizan una transferencia y, aunque la implementación actual simplemente retrasa el comienzo de esa transferencia hasta que la transacción termina, es factible trasladar esa coordinación a otro miembro del grupo. Sin embargo, no hay ninguna alteración en el proceso de transferencia por el hecho de que un miembro posea un monitor, a pesar de que bloquea igualmente a las demás réplicas.

El motivo de esta diferencia es que, en SenseiDomains, un monitor es un componente más y el dominio no tiene conocimiento del estado de cada componente.

---

[8] La especificación JavaRMI de SenseiDomains define cinco veces más tipos que la de SenseiGMS. La implementación de los algoritmos de SenseiDomains emplea dos veces y media más líneas de código (excluyendo comentarios) que en SenseiGMS.

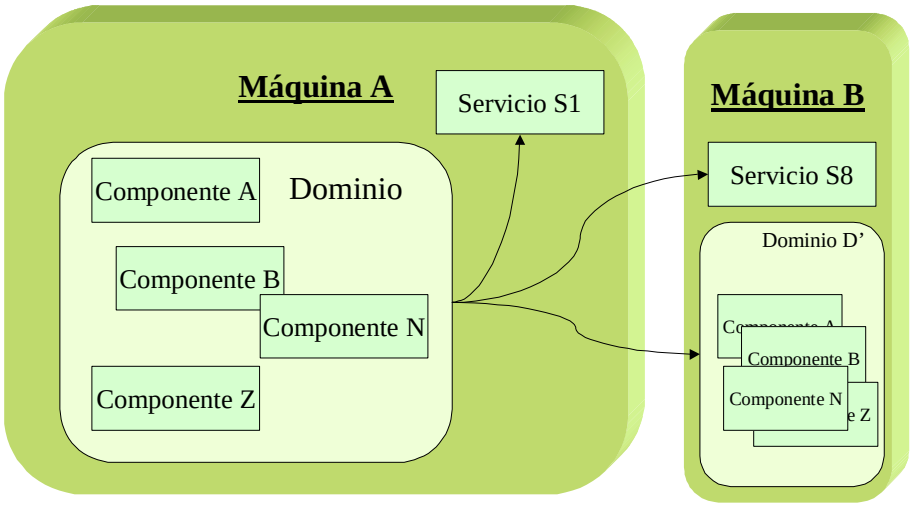

Figura 10.3. Modelo básico de dominio

Con el diseño actual, sería necesario que el dominio verificara su composición de componentes y obtuviera de los identificados como monitores su estado de bloqueo. O bien que la interfaz *DomainGroupHandler* incluyera funcionalidad que permitiera a los monitores informar de su estado pero, al ser una interfaz pública, no habría forma de comprobar la consistencia de los accesos a esa funcionalidad, comprometiendo la consistencia del dominio.

Una alternativa al diseño sería que la implementación de monitores se realizara en el dominio mismo, tal como ocurre con las transacciones, ofreciendo en su interfaz las operaciones *lock* y *unlock*. Esta aproximación emplearía cualquier componente como monitor, de una forma similar a Java, donde todos los objetos son monitores[9]. Sin embargo, la implementación implicaba una fuerte carga de trabajo sobre el dominio, al tener que mantener el estado de cada componente en su faceta de monitor: como un monitor emplea mensajes no transaccionables, su transferencia de estado es más costosa de realizar, coste que se transferiría al dominio, implicando un peor rendimiento general.

Por último, la base de esta metodología es la factorización de un servidor en componentes, para facilitar su replicación. Este servidor reside inicialmente en una máquina dada, aunque puede emplear otros componentes externos como, por ejemplo, un servicio de archivo, que residan en esa misma máquina o en otra cualquiera. Al dividir el servidor en componentes, el modelo del dominio asume, por

[9] Es discutible la relación entre objeto y monitor en Java, pues en puridad el objeto *contiene* un monitor, no *es* un monitor (problema de delegación frente a herencia).

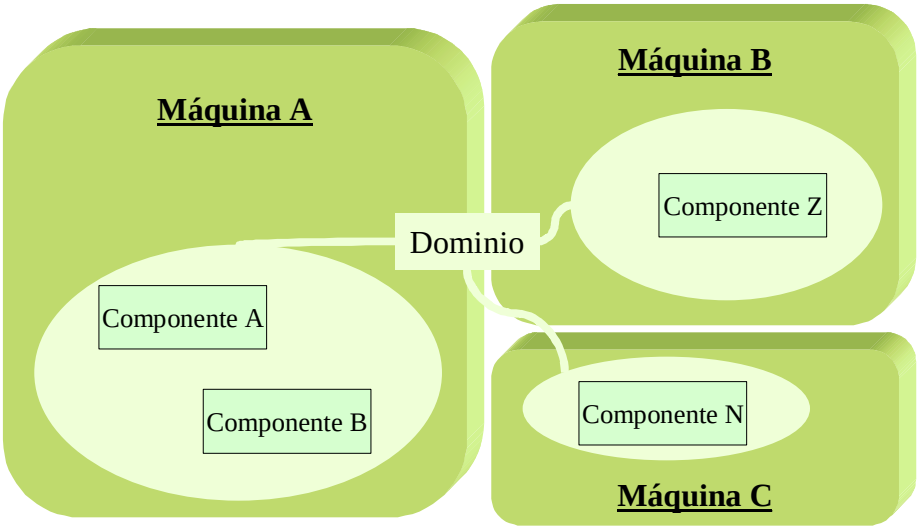

Figura 10.4. Modelo extendido de dominio

extensión, que los componentes residen en una única máquina, y la interacción entre el dominio y los componentes es local. La figura 10.3 muestra este modelo, donde un dominio y sus componentes residen en una máquina dada, pudiendo acceder a otros dominios y servicios o componentes en esa u otras máquinas.

A pesar de esta idea inicial de modelo, el dominio define su interacción con los componentes mediante una interfaz CORBA, por lo que no es preciso que su relación sea local: un dominio puede componerse de componentes esparcidos por diferentes máquinas, como muestra la figura 10.4. En la definición JavaRMI, la relación se define mediante interfaces remotos, por lo que dispone de la misma flexibilidad.

10.9. Conclusiones

SenseiDomains es un sistema que define y soporta un modelo de replicación basado en componentes, facilitando la gestión y empleo de componentes replicados para así poder diseñar servicios tolerantes a fallos a alto nivel. Estos servicios aún están basados en el modelo de sincronía virtual pero se les ocultan los detalles de bajo nivel asociados al modelo, como el manejo de vistas o el uso de primitivas de comunicaciones basadas en mensajes, que se relegan a la implementación de los componentes.

SenseiDomains no implementa directamente el modelo de sincronía virtual, precisa de un substrato de comunicaciones que lo soporte. El soporte que emplea es SenseiGMS, pero limitando sus dependencias a su interfaz, de tal modo que la

implementación pueda migrarse con facilidad a otros sistemas de comunicaciones en grupo, de acuerdo con la filosofía de SenseiGMS. Al igual que éste, SenseiDomains soporta CORBA y JavaRMI, con una interfaz totalmente equivalente entre ambos y una implementación común de los algoritmos.

La metodología que soporta SenseiDomains define una extensa funcionalidad, incluyendo una transferencia de estado flexible, el empleo de propiedades, control de concurrencia mediante monitores, soporte de transacciones, etc. La implementación se realiza de tal manera que no se afecte negativamente, en dificultad de uso o en rendimiento, a un dominio que no necesite una funcionalidad dada.

La factorización de servidores en componentes es especialmente ventajosa si se dispone de implementaciones ya preparadas de esos componentes. SenseiUMA es la parte de este proyecto que define una librería de contenedores replicados, como listas, pilas, etc. Estos componentes deben ser aún manejados conjuntamente, tarea específica del concepto de dominio, que evita, además, los problemas de acceso concurrente mediante transacciones y monitores, implementados en SenseiDomains de acuerdo a la metodología desarrollada.

Además, SenseiDomains implementa facilidades necesarias internamente por esos componentes, fundamentalmente la transferencia de estado y los patrones de sincronización de mensajes.

La metodología expuesta en el capítulo anterior también contempla la generación automática de componentes replicados. Las herramientas necesarias para esta generación son independientes del concepto de dominio y no están integradas en SenseiDomains.

Este capítulo no ha ejemplificado el uso de SenseiDomains. La razón es que el siguiente servicio desarrollado en Sensei, el servicio de localización de miembros, está construido de acuerdo a este modelo de componentes, empleando gran parte de las facilidades soportadas por SenseiDomains. Este servicio se denomina SenseiGMNS, y su funcionamiento e implementación se detallan en el siguiente capítulo, que es por lo tanto un ejemplo real de aplicación de SenseiDomains.