ISMED: Intelligent Semantic Middleware for Embedded Devices

Informe científico anual 2009

RESUMEN

Este informe contiene la memoria científica correspondiente al trabajo realizado durante 2009 para el proyecto de investigación ISMED. En tal proyecto cooperan la Universidad de Deusto (UD) y Mondragón Goi Eskola Politeknikoa (MGEP).

HISTORIAL DE CAMBIOS

Versión	Descripción	Autor	Fecha	Comentarios
V0.1	Creación de las partes relativas al modelado y coordinación semántica	Aitor Gómez	11/12/2009	
V0.2	Creación de las partes relativas al descubrimiento	Víctor Martín	14/12/2009	
V0.3	Creación de las partes relativas al aprendizaje	Asier Aztiria	14/12/2009	
V0.4	Creación de las partes relativas a la composición de servicios	Aitor Urbieta	14/12/2009	
V0.5	Integración de las distintas contribuciones de UD y MGEP	Aitor Gómez	18/12/2009	
V0.6	Revisión del documento	Diego Lz. de Ipiña	19/12/2009	
V0.7	Revisión final del documento	Aitor Gómez	21/12/2009	

TABLA DE CONTENIDOS

Resumen 3

Historial de cambios 4

Tabla de contenidos 5

1 Análisis Crítico del Estado del Arte 7

1.1 Módulo de modelado y coordinación semántica 7

1.1.1 Proyecto TripCom 7

2 Diseño del sistema 23

2.1 Módulo de modelado y coordinación semántica 23

2.1.1 Alternativas seleccionadas 23

2.1.2 Arquitectura propuesta 30

2.1.3 Interrelación capa de coordinación semántica y módulos de descubrimiento, composición, razonamiento y aprendizaje 33

2.1.4 Diagrama de clases 34

2.1.5 Ontología para ISMED 39

2.2 Módulo de aprendizaje 43

2.2.1 Capa de transformación 43

2.2.2 Capa de aprendizaje 45

2.2.3 Capa de aplicación 48

2.3 Módulo de composición de servicios 48

2.3.1 Propiedades del mecanismo de emparejamiento basado en *Conversation* 49

2.3.2 Integración de Conversaciones orientada a Conversaciones 52

2.4 Módulo de descubrimiento 61

2.4.1 Consideraciones 61

2.4.2 Primitivas a implementar 61

2.4.3 Diagrama de clases 64

2.5 Módulo de razonamiento 65

3 Implementación 66
3.1 Módulo de modelado y coordinación semántica 66
3.1.1 tsc++ modificado 66
3.1.2 tscME 67
3.1.3 Ejemplo de funcionamiento 69
3.2 Módulo de aprendizaje 71
3.2.1 Integración del módulo de aprendizaje con el módulo de coordinación semántica 74
3.3 Módulo de composición de servicios 76
3.3.1 Selección de candidatos 79
3.3.2 Generación de registro de conversaciones candidatas 82
3.3.3 Adaptación de la conversación solicitada 91
3.3.4 Composición de conversaciones 95
3.4 Módulo de descubrimiento 105
3.4.1 Consideraciones sobre la implementación 105
3.4.2 Estructura del módulo 105
3.4.3 Esquema de arquitectura y funcionamiento 107
3.4.4 Capturas de pantalla de la aplicación 113
3.5 Módulo de razonamiento 117
4 Conclusiones 122
5 Bibliografía 123

1 ANÁLISIS CRÍTICO DEL ESTADO DEL ARTE

1.1 Módulo de modelado y coordinación semántica

1.1.1 Proyecto TripCom

A continuación se muestra un análisis de los componentes del proyecto TripCom relacionados con la capacidad de realizar consultas SPARQL complejas, que más adelante (en el epígrafe 2.1.1.4.3) dará paso a una reflexión sobre la idoneidad de adaptar las distintas técnicas usadas en TripCom o no al proyecto de investigación ISMED.

1.1.1.1 Introducción

El proyecto TripCom es altamente modular y cada kernel está compuesto por los siguientes componentes [1]:

- Triple Store Adapter recibe las consultas y lee los datos almacenados en el repositorio pertinente.
- Security Manager asegura que las operaciones no violan la política de seguridad especificada.
- Mediation Manager ejerce de intermediario entre las tripletas o grafos entrantes y salientes.
- Metadata Manager es responsable de la implementación de la funcionalidad descrita por la ontología de Triple Space. Además, ejerce de razonador, validando la instancia de la ontología con su esquema y razona para encontrar información de los datos requeridos por los usuarios tales como estadísticas de acceso para tripletas, grafos, etc.
- Query Processor descompone las consultas en partes que pueden ser satisfechas por el repositorio local y en partes que deben ser enviadas a otros kernels.

- Transaction Manager gestiona las transacciones locales y participa en las transacciones distribuidas, en las que puede ser el coordinador de la misma si el kernel al que pertenece inició dicha transacción.

- Distribution Manager es el responsable de realizar un Triple Space distribuido, reenviando consultas y peticiones a kernels que probablemente sean capaces de satisfacerlas parcialmente y reenviando las peticiones de escritura a los kernels apropiados.

- Management API se usa por los administradores para inicializar las estructuras de datos y coordinar componentes del kernel. Es quien inicia todo el proceso del kernel.

- Triple Space API define el conjunto de operaciones que se ofrecen a los clientes del TSC.

- Web Service Invocation traduce las interacciones con los servicios web a operaciones del TS API.

- Web Service Discovery buscar descripciones de servicios web que concuerden con los objetivos de los solicitantes del servicio.

- Web Service Registry ofrece la funcionalidad necesaria para transformar descripciones de servicios web en grafos RDF y para almacenarlos en el Triple Space.

Además, existen las siguientes entidades externas al kernel.

- Triple Space Client permite crear clientes que acceden al Triple Space usando directamente su API.

- Service servicios webs definidos a un nivel superior del Triple Space que pueden ser usados por clientes de servicios web.

- Mediation Service provee la funcionalidad para mapear entre conceptos y ontologías, bien vía la API del servicio web bien vía el TS API,

- Orchestration Engine permite la composición de servicios.

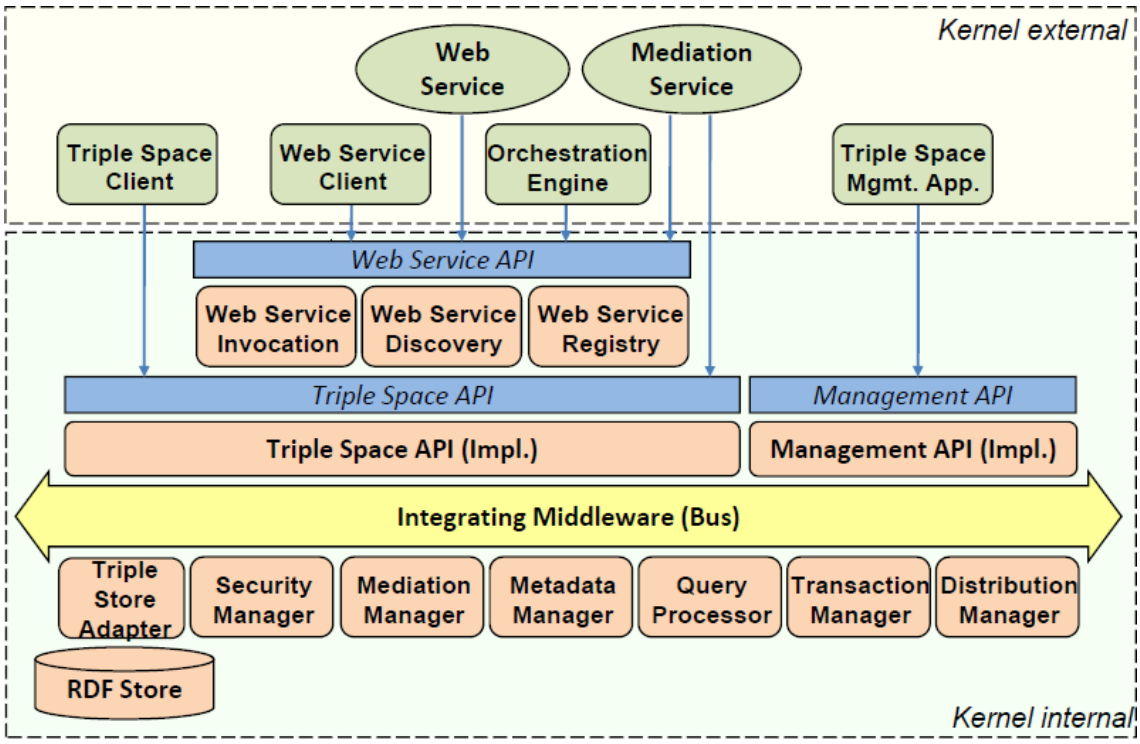


Ilustración 1. Esquema de los componentes de TripCom.

Para la problemática concreta que nos preocupa, se ha analizado el Triple Space API (dónde se especifican las operaciones que soporta cada Kernel), el Distribution Manager (encargado de localizar la información distribuida en los espacios y kernels) y el Query Preprocessor (por su capacidad de realizar consultas de una expresividad mayor).

En la **Ilustración 2**, se puede ver a un nivel alto de abstracción la relación entre los tres componentes anteriomente mencionados. El TS API provee a los clientes de una serie de operaciones. Para cumplir dichas operaciones, se apoya en el Distribution Manager, que se coordina con otros kernels remotos y con el repositorio semántico local y que por otro lado puede apoyarse en el Query Preprocessor para llevar a cabo consultas de una mayor complejidad.

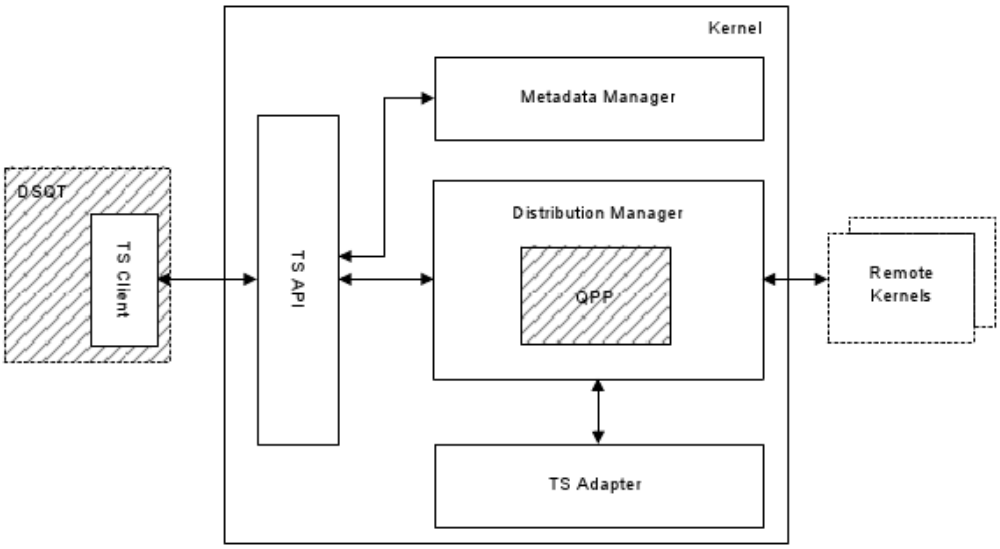


Ilustración 2. Los Interrelación entre los tres componentes analizados con mayor énfasis.

1.1.1.1.1 Definiciones

Para comprender plenamente lo que se explicará en las siguientes secciones, cabe aclarar cierta terminología usada en TripCom [1].

- Entidades lógicas:
    - Triple Space: es un conjunto de datos semánticos identificados por un mismo nombre. Puede estar distribuido a lo largo de un conjunto de nodos al que se puede acceder a través de las primitivas del API Triple Space.
    - Subespacio: es un subconjunto de datos en un Triple Space identificado por un contexto específico. Cumple con la definición de un Triple Space y puede tener sus propios subespacios.
    - Nodo: un nodo es una instancia de un kernel que consecuentemente ofrece un API a sus clientes. Una autoridad única gestiona cada nodo que es identificado por un identificador lógico único.
- Entidades físicas:
    - Kernel: es una implementación física única compuesta de todos los componentes que implementan el API Triple Space. Un kernel puede

desplegarse en una infraestructura física y puede ser direccionado usando una dirección de red física (bien como una única máquina física, bien como un único clúster de máquinas) como podría ser tsc://kernel1.morelab.deustotech.es:2588. Finalmente un kernel puede albergar un número arbitrario de nodos.

- Kernel local. Para un cliente determinado, un kernel local es aquel al que se encuentra conectado y el que usa para realizar operaciones sobre el Triple Space.

- Kernel remoto. Para un determinado cliente, es aquel kernel al que no se encuentra directamente conectado.

o Componente del kernel: es una parte del kernel que proporciona una funcionalidad específica y asume cierta responsabilidad en la arquitectura del kernel.

1.1.1.2 TS API

TripCom ofrece tres APIs distintas [2] sobre las que realizar operaciones de diferentes niveles de complejidad sobre el Triple Space. Estos tres niveles son: básica (core API), extendida (extended API) y otra más extendida aún (further Extended API).

1.1.1.2.1 Core API

Operación	Parámetros	Devuelve	Descripción
out	Triple, URL	-	Escribe atómicamente una tripleta en el espacio especificado.
rd	SingleTemplate, URL, timeout	Set<Triple>	Devuelve un resultado que coincida con el patrón especificado en el espacio identificado por su URL.
rd	SingleTemplate, timeout	Set<Triple>	Misma operación que la anterior, pero dado que no especifica la URL del espacio, debe seleccionar un kernel sobre el que tenga permiso de lectura y leer.

En este nivel inicial, las plantillas sobre los que se realizan las busquedas de tripletas, son patrones de tripletas individuales (tripletas que pueden contener variables).

1.1.1.2.2 Extended API

Operación	Parámetros	Devuelve	Descripción
out	Set<Triple>, URL	-	Escribe atómicamente un conjunto de tripletas.
rd	Template, URL, timeout	Set<Triple>	Es la misma operación descrita en la Core API pero usando plantillas más expresivas para la consulta.
rd	Template, timeout	Set<Triple>	Mismo caso que la anterior.
rdmultiple	Template, URL, timeout	Set<Set<Triple>>	Devuelve múltiples coincidencias para una plantilla dada.
subscribe	Template, Callback, URL	URI	Se crea suscripción de modo que se avise al Callback cuando se cumpla la plantilla dada en el espacio identificado por la URL.
unsubscribe	URI	-	Cancela una suscripción.

En este nivel se permite el uso de plantillas más expresivas (por ejemplo, consultas SPARQL).

1.1.1.2.3 Futher extended API

Operación	Parámetros	Devuelve	Descripción
in	Template, URL, timeout	Set<Triple>	Funciona como la rd, pero elimina los triples devueltos dentro del espacio definido.
Inmultiple	Template, URL, timeout	Set<Set<Triple>>	Elimina las múltiples respuestas que devuelve.
createTransaction	String [“local”, “shared”]	URI	Crea una transacción en un cliente (sólo él conoce la URI que la identifica), que puede compartirse con otros clientes.
getTransaction	URI	boolean	Se usa para unir transacciones de tipo shared con otros clientes.
beginTransaction	URI	boolean	Comienza la transacción y consecuentemente todas las interacciones entre agentes se realizarán “o todas o ninguna”.
commitTransaction	URI	boolean	Se ejecuta una operación de commit sobre la transacción dada.
rollbackTransaction	URI	boolean	Se ejecuta una operación de rollback sobre la transacción dada.

1.1.1.3 Distribution Manager

El conocido como Distribution Manager, en adelante DM, es el componente encargado de buscar kernels remotos, encaminar operaciones de otros kernels y recopilar información para dar respuestas a las operaciones pedidas por los clientes.

Como se ha comentado previamente, una URL identifica cada espacio. Sabiendo sobre qué espacio queremos realizar una operación, para llegar a él es tan sencillo como resolver mediante DNS a que máquina debemos dirigir dicha operación.

Sin embargo, cuando en una operación no se facilita la URL de un determinado espacio, debemos localizar quien es el encargado de la información concreta tratada en dicha operación para que la resuelva. En este caso el DM cobra especial importancia dado que es quién nos permitirá saber qué espacio buscamos.

Para dicha tarea, el DM se apoya en cuatro estrategias diferentes: tres que hacen uso de un conocimiento semántico local y que permiten un mayor rendimiento y otra que hace uso de índices sobre las tripletas manejadas.

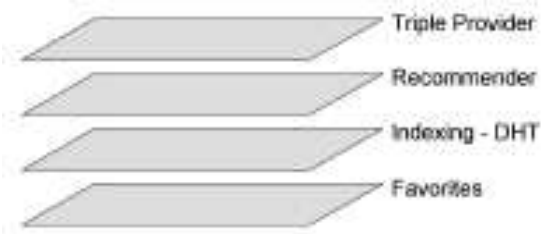


Ilustración 3. Esquema de las capas en las que se apoya el DM. para localizar el espacio que albega determinada información.

Las capas a las que se ha hecho referencia anteriormente son las siguientes [3]:

- **Triple Provider**. Esta capa sabe quién supo responder a una determinada consulta en el pasado (el llamado Triple Provider Kernel). Para ello crea atajos a los kernels siempre que recibe una respuesta y mantiene una tabla local con dicha información.

- **Recommender**. De forma semejante al Triple Provider almacena atajos a kernels que pese a que no respondieron directamente una consulta (no son TPKs), sí que supieron encaminarla hacía algún kernel que supo responderla

(TPK). Así, tras consultar a esta capa, se reenviará la consulta a un kernel que supo responderla en el pasado, bajo la premisa de que si supo responderla en el pasado sabrá responderla en el presente (dicho de otro modo, seguirá conociendo al TPK).

El uso de esta segunda capa, puede llevar al reenvío de operaciones a través de múltiples kernels, lo cual a su vez podría llevar a que se creen ciclos. Para evitar dicha situación, con cada consulta se reenvía una lista de kernels visitados en el proceso que será tenida en cuenta en cada DM.

Estas dos primeras capas almacenan sus atajos de forma similar (especificando en el tipo T o R dependiendo del caso del que se trate) en el mismo repositorio local, que tendrá una estructura semejante a la mostrada en la Tabla 1.

Graph Patterns	KernelURL Basic	Hits	Type	Timestamp
(?s, rdf:type, ?o)	tsc://fu-berlin.de	43	T	54656476476
(?s, rdf:type, dam:cc)	tsc://inf.fu-berlin.de	12	R	45747473457

Tabla 1. Repositorio de atajos para la capa Tiple Provider y Recommender.

- **Indexing – DHT**. Esta capa utiliza índices para localizar información relativa a los espacios. Estos índices se almacenan en una base de datos distribuida (UniStore), que usa el sistema de P2P PGrid como red subyacente.

Clave	Valor
Hash(s)	(s,p,o,SpaceURL)
Hash(p)	(s,p,o,SpaceURL)
Hash(o)	(s,p,o,SpaceURL)
Hash(SpaceURL)	(s,p,o,SpaceURL)

Tabla 2. Ejemplo de pares clave-valor insertadas por cada tripleta almacenada.

Para comprender su funcionamiento, bastaría con poner a modo de ejemplo un proceso de consulta, en el que partiendo de la misma, se realizarían los siguientes pasos:

1. Determinar las claves (basic triple patterns) en base a la consulta SPARQL.

2. Construir una consulta SQL para las claves obtenidas en el primer paso, y enviarlas al almacén de índices distribuido. El sistema P2P encaminará la consulta y la respuesta hacía y desde el kernel con la instancia del almacén de índices que contiene los datos relativos a la clave consultada.

3. Obtener la respuesta, con una serie de spaceURLs que contienen datos para las claves facilitadas.

4. Reenviar la consulta SPARQL a uno de los espacios.

- **Favorites**. Partiendo de los atajos almacenados en el repositorio de atajos local usado para las capas Triple Provider y Recommender, se crea otra tabla con los kernels más activos y que consecuentemente se cree que pueden responder más rápido a una consulta.

  Para ello se clasifican los kernels atendiendo a un factor que se obtiene multiplicando el número de atajos que un kernel ha creado por el número de kernels remotos que conoce (dato que se conoce preguntando directamente a los mismos). Dicho factor se actualiza siempre que se actualiza la tabla de atajos. En dicha tabla sólo se mantienen los kernels más favoritos, dado que serán los que se usarán para las operaciones. De esa forma se consiguen reducir sustancialmente el número de mensajes necesarios para actualizar cada tupla.

KernelURL	Shortcuts	Kernels	F. Factor
tsc://fu-berlin.de	20	30	600
tsc://inf.fu-berlin.de	32	10	320
tsc://mi.fu-berlin.de	15	20	300

Tabla 3. Tabla con los atajos a kernels favoritos.

1.1.1.3.1 Proceso de almacenamiento

Para comprender mejor el papel que juega el Distribution Manager al almacenar tripletas, por ejemplo al dar respuesta a una operación out, a continuación se detalla el proceso que seguiría [3].

1. Un cliente se conecta directamente a un kernel para acceder al Triple Space API.

2. El cliente realiza una consulta out especificando un espacio.

    a. El kernel local comprueba si es responsable de dicho espacio y por lo tanto es el encargado directo a la hora de resolver la operación.

        i. Si no es el responsable, reenvía la operación al kernel pertinente que será quien lleve a cabo la operación.

    b. Si el kernel local es responsable, se comprueba si la indexación está permitida

        i. En caso de ser así, envía una petición de almacenamiento para el valor (s,p,o,SpaceURL) al sistema de almacenamiento de índices.

    c. Almacena la tripleta en la capa de almacenamiento RDF mediante el TS Adapter.

Dado que en una operación out se índica en todo momento en que URL se desea almacenar una tripleta, queda patente que la distribución es controlada en todo momento por el usuario (pese a que una vez escrita no se permita la reubicación de la misma en otro kernel distinto)

Por ello, no es misión del DM distribuir dichas tripletas en sí (pese a que se contempló la posibilidad de usar distintas estrategias de distribución en [3]), sino localizarlas a través de los espacios en los que se encuentran distribuidas.

1.1.1.3.2 Proceso de consulta

Cada consulta se procesa en el Triple Store Adapter del kernel responsable de una determinada tripleta.

Existen dos tipos de consulta, con espacio y sin espacio. Las consultas con espacios se ejecutan de forma similar al proceso de almacenamiento, las consultas sin espacios deben hacer uso de las cuatro capas explicadas con anterioridad para localizar el kernel al que reenviar una consulta.

- Operaciones Read con Space URL. El DM comprueba si el espacio objetivo está en el kernel local o en un remoto.
    - Para reenviar la operación de lectura a un kernel remoto, el DM coge la URL del espacio y resuelve la dirección IP del kernel usando DNS. La comunicación entre kernels se realiza usando el protocolo SOAP y servicios web (DM usa un servicio web para conectarse a la implementación del API del kernel remoto).
    - Por cada lectura exitosa, crea un atajo para obtener más conocimiento de la red gradualmente.
- Operaciones READ sin Space URL.
    - Comprueba los espacios locales. Si la consulta se puede resolver localmente, devuelve los resultados al cliente.
    - Si no, comprueba la tabla de atajos y si existe una coincidencia exacta con reenvía la consulta a dicho kernel.
    - Si no encuentra un atajo, envía la consulta al almacén de índices distribuidos para obtener valores para dichos índices. El sistema P2P encaminará la consulta hacía el DM del kernel correspondiente y devolverá sus resultados.
    - Si no obtiene respuesta alguna del almacén, se reenviará la consulta a los kernels favoritos.
    - Pasado un periodo de *timeout*, si no consigue localizar un atajo, probará

con la capa por defecto que es la tabla de índices distribuida.

Finalmente, para realizar procesamiento de consultas distribuido (sólo para read sin espacio), hay que usar el subcomponente QPP que se explicará con mayor detenimiento en el siguiente epígrafe.

1.1.1.3.3 Sobre la inferencia de nuevas tripletas

Una vez realizado el proceso de consulta y encontrado kernels que potencialmente respondan una consulta, se reenvía la consulta a los mismos y la consulta se resolverá en su almacén RDF local usando su TS Adapter.

En este momento, las tripletas inferidas o virtuales (que no existen explícitamente en el almacén) se pueden añadir a la respuesta en cada kernel local usando el razonador del mismo.

Desafortunadamente, el proceso de razonamiento distribuido sobre el conocimiento global completo de la infraestructura Triple Space no fue objeto del proyecto TripCom.

1.1.1.4 Query PreProcessor

QPP es el componente de TripCom encargado de realizar el procesamiento de consultas distribuidas para operaciones *read sin espacio*. Este modulo es completamente independiente a la capa de razonamiento y almacenamiento, y se basa en SPARQL cost model, que es una técnica adaptada del campo de las bases de datos distribuidas.

QPP tiene a su vez dos partes principales [4]:

- **Query Optimizer**. Este componente permite reescribir consultas, estimar costes, ordenar joins y select, descomposición de consultas y construcción de respuestas.

- **Inconsistency reasoner**. Este detecta y evita inconsistencias como primer paso, de modo que asegura que los resultados obtenidos del procesador de consultas locales es consistente con la ontología antes de devolverlos al cliente.

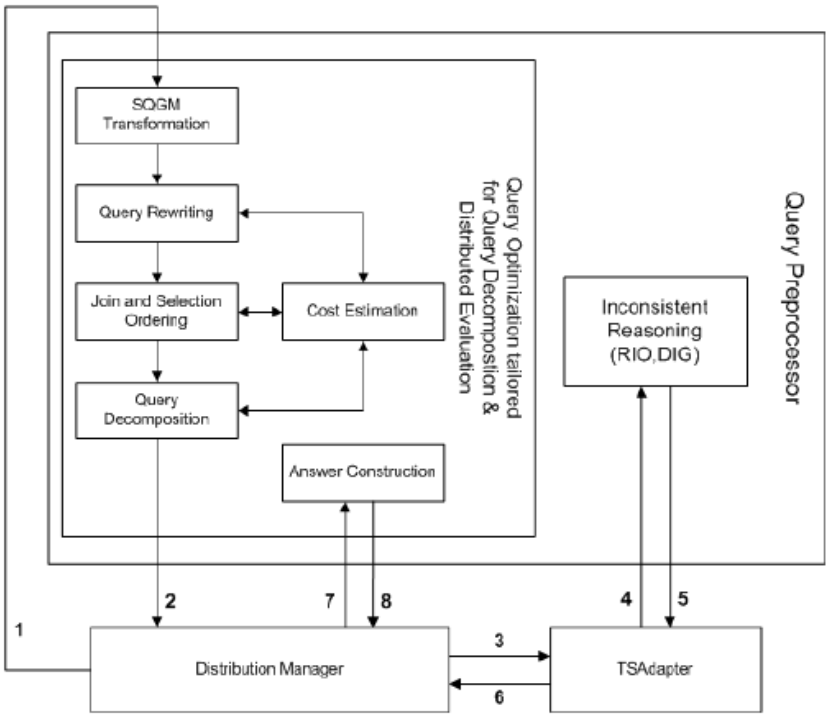


Ilustración 4. Esquema que muestra el modo en el que QPP se relaciona con otros componentes de TripCom.

En la Ilustración 4 se muestra un esquema del modo de proceder del QPP para dar respuesta a una consulta. En ella se pueden apreciar las interacciones de los distintos subcomponentes del QPP con otros componentes de TripCom.

1.1.1.4.1 Query Optimizer

Para comprender mejor el funcionamiento del QueryOptimizer del QPP, se describirá a continuación el proceso que sigue el componente analizado al responder a una consulta [5].

1. Parsing. Se obtiene un árbol de operadores de la consulta SPARQL mediante el motor ARQ.

2. Logical optimization. Se optimiza la consulta transformándola mediante reglas de escritura en una equivalente pero más eficiente en coste temporal.

3. Query decomposition. En base a los metadatos recibidos del DM sobre los kernels, se encuentra el camino más eficiente mediante el que resolver la consulta y en base a ello se descompone la consulta en subconsultas (más

concretamente en planes de consulta).

4. Optimización física y ejecución de la consulta.

    a. Se estima el coste físico en base a estadísticas (número de instrucciones de CPU, número de operaciones de E/S y retrasos introducidos por la red a la hora de acceder a datos remotos) para escoger el mejor plan a ejecutar

    b. Se reenvía a los DM los planes de consulta parciales

    c. El DM devuelve los resultados al QPP cuando le llegan

    d. El QPP construye respuestas parciales (realizando los bindings correspondientes)

    e. En base a los resultados obtenidos el QPP crea las siguientes consultas parciales a procesar y repite el proceso desde el punto b

    f. Eventualmente, tras un número finito de repeticiones, se obtendrá un conjunto de soluciones mapeadas

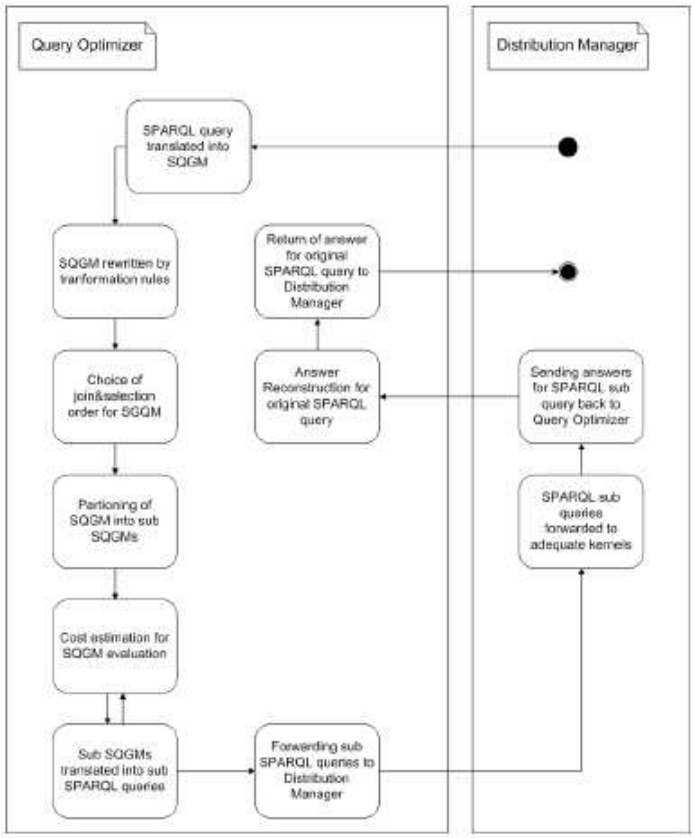


Ilustración 5. Diagrama de actividad para el Query Optimizer y el DM.

1.1.1.4.2 Inconsistency reasoner

Este subcomponente del QPP provee de datos al TS Adapter de modo que le permita soportar comprobación de la inconsistencia en las respuestas que obtiene del repositorio semántico local [4].

En caso de encontrar inconsistencias, se obtiene un subconjunto del resultado que no contiene ninguna inconsistencia con las ontologías del Triple Space.

Para realizar esa comprobación, se usa un razonador especializado conocido como RIO. El RIO encuentra y resuelve cualquier inconsistencia devolviendo un subconjunto consistente de la ontología.

De esa forma, se puede realizar de nuevo la consulta, tras haber eliminado la inconsistencia de la ontología, sobre una teoría lógica consistente.

2 DISEÑO DEL SISTEMA

2.1 Módulo de modelado y coordinación semántica

2.1.1 Alternativas seleccionadas

A continuación se describirán las alternativas utilizadas para realizar el middleware de coordinación semántica de ISMED. Esta lista no es más que una actualización de la sección con idéntico nombre de la memoria científica del 2008 [6], por lo que se incidirá especialmente en aquellas alternativas que haya sido necesario alterar, indicando los motivos que nos llevaron a ello.

2.1.1.1 Plataformas empotradas elegidas

A continuación se enumeran las plataformas empotradas para las que se va a desarrollar el middleware ISMED.

Además, cabe destacar que tal y como se adelantó en la memoria del año pasado, se ha concluido que es inviable implementar en ellas todos los módulos funcionales que componen ISMED por falta de recursos computacionales.

Pese a ello, se ha creado para todas ellas una versión del módulo de coordinación semántica, de forma que puedan proveer de los datos que dichas plataformas capturen al espacio Triple Space al que se encuentren unidas.

Para aquellas plataformas que no disponen capacidad de computo suficiente (MicaZ, SunSpot y XBee), se ha optado por utilizar una arquitectura híbrida en la que elementos de la red que pueden ofrecer de mayor capacidad computacional asistan al middleware desplegado en dichas siguientes plataformas.

- Sunspot [7]
- MicaZ [8]
- Gumstix [9]

- XBee [10]. De las cuatro alternativas enumeradas, esta es la única que no se incluyó en la memoria del año pasado. Los sensores XBee utilizados se comunican de forma transparente para el desarrollador con un Gateway [11]. Dicho Gateway tiene conectividad IP (mediante wireless o Ethernet) y en él se pueden cargar programas hechos en Python que manejarán los datos aportados por los sensores. Este sensor se ha escogido por la sencillez y transparencia con la que permite capturar datos del entorno, permitiendo al desarrollador centrarse en el Gateway y no preocuparse por el protocolo de comunicación entre este y los sensores.

2.1.1.2 Dispositivos móviles

Aparte de las plataformas de sistemas empotrados mencionadas en el apartado anterior también se hará uso de teléfonos móviles. Éstos son dispositivos de uso común que formarán parte del ecosistema de dispositivos cooperativos que pretende coordinar el middleware de ISMED.

- Nokia Series 60 soportando CLCD 1.1 Java ME. Concretamente se hará uso de dispositivos Nokia N96.

2.1.1.3 Librerías y tecnologías

A continuación se enumeran varias librerías que se han usado para implementar el módulo de coordinación de ISMED.

- Jxta

  Se ha escogido por tratarse de un protocolo bien definido con versiones para distintas plataformas, que permite gestionar redes P2P con un alto nivel de abstracción.

  En concreto, se usarán las siguientes versiones:

  - JXTA-JXSE 2.5, se trata de la implementación principal de Jxta realizada en Java, que será usada en aquellas plataformas empotradas que lo permitan y en las máquinas que sirvan de intermediarias para dispositivos demasiado simples.

- JXTA-JXME 2.5, es la versión para dispositivos móviles que usen Java ME, tanto CDC como CLDC. La principal diferencia con la versión que en la memoria del 2008 se indicó que se iba a usar, es que la versión 2.5 permite a los dispositivos móviles CLDC no depender de Proxys que los comuniquen con el resto de nodos Jxta. Si bien es cierto que debido a las limitaciones impuestas por los dispositivos móviles no se soportan todos los protocolos de la versión JXSE, sí que implementa los suficientes como para que un nodo móvil pueda comunicarse con el resto de nodos de la red mediante un Rendezvous.

- Repositorios semánticos

  Se usa Sesame (con Owlim, opcionalmente), para aquellas plataformas empotradas que permitan usar Java y en las máquinas que sirvan de intermediarias para dispositivos demasiado simples.

- Jena

  Se usa para filtrar resultados de consultas complejas y para, apoyándose en su procesador de SPARQL ARQ, descomponer dichas consultas SPARQ en templates que todos los nodos pueden interpretar.

- Microjena

  Se usa para expresar la información semántica en los dispositivos con Java ME.

- Motores de inferencia limitados

  Se analizará la posibilidad de crearemos nuestro propio motor de inferencia, considerando la adopción de otro hipotético motor de inferencia más completo para entornos empotrados y programado en Java, siempre que se disponga de una versión del mismo antes del fin del proyecto.

2.1.1.4 Análisis de la aplicabilidad de las técnicas de TripCom a ISMED

Tal y como se explicó en la memoria del 2008 [6], el módulo de coordinación semántica estará basado principalmente en la librería tsc++ (a la que se le aplicarán los cambios explicados más adelante). Sin embargo, a lo largo del año 2009 finalizó el proyecto TripCom (ver epígrafe 1.1.1), y con ello quedo patente la capacidad del mismo para resolver algunos de los problemas detectados en tsc++.

Así, a continuación se identificarán aquellas características o técnicas de TripCom que se pueden aplicar a ISMED para paliar alguna de las limitaciones detectadas en tsc++.

2.1.1.4.1 Diferencias entre TripCom e ISMED

En primer lugar, cabe comentar que existen tres **estrategias** principales a la hora de almacenar y devolver datos en sistemas **distribuidos**: centralización, flooding e índices distribuidos [3].

Así, mientras TripCom usa PGrid que está basado en **índices distribuidos**, una aproximación completamente descentralizada dónde los peers manejan referencias a otros peers y mapean la información que estos les ofrecen, ISMED está basado en tsc++ que a su vez usa Jxta que se ayuda del **flooding**, dónde los peers no almacenan información sobre los datos almacenados por otros peers y se propaga a todos los nodos participantes.

Concretamente tsc++ utiliza negative broadcast, dónde las operaciones de escritura son locales y las consultas son distribuidas (en positive broadcast las consultas son locales y la escritura se propaga a todos los nodos).

En segundo lugar, cabe destacar que si bien ISMED está pensado para ser ejecutado en un **entorno local** dónde los retrasos por comunicación entre dispositivos serán más reducidos y homogéneos entre sí, la filosofía de TripCom busca crear un “internet para máquinas” en el que dichas máquinas se ubican en redes WAN y que sea altamente escalable.

Bajo este entorno de redes locales en el que ISMED trabajará, no es necesario tener diferentes estrategias para optimizar la búsqueda del peer que debe resolver una

respuesta concreta mediante índices distribuidos y permite utilizar un enfoque mucho más sencillo como el flooding en su lugar.

En tercer lugar, los kernels del proyecto TripCom están compuestos de componentes que no tienen porque ejecutarse necesariamente en la misma máquina y que aíslan el proyecto en partes de alta complejidad. No todas las partes tienen que ser implementadas para ofrecer una API que siga el paradigma Triple Space, y en ese sentido el hecho de que ISMED deba ejecutarse en **dispositivos de una capacidad de cómputo reducida** hace interesante reducir al máximo el tamaño del kernel a utilizar dejando en el sólo lo estrictamente necesario para tener un sistema funcional.

En ese aspecto, cabe destacar también que el enfoque de TripCom aboga por unos **clientes** que son **ajenos** a toda la distribución y manejo de los datos, y sólo interactúan con un servidor (uno de los kernels) consultando y recibiendo las consultas a través del API que este les proporciona. Mientras tanto todo kernel de tsc++ es en sí un cliente con capacidad de hacer consultas sobre un espacio. En tsc++ el nodo que sigue el paradigma Triple Space es en sí quien pregunta al espacio, no un simple intermediario de los clientes con TripleSpace como en TripCom.

Finalmente, cabe destacar que los dispositivos pueden entrar y salir con relativa facilidad de la red de sensores que se formará en ISMED, mientras que en TripCom pese a que los kernels pueden incorporarse a la red y salir de ella, este hecho es más complejo y por lo tanto no es un proceso pensado para ser realizado con asiduidad.

2.1.1.4.2 Limitaciones de ISMED

Se han detectado las siguientes limitaciones a la hora de adaptar el paradigma del Triple Space Computing al proyecto ISMED:

- **Razonamiento distribuido**. El poder razonar sobre el conjunto de la información existente en un espacio completo es una problemática de alto nivel de complejidad que no ha sido abordado tampoco por TripCom.

- **Razonamiento en dispositivos móviles**. Punto que se tratará en el epígrafe correspondiente al módulo de razonamiento.

- **Limitación en la expresividad de las consultas**. Actualmente ISMED trabaja

con consultas realizadas con patrones de tripletas individuales. Al igual que TripCom con su Core API y su Extended API, se ha planteado paliar esta solución en ISMED con una serie de medidas:

- Aumentar la expresividad del lenguaje de consultas mediante la inclusión de primitivas para la comparación de cifras y arreglos de caracteres.

- Permitir usar consultas SPARQL en aquellos kernels que tengan capacidad para responderlas localmente.

- **Consultas distribuidas**. Se puede tratar de adaptar los métodos usados en el QPP de TripCom.

2.1.1.4.3 Adaptación de funcionalidades de TripCom

Siendo conscientes de las diferencias que existen entre TripCom y el middleware a desarrollar en ISMED, cabe hacer un análisis de las funcionalidades que ofrece TripCom pero no ISMED con su versión de tsc++, para pensar cuál de ellas sería conveniente implementar en ISMED y de qué forma se podría adaptar.

Los métodos para identificar kernels descritos en él para el DM de TripCom no son aplicables a ISMED por la diferencia en su estrategia de distribución vista anteriormente (ISMED no puede dirigir una consulta a un kernel concreto).

Sin embargo, alguna de las estrategias usadas en el QPP de TripCom podría ser usada para realizar consultas distribuidas. Así, de las dos partes principales del QPP [4]:

- **Inconsistency reasoner**. En una primera iteración no se planea implementar este módulo, dado que no soluciona ninguna de las limitaciones descritas anteriormente.

- **Query Optimizer**. De este componente no se implementará la estimación de coste y la reescritura de consultas (por ser más difíciles de implementar en un sistema de flooding). La descomposición de consultas y construcción de respuestas, sin embargo, parecen buenas estrategias para permitir las

consultas distribuidas.

Así, partiendo de una consulta SPARQL compleja, se puede descomponer en templates de tipo sujeto-predicado-objeto, que cualquier nodo de tsc++ sabrá responder.

El problema, una vez más, es que no cualquier dispositivo tendrá la capacidad suficiente como para realizar esa descomposición (que en TripCom se hace haciendo uso del procesador SPARQL de Jena ARQ). Para salvar esa limitación, y guiándonos por el diseño de TripCom, nos apoyaremos en dos niveles de APIs distintos.

El API más básico de todos ellos será el de TripleSpace básico. El extendido, tendrá aquellas primitivas que se apoyen en la descomposición de consultas.

Otra solución un poco más compleja, sería la de implementar ambos APIs en todos los dispositivos, pero hacer que aquellos que no sean capaces de descomponer las consultas las redirigiesen a aquellos dispositivos que si pudiesen hacerlo. De esta forma, estos últimos nodos capaces, serían los responsables de obtener las respuestas y reenviarlas al nodo limitado que redirigió la pregunta inicialmente.

En un primer momento, se descartará la segunda solución por la complejidad que implica, adoptando la de los dos APIs.

2.1.1.5 Primitivas a implementar

El API básica de ISMED implementa las siguientes primitivas:

- *Write*: almacena información en el repositorio RDF.
- *Read*: realiza una búsqueda en base a un patrón en todos los grafos devolviendo los grafos que contienen tripletas que cumplen dicho patrón.
- *Take*: es análogo a read, pero tras leer las tripletas, las elimina.
- *Query*: realiza una búsqueda en base a un patrón en todos los grafos devolviendo solamente las tripletas que lo cumplen.
- *Subscribe*: para solucionar el problema de la diseminación de información se

suelen usar el paradigma publish/subscribe. Así los suscriptores pueden expresar su interés por determinados datos suscribiéndose.

- *Advertise*: permiten publicar cierta información cumpliendo con el paradigma previamente mencionado.

Mientras que el API extendida sólo consta de una primitiva a día de hoy.

- QueryMultiple: permite realizar consultas SPARQL sobre el espacio. Para ello descompone en templates más básicos la consulta y la vuelve a aplicar en global sobre el conjunto de las respuestas recibidas (para comprender mejor su funcionamiento, se recomienda acudir al epígrafe 3.5).

2.1.2 Arquitectura propuesta

En base a lo indicado se pretenden usar las siguientes librerías en los siguientes tipos de hardware:

- Móviles CLDC: J2ME, microjena (manejo de tripletas y filtrado básico), jxme-proxyless y repositorio normal (RecordStore) o ficheros para persistir dichas tripletas.

- Sunspot, los sunspots se comunicarán con su base mediante sockets de forma que la aplicación desplegada haga de nodo tsc++.

- MicaZ, deberán comunicarse con una estación base conectada a un ordenador que será el verdadero nodo de la red Jxta y tendrá un repositorio semántico.

- Gumstix: JXSE y Sesame.

    - Podría incluso hacer de proxy para los dispositivos móviles que usen la versión de JXME con proxy.

A modo de resumen sobre lo comentado previamente, se muestra un esquema de la arquitectura propuesta y las tecnologías a aplicar en cada uno de los dispositivos seleccionados. Esta arquitectura es el punto de partida para el diseño del resto de módulos que conforman el middleware ISMED: descubrimiento, composición, razonamiento y aprendizaje. Obsérvese que dadas las limitadas capacidades

computacionales de las plataformas empotradas consideradas, al menos en una primera fase, tendremos que apoyarnos en diversas pasarelas que permitan la intermediación entre el mundo TCP/IP usado por los protocolos de coordinación adoptados en ISMED y los mundos de redes puras ad-hoc Zigbee usados por plataformas como Gumstix o SunSPOT. En esas pasarelas con mayor capacidad computacional podrá razonarse, aplicar algoritmos de aprendizaje e incluso en muchas ocasiones guardar en modo persistente el estado semántico generado por los distintos dispositivos que conforman el ecosistema de ISMED.

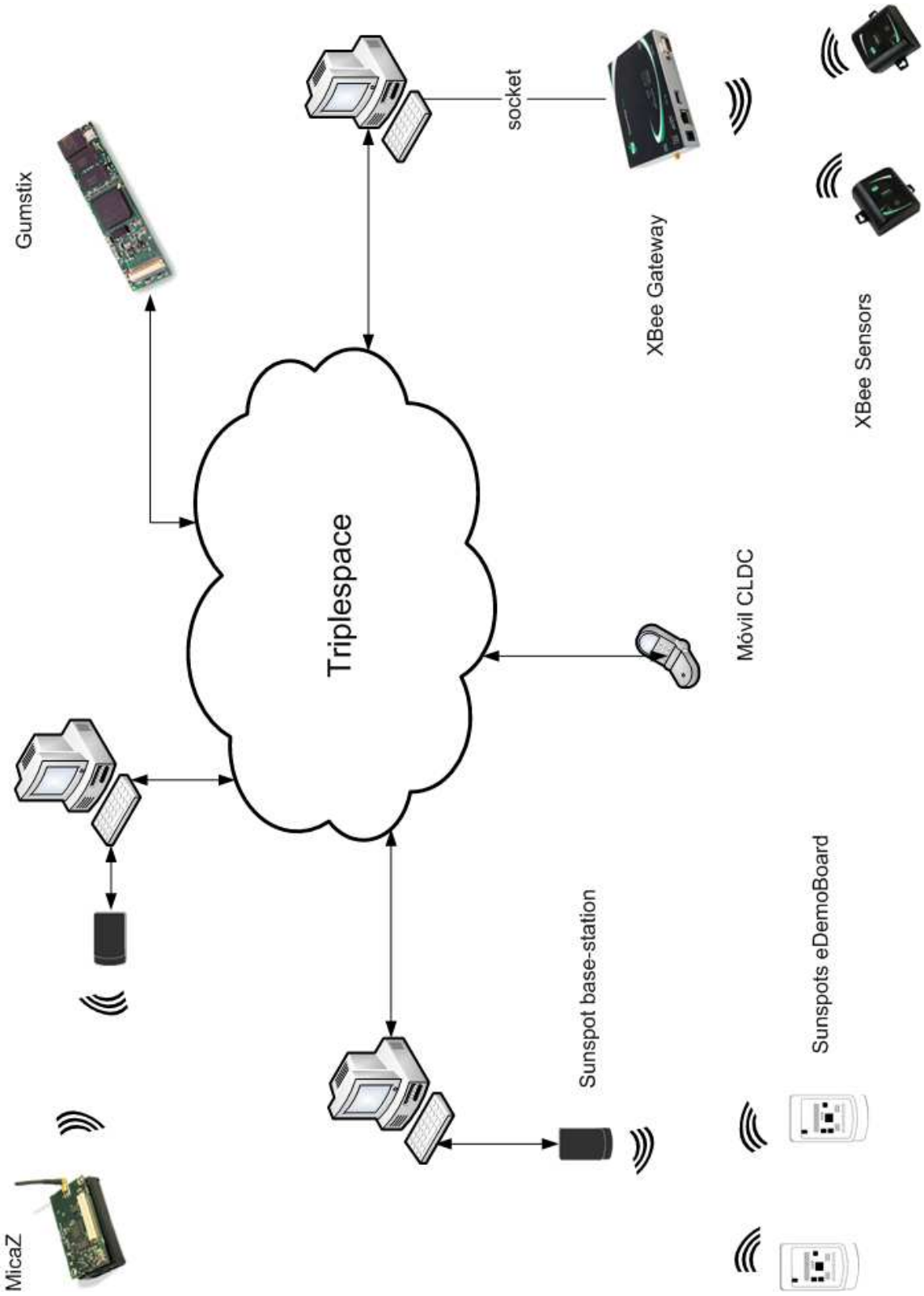


Ilustración 6. Esquema de la arquitectura propuesto.

2.1.3 Interrelación capa de coordinación semántica y módulos de descubrimiento, composición, razonamiento y aprendizaje

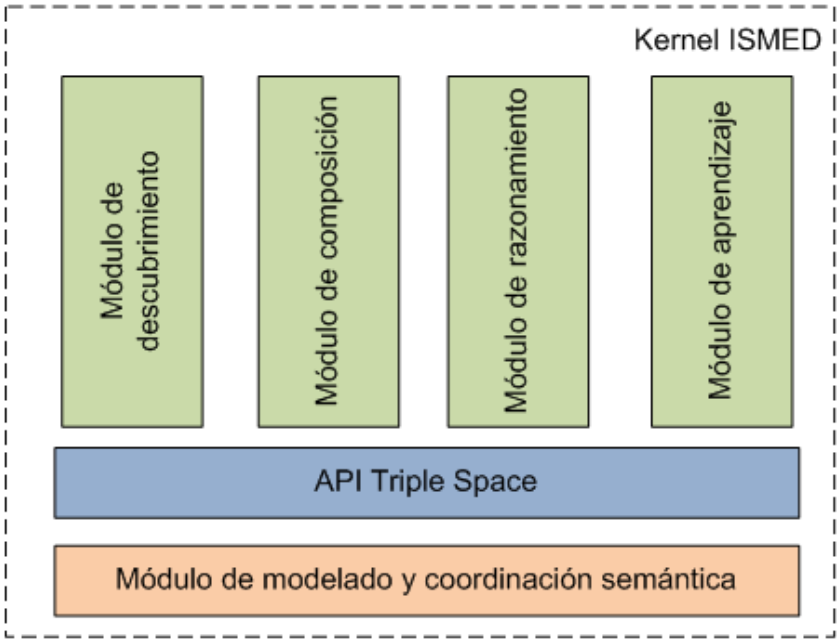


Ilustración 7. Esquema de las capas que compondrán el kernel ISMED.

Los módulos de descubrimiento, composición, razonamiento y aprendizaje van a mediar con el módulo de modelado y coordinación semántica a través del API siguiendo el paradigma Triple Space Computing descrito en detalle en la memoria del 2008.

Tal API permitirá al módulo de descubrimiento formular consultas por medio de las primitivas que provee sobre servicios en el entorno. Asimismo, el módulo de descubrimiento podrá también registrarse para recibir notificaciones sobre la aparición de nuevos servicios.

El módulo de composición de servicios se apoyará en el módulo de descubrimiento para encontrar candidatos de servicios que puedan emparejarse. Además, se apoyará en módulo de razonamiento para poder inferir nuevas asociaciones entre servicios.

El módulo de razonamiento permitirá a los dispositivos equipados con middleware ISMED tener capacidad de reactividad, para de modo proactivo adaptarse a las nuevas circunstancias del entorno que les rodea.

El módulo de aprendizaje tendrá por misión identificar nuevas reglas de

comportamiento del entorno y la optimización de las reglas actualmente existentes, que nutran el módulo de razonamiento.

La cooperación y entendimiento entre los diferentes módulos de ISMED requiere **la existencia de una ontología común entre ellos**, es decir, un vocabulario que permita describir los servicios exportados por los diferentes dispositivos del ecosistema de ISMED. Además, estos servicios habrán de estar anotados para que los módulos de razonamiento y aprendizaje sean capaces de interrogarlos para obtener conocimiento sobre el que razonar y aprender.

En conclusión, la API de Triple Spaces que proporciona el módulo de modelado y coordinación semántica constituye la infraestructura de consulta, coordinación y comunicación que puede usarse en la implementación de los demás módulos.

A este respecto, cabe destacar que debido a la limitada capacidad de cómputo de algunos de los dispositivos que se usarán, sólo la capa del módulo de modelado y coordinación semántica será implementada por completo en todos ellos (requiriendo en muchos casos además distintas versiones de dicho middleware).

De esta forma, si bien el aprendizaje, descubrimiento o la composición de servicios se realizarán en un nodo concreto de la red, los datos en los que se apoyarán se encontrarán distribuidos en todo momento.

2.1.4 Diagrama de clases

En este epígrafe se tratará de aportar una visión más detallada del diseño mediante sendos diagramas de clases tanto en nodos de dispositivos móviles como en nodos que se ejecuten en computadoras normales o Gumstix.

Para estos últimos, tal y como se ha descrito anteriormente, se ha utilizado de base la versión 1.2 de la librería tsc++ [12], sobre la que se han realizado numerosos ajustes. El más importante de ellos, es que respetando su forma original, se le ha añadido la capacidad de comunicarse mediante pipes de Jxta, para poder comunicarse con los nodos en dispositivos móviles. Además, se ha añadido la primitiva queryMultiple que no existía en tsc++.

Todos estos cambios se han realizado de forma que el código original quedase lo

menos afectado posible, a fin de poder integrar fácilmente los cambios que se vayan produciendo en la librería original.

2.1.4.1 Tsc++ modificado

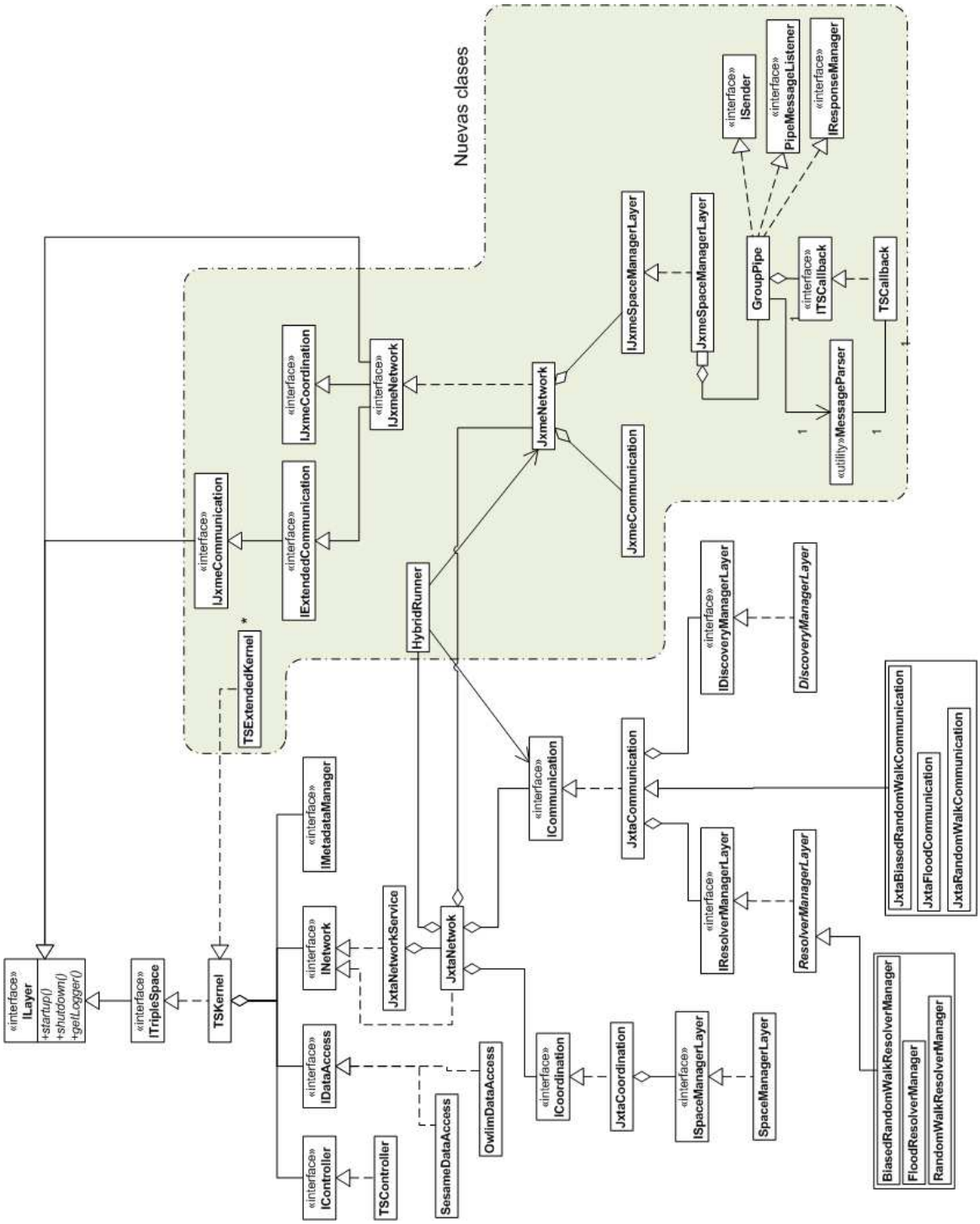


Ilustración 8. Diagrama de clases de la tsc++ modificado.

En el diagrama se muestran todas las clases añadidas para dar soporte a la comunicación con móviles en la parte derecha sombreada. Así, se puede apreciar el limitado impacto que tiene en las clases originales de tsc++ y que, como se comentaba anteriormente, es uno de los objetivos primordiales.

Para instanciar un kernel de tsc++, basta con crear una clase *TSKernel*, que esencialmente estará compuesta por una clase de acceso a datos *IDataAccess* (que tratará con los repositorios semánticos Sesame u Owlim) y otra que manejará los aspectos relativos a la red (*INetwork*).

A ese nivel aparece la primera particularidad: el API extendida de la que se hablaba antes, tendrá las primitivas del API básica, más otras primitivas, y se podrá hacer uso de ella instanciando *TSExtendedKernel*, que tendrá una estructura interna similar a *TSKernel* y que ha sido obviada por claridad.

También se puede apreciar que todo *INetwork* se compone de un elementos de coordinación (*ICoordination*) y otro de comunicación (*ICommunication*). La primera aglutina la respuesta a las primitivas que realizan operaciones sobre los espacios (*create*, *join* y *leave*) y la segunda el resto de primitivas de Triple Space.

La comunicación entre nodos tsc++ se realiza mediante el intercambio de advertisements personalizados. Este tipo de comunicación no está plenamente soportada por la versión para Java ME de Jxta, por lo que se ha tenido que usar un enfoque híbrido en el que con otros nodos tsc++ se utilice ese tipo de comunicación, y con los nodos móviles se comunique mediante Pipes.

Es por ello que se ha creado la clase denominada *HibridRunner* y toda la estructura de clases de la zona derecha del diagrama. Así, en *JxtaNetwork* se llamará a *HibridRunner* que será la clase encargada de llamar en sendos Threads a las primitivas mediante el método original de tsc++ (haciendo uso de *ICoordination* e *ICommunication*) y mediante pipes (haciendo uso de *IJxmeCommunication* e *IJxmeSpaceManagerLayer*).

JxtaCommunication, que implementa *ICommunication*, tiene otras clases entre las que cabe destacar *IResolverManagerLayer* e *IDiscoveryManagerLayer*. En la primera se reciben llamadas a las primitivas *query*, *read* o *take* de otros nodos, mientras que en el segundo se tratan las relacionadas con las suscripciones (*advertisement*, *subscribe*).

El enfoque seguido en la otra rama (*JxmeCommunication*), es pedir a la capa de coordinación (*IJxmeSpaceManagerLayer*) el pipe asociado a un espacio y escribir sobre él mensajes. Así, *JxmeSpaceManagerLayer* se encarga de crear, eliminar y devolver referencias a pipes, cuyos métodos de escritura y lectura se implementan en la clase *GroupPipe*.

2.1.4.2 tscME

A continuación se detalla el esquema de clases diseñado para la versión del middleware que se desplegará en dispositivos móviles, a la que hemos llamado **tscME** y para la que nos hemos inspirado en la estructura utilizada en tsc++.

Como se puede apreciar en un primer vistazo, al igual que en tsc++, la interfaz *ITripleSpace* define las primitivas del TripleSpace definidas con anterioridad. *TSKernel* implementa dicha interfaz, y realiza las primitivas delegando en *IDataAccess* y *INetwork*, que a su vez delega en *ICoordination* e *ICommunication*.

Como particularidad, cabe destacar que *IDataAccess* en este caso tiene dos estrategias básicas de tratamiento de datos: en memoria o usando el RecordStore de Java ME CLDC. El almacenamiento de tripletas en ficheros también se contempló, pero resultaba excesivamente lenta y al desplegarla en dispositivos móviles requería firmar la aplicación (dado que de otro modo saltaban mensajes emergentes constantemente y no permitían el normal funcionamiento de la aplicación que usase la librería tscME).

JxmeCoordination usa una clase *JxmePeerBase*, que le proporciona la configuración básica necesaria para crear un nodo en Jxme. Además, tiene dos mapas en los que guarda espacios creados y unidos.

Los espacios se representan mediante *JxmeSpace*, que es quien escribe y lee en el pipe asociado a su espacio. Además, tiene una *IncomingList*, en las que asocia las posibles respuestas a los mensajes enviados con anterioridad.

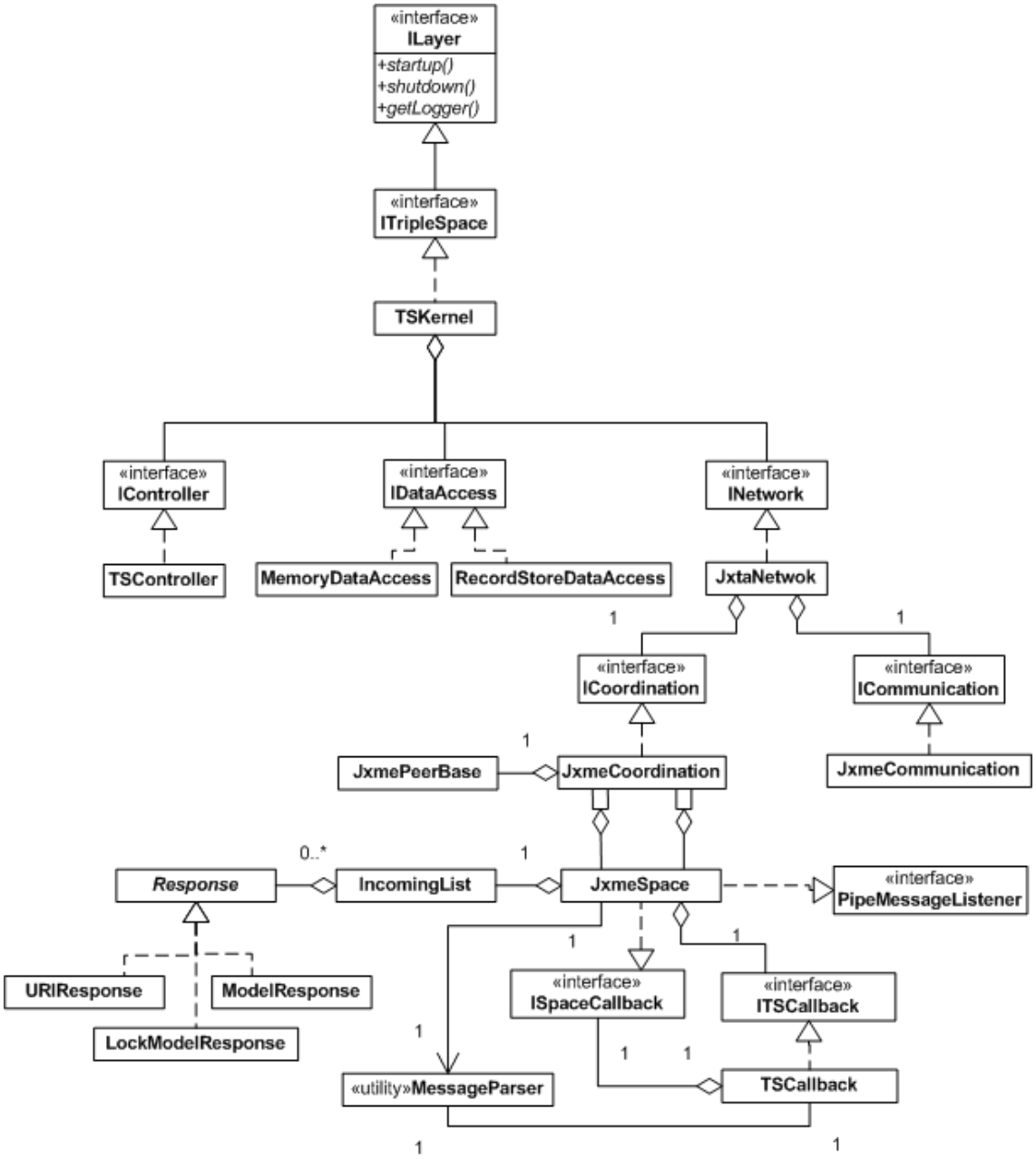


Ilustración 9 . Diagrama de clases de la librería tscME.

2.1.5 Ontología para ISMED

Esta sección describe brevemente la ontología desarrollada para ISMED que será la *lingua franca* que permita el entendimiento entre los objetivos dispares pero complementarios de los módulos de descubrimiento, composición, razonamiento y aprendizaje que conforman ISMED.

Para realizar esta ontología, se consultaron numerosas ontologías para realizarlo, entre las que cabe destacar: la de la tésis de Ramón Hervás [13] y [19].

En un primer nivel, encontraríamos el entorno (*Environment*), que estará compuesto por un conjunto de áreas (*Area*), que a su vez podrán ser edificios (*Building*), o alas (*Wing*) o plantas (*Floor*) del mismo. Un área puede a tener a su vez lugares (Place), que pueden ser cuartos (*Room*) o pasillos (*Corridor*).

Otro elemento importante, serían la entidades (*Entity*), que en nuestra ontología podrían ser los propios usuarios del sistema (*User*) o los dispositivos que forman la red (*Device*).

Las medidas (*Measure*), a su vez, estarán asociadas al dispositivo que las toma y podrían ser de tipo: temperatura (*TemperatureMeasure*), el encendido o apagado de un switch (*SwitchMeasure*) o de un led (*LEDMeasure*), ruido (*NoiseMeasure*), luz (LightMeasure) o humedad (HumidityMeasure).

Con la definición de ítems P2P (*P2PItem*), se ha pretendido aportar información relativa a la propia red P2P que se genera al usar el módulo de coordinación semántica. En ella hay espacios (*Spaces*) que son las subdivisiones lógicas con las que trabaja Triple Space, *Rendezvous* que es un elemento necesario en una arquitectura Jxta para que los mensajes se propaguen a través de la red o permitir descubrimiento de nuevos nodos en la red sin necesidad de usar multicast, y nodos (*Peer*). Los nodos pueden ser de tipo tsc++ (*tscppPeer*) en caso de que usen la versión modificada de esta librería o *tscME* en el caso de aquellos que usen la librería para móviles realizada en ISMED.

También existe la propiedad *de* recursos multimedia (*ResourceMultimedia*), que son clases con las que se podrá especificar algún metadato que sirva para representar una entidad y enriquecer la posible interfaz en la que se muestren dichas entidades. Estos datos podrán ser de tipo texto (*ResourceText*), imagen (*ResourceImage*), audio

(*ResourceAudio*) y vídeo (*ResourceVideo*).

Finalmente, también se pueden apreciar otras clases que representan tipos más básicos:

- *Color*
- *Uncertainty*. Con la que se especificará el nivel de confianza se le otorga a una medida concreta.
- *DeviceStatus*. Con la que se expresa la disponibilidad de un dispositivo y otras propiedades relativas al mismo (disponibilidad, estado de carga de la batería, etc.).
- *PositionItem*. Es una clase que agrupa dos formas diferentes de expresar una posición: mediante ejes x, y y z (*Axis*) y mediante latitud y longitud (*Location*). La posición mediante ejes es especialmente útil para dispositivos como los acelerómetros, mientras que la localización sirve para dispositivos que tengan algún mecanismo de posicionamiento global integrado.

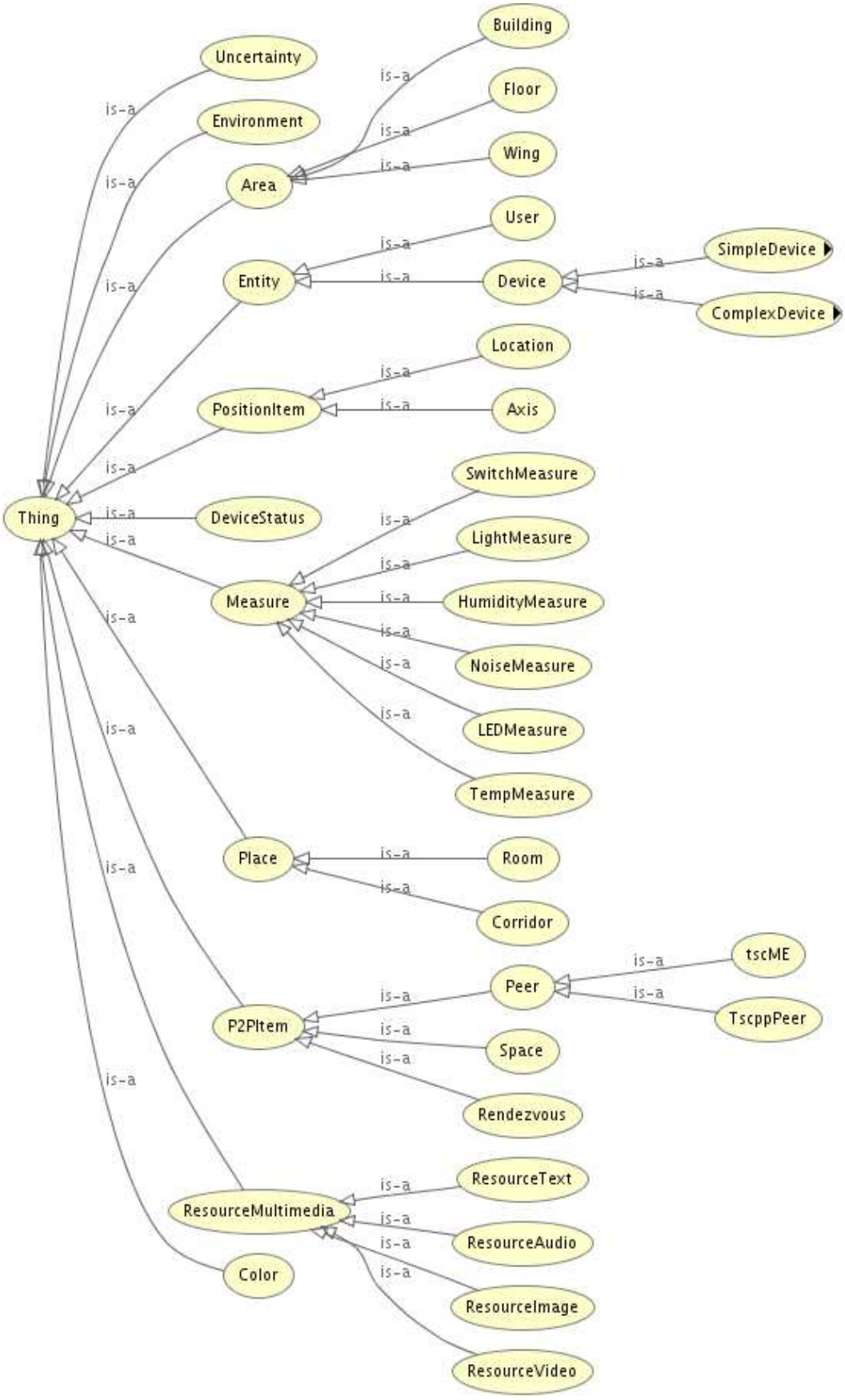


Ilustración 10. Esquema de la ontología de ISMED.

2.1.5.1.1.1 Jerarquía de la clase Device

Como se puede apreciar en la Ilustración 10, se ha diseñado una jerarquía en la que se especifican los tipos de dispositivos que formarán parte del proyecto ISMED y que se puede extender con facilidad.

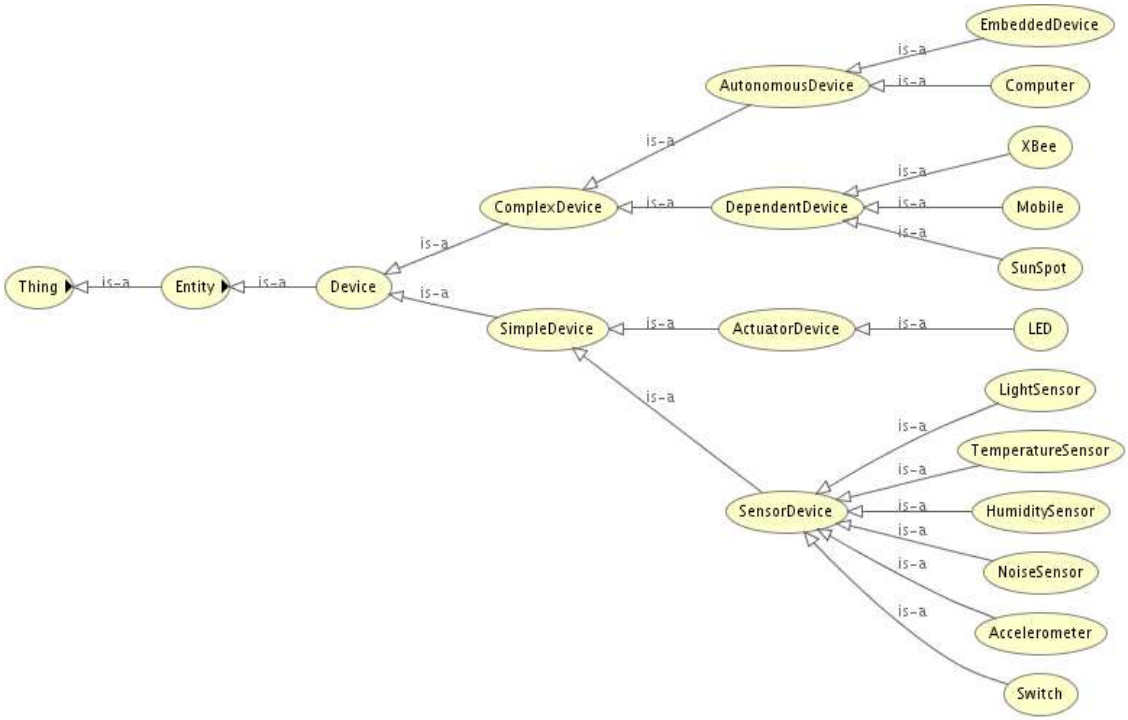


Ilustración 11 Jerarquía de la clase Device

En ella se puede apreciar una primera división en la que se distinguen los dispositivos complejos (*ComplexDevice*), que en esencia serán aquellos que están compuestos por dispositivos simples (*SimpleDevice*), que son los sensores y actuadores que nos permitirán capturar datos básicos del entorno y actuar de forma simple en él..

Entre los dispositivos complejos podemos ver otra subdivisión formada por los dispositivos autónomos (*AutonomousDevice*), que serán aquellos plenamente capaces de soportar el API extendida, y los dispositivos dependientes (*DependentDevice*), que serán aquellos que no soportan el API extendida por tener capacidad de computo más reducida.

Entre los dispositivos autónomos cabe destacar las computadoras (*Computer*) y los dispositivos embebidos (*EmbeddedDevice*), con los que se ha querido hacer referencia a aquellos dispositivos como los Gumstix, que pese a no ser una computadora al uso, si que tienen capacidades equiparables a estas.

2.2 Módulo de aprendizaje

El aprendizaje de patrones de comportamiento a partir de la información recolectada por los sensores, por la complejidad que tiene, demanda una arquitectura dividida en varias capas que al mismo tiempo tendrán varios módulos.

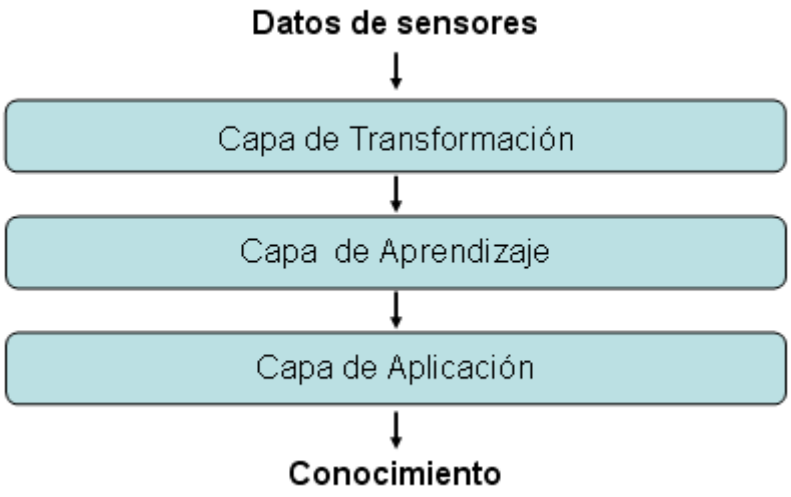


Ilustración 12: Arquitectura global del aprendizaje

2.2.1 Capa de transformación

Una vez que los datos de los sensores hayan sido recogidos, la primera tarea a realizar es inferir acciones relevantes a partir de dichos datos. A veces, la información proveniente de los sensores será relevante en sí, por ejemplo:

desde

2009-10-20T08:15:57,InterruptorLuzOficina,on

inferimos que:

2009-10-20T08:15:57, LuzOficina,on

En este caso, la acción en sí es relevante debido a que podemos inferir la acción que el usuario ha llevado a cabo. Hay otras acciones que necesitan ser inferidos a partir un conjunto de acciones recogidos por los sensores. Por ejemplo la acción de “Entrar a la oficina” no puede ser inferida a partir de la activación de un solo sensor. Para ello,

utilizamos un conjunto de acciones simples. Así,

desde

2009-10-20T08:15:54,MovimientoPasillo,on

2009-10-20T08:15:55,RFID Oficina,on

2009-10-20T08:15:54,MovimientoOficina,on

inferimos que:

2009-10-20T08:15:57, EntrarOficina,on

La manera más básica de inferencia de dichas acciones es mediante plantillas que definen qué acciones tienen que combinarse para inferir acciones relevantes. Además de la acciones, se especifican las constraints a cumplir. Estos constraints son relativos a la orden de las acciones como a la duración de las mismas. Por ejemplo, la plantilla para la acción de “Entrar Oficina” es:

Acciones:

Movimiento Pasillo

RFID Oficina,

Movimiento Oficina

Constraints:

Orden,

$$Movimiento\ Pasillo < RFID\ Oficina < Movimiento\ Oficina$$

Tiempo,

$$T_{Movimiento\ Oficina} - T_{Movimiento\ Pasillo} < 3\ seg.$$

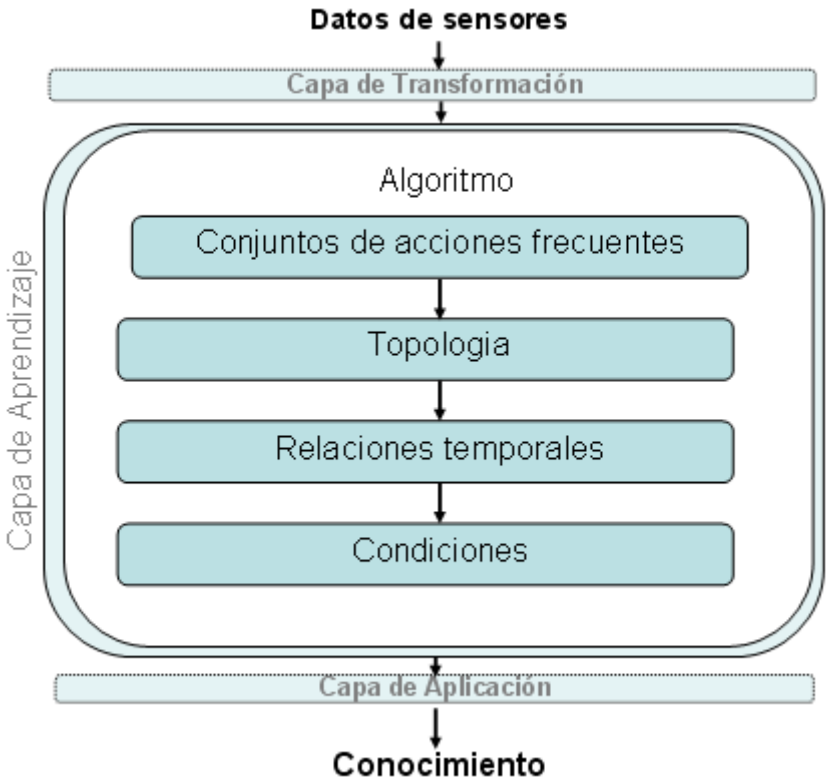
Está claro que la definición de las plantillas está determinada por el conjunto de acciones que queramos identificar así como por los sensores que hay en cada entorno.

2.2.2 Capa de aprendizaje

El objetivo de esta capa es la de extraer los patrones de comportamiento más frecuentes a partir de los datos provenientes de la capa de transformación.

Para ello, es necesario definir qué tipo de patrones se quieren descubrir. En este sentido, el objetivo propuesto en este desarrollo ha sido el de descubrir aquellos patrones que mejor (o más frecuentemente) definan el comportamiento de los usuarios. Además de decidir qué tipo de patrones se quieren descubrir se ha definido cómo se van a representar, ya que la forma en que se quieran representar los patrones influye en el proceso de aprendizaje. Así, se ha decidido representar los patrones mediante secuencias de acciones.

El núcleo de la capa de aprendizaje es el algoritmo que se ha desarrollado. Dicho algoritmo tiene varios módulos que secuencialmente ejecutados descubren los patrones frecuentes. La figura que representa dichos módulos:



2.2.2.1 Descubrimiento de los conjuntos de acciones frecuentes

Una vez que los datos han sido recolectados y las acciones de los usuarios hayan sido definidas, el primer paso es identificar qué conjuntos de acciones son frecuentes en el comportamiento del usuario. Es decir, el objetivo de este primer módulo es la de descubrir qué acciones ocurren frecuentemente juntos.

Para ello, se ha utilizado el algoritmo A priori [14], utilizado para el descubrimiento de reglas de asociación. Este algoritmo identifica aquellos conjuntos de acciones que ocurren más veces que el threshold definido. En este caso, dependiendo de las necesidades de cada entorno, dicho threshold puede variar para encontrar los conjuntos de acciones que se deseen.

2.2.2.2 Descubrimiento de la topología

El conjunto de acciones descubierto en el primer módulo define las acciones que el usuario lleva a cabo frecuentemente, pero no define el orden en que los lleva a cabo. Parece lógico que si esas acciones implican alguna actividad que dichas acciones se lleven a cabo en un orden concreto, siempre teniendo en cuenta la incertidumbre que produce trabajar con el comportamiento de los seres humanos.

Para ello, identificamos las secuencias donde se dan dichas acciones y recogemos el orden de las mismas. Basándose en algoritmos de Workflow Mining [15] se plantea la topología base. Aunque los algoritmos de workflow mining son eficientes, no tienen en cuenta ciertos aspectos que son importantes en los entornos inteligentes.

Puede haber un sub-conjunto de acciones que no tengan un orden definido o que el usuario los haga de forma no ordenada. Un ejemplo sencillo y práctico de ello puede ser que al prepara el desayuno a veces cogemos primero la leche y luego el zumo o viceversa. Para estos casos se ha decidido crear grupos de acciones no-ordenadas. La creación de dichos grupos es definida por varios parámetros que estableces que relaciones bidireccionales serán considerados como grupos no-ordenados.

La repetición de las acciones es otro de los aspectos a tener en cuenta. Algunas acciones que son frecuentes se pueden dar más de una vez en la secuencia. Eso implica el tener que decidir si las diferentes instancias de dichas acciones serán consideradas como diferentes o iguales.

2.2.2.3 Descubrimiento de las relaciones temporales

En el módulo de descubrimiento de la topología ya se le da una dimensión temporal a la secuencia, explicitando qué acción precede a la siguiente acción. Pero esta relación es sólo cualitativa, es decir, definida solamente mediante relaciones temporales de Allen. En un entorno inteligente, por ejemplo para automatizar la siguiente acción, es interesante definir de la forma más precisa posible las relaciones temporales. Así, este módulo trata de descubrir las relaciones temporales más precisas posibles.

Las relaciones temporales se obtienen agrupando las relaciones temporales particulares. Para la agrupación de los valores particulares se utiliza la siguiente fórmula:

$$[\min, \max] = \bar{x} \pm (\bar{x} * tolerance) \text{ donde } \bar{x} = \frac{\sum_{i=1}^{n} a_i}{n}$$

Donde: *tolerance* = desviación tolerada de $\bar{x}$ (%); a_i = la distancia temporal de un elemento; y *n* = número de elementos.
Una vez llevado a cabo el proceso de agrupación, si existe algún grupo que agrupe más instancias que las demandadas, la media de ese grupo será considerada como una relación temporal válida para definir cuantitativamente dicha relación.

2.2.2.4 Descubrimiento de las condiciones

Las relaciones entre dos acciones difícilmente definirán perfectamente todas las instancias de dichas acciones, por lo que hay que definir en qué casos lo definen bien y en cuáles no. Por ello, es necesario este último módulo de descubrimiento de condiciones.

Para ello se crear dos tablas, uno para aquellos casos que son cubiertos por la relación definida, y otro para aquellos que no son cubiertos. Una vez creadas esas dos tablas se utilizan técnicas de clasificación para determinar qué parámetro define mejor las diferencias entre esas dos tablas.

2.2.3 Capa de aplicación

Una vez que los patrones han sido aprendidos, se pueden utilizar para diversas aplicaciones. En este proyecto la aplicación que se ha propuesto es la de entender el comportamiento del usuario. Aún así, una de las aplicaciones más atractivas es la de automatizar la activación/desactivación de los dispositivos implicados.

2.3 Módulo de composición de servicios

Esta sección tiene como base la descripción del diseño del módulo de composición de servicios que tienen como objetivo determinar que combinaciones de funcionalidades ofrecidas por dispositivos del entorno satisfacen la tarea solicitada por el usuario. Para ello se ha decidido emplear un enfoque basado en conversaciones en el que tanto la petición de servicio así como los servicios ofrecidos por los dispositivos están descritas mediante conversaciones que describen el flujo informacional y el contextual de los servicios, siendo este enfoque empleado por primera vez en su aplicación para servicios descritos en base a efectos y condiciones. De esta manera se define que la *Conversation* asociada a un *CompositeCapability* está representada mediante tres elementos principales (ver Ilustración 13): el *Automaton* que describe el flujo de invocación de las *Capability* y/o *Method*[1], el *DataFlow* que determina el flujo de información entre los *Method* (relación entre los *ReturnValue* y los *Parameter*) y por último el *ContextFlow* que define el flujo de la relación de los Effect y las Condition de las *Capability*, así Conversation=<Automaton,DataFlow,ContextFlow>. Así se propone un proceso de composición que integra estos tres aspectos, los cuales son evaluados con el fin de determinar que la composiciones creadas satisfacen las restricciones definidas en la $Conversation_{Req}$. Como se verá más adelante no será necesario evaluar los tres elementos en todos los tipos de composición planteados, siendo únicamente necesario en los casos en los que debido a la posibilidad de existencia de interdependencias entre las *Capability* de la $Conversation_{res}$ el orden de las *Capability* de la misma y la $Conversation_{res}$ resultante sean diferentes (donde 1≤res≤RES, siendo RES el número total de elementos *Conversation* calculados en el proceso de composición y que emparejan con $Conversation_{Req}$, siendo *ResultConversationList* =

[1]De aquí en adelante con el fin de simplificar las expresiones se va a emplear la denominación *Capability* para representar la expresión *Capability* y/o *Method*.

{ < $Conversation_{res}$, $MatchResult_{res}$ > }$_{res=1,...,RES}$) y $MatchResult_{res}$ el valor que representa la distancia entre $Conversation_{res}$ y $Conversation_{Req}$.

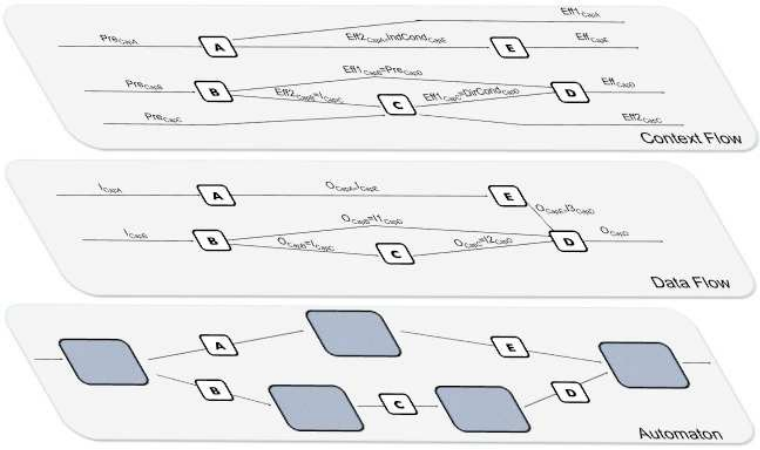


Ilustración 13. *Conversation* representado mediante *Automaton*, *DataFlow* y *ContextFlow*.

2.3.1 Propiedades del mecanismo de emparejamiento basado en *Conversation*

El elemento principal a emparejar será el *Automaton*, en el cual el mecanismo de composición basado en *Automaton* tomará como base la solicitud ($Automaton_{Req}$) y lo emparejará respecto al conjunto de *Automaton* de las *Capability* de los *Service* que se encuentran en *Registry*). Para ello el sistema de composición empleará diferentes métodos de emparejamiento relativos a la granularidad del proceso de composición soportado, ofreciendo así un ámplio espectro de configuración para así ofrecer diferentes rangos de rendimiento y resultado, que serán empleados para seleccionar que tipo de composición es más adecuado para que entorno. Así, el $Automaton_{Req}$ representado en la *Ilustración 14* podrá ser compuesto de varias maneras como será descrito más adelante.

Previa a la definición descripción de los diferentes métodos de composición soportados por el sistema a continuación se describen las premisas básicas que tienen que ofrecer las $Conversation_{res}$ creadas tras el proceso de composición.

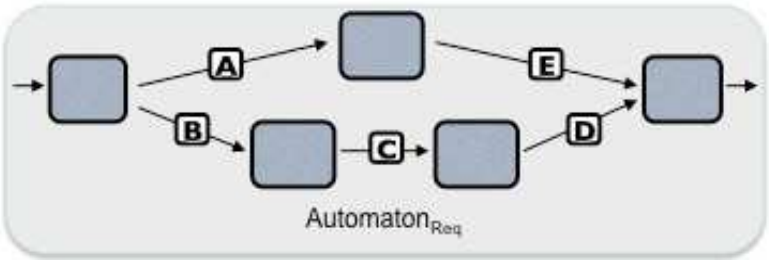


Ilustración 14. *Automatón requerido*

- Todas y cada una de las *Conversation*$_{res}$ generadas tras el proceso de composición tienen que emparejar correctamente con la requerida (*Conversation*$_{Req}$). Las *Conversation*$_{res}$ no solo tienen que emparejar a nivel de *Automaton*, *DataFlow* y *ContextFlow* sino que también tienen que emparejar en términos de descripción global, es decir, tienen que emparejar tanto la signatura como la especificación global de la *Conversation*$_{Req}$ y las *Conversation*$_{res}$ (ver Ecuación 1). Para tal fin se empleará el mecanismo ***resultChecking(Conversation***$_{ada}$***, Conversation***$_{toCheck}$***)***, donde *Conversation*$_{toCheck}$ representa a la *Conversation* que hay que validar (mediante el proceso descrito en la sección 3.3.4), pasando ésta a ser una *Conversation*$_{res}$ en caso de que sea válida respecto a *Conversation*$_{ada}$, siendo *Conversation*$_{ada}$ una de las *Conversation* generadas en el proceso de adaptación de la *Conversation*$_{Req}$ descrito en la sección 3.3.3:

Ecuación 1

$\exists Conversation_{res} \Rightarrow (\exists AutomatonMatch, \exists ContextFlow_{Match}, \exists DataFlow_{Match}$ y $resultChecking(Conversation_{ada}, Conversation_{toCheck}) = true)$

- Todas y cada una de los *Automaton*$_{reg}$ (donde $1 \leq reg \leq REG$, siendo *REG* el número de elementos *Conversation*$_{reg}$ que se encuentran en el *Registry*) que formen parte del *Automaton*$_{res}$ resultante deben de ser ejecutados en toda su extensión. Por ejemplo, en caso de que el *Automaton*$_{Req}$ esté compuesto por tres *Automaton*$_{reg}$ éstos tienen que ser representados en toda sus extensión, es decir, tienen que comenzar en un *InitialState* (*State*$_{0reg}$) y tienen que finalizar en un *FinalState* (*F*$_{reg}$) válido. De esta manera se asegura que todos los Automaton$_{reg}$ que forman parte de *Automaton*$_{res}$ podrán ser invocados

correctamente y todos ellos habrán llegado al *FinalState* una vez finalizada la ejecución del $Automaton_{res}$, evitando así que queden elementos $Automaton_{reg}$ en espera (ver Ecuación 2).

Ecuación 2

$$inv(Conversation_{res}) \Rightarrow (\text{si } state(Automaton_{res}) = F_{res} \Rightarrow \forall\, Automaton_{reg} \in Automaton_{res}\ state(Automaton_{reg}) = F_{reg})$$

En toda su extensión, el enfoque de composición presentado empleará diversos tipos de esquemas de composición, los cuales ofrecen diferentes resultados para la Ilustración 14. Inicialmente esta clasificación de esquemas se basa en la diferenciación del tipo de composición empleado, donde *CompositionType* = {compType : *compType* ∈ {*CapabilitySelection*, *ConversationSelection*}}, tomando como base la granularidad empleada para emparejar la solicitud respecto a lo ofrecido en el repositorio (ver Ilustración 15):

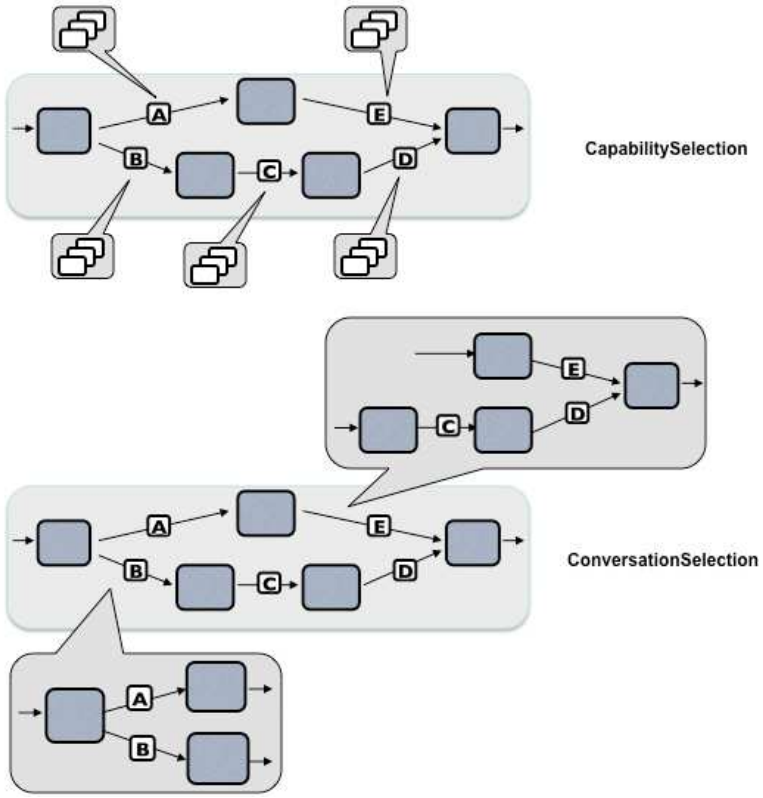


Ilustración 15. Tipos de composición soportada.

1. **Selección de operaciones orientada a conversaciones (*CapabilitySelection*):** Mediante esta técnica el $Automaton_{res}$ es calculado mediante la integración de los $Automaton_{reg}$ asociados a los $SimpleCapability_{reg}$ del repositorio. Este método es uno de los más empleados para la composición de servicios en entornos de computación ubicua. Este método trata de emparejar una a una cada una de las *SimpleCapability* solicitadas en la $Conversation_{Req}$, con al menos una de las $SimpleCapability_{reg}$ que forman parte del repositorio. Así se obtiene una $Conversation_{res}$ construida exclusivamente en base a $Conversation_{reg}$ de tipo más simple, las cuales son aquellas *Conversation* que representan las *SimpleCapability*.

2. **Integración de Conversaciones orientada a Conversaciones (*ConversationSelection*)**: Mediante esta técnica el $Automaton_{res}$ es calculado mediante la combinación de los $Automaton_{reg}$ representados mediante elementos *Conversation*, obteniendo así una $Conversation_{res}$ construida en base a una o más $Conversation_{reg}$ que pueden ser tanto de tipo simple (*Conversation* que describe la invocación de un único *Capability*) o complejo (*Conversation* con construciones avanzadas, como: bucles, condicionales, etc.). Este tipo de composición representa un superconjunto del anterior: $CapabilitySelection \subseteq ConversationSelection$.

2.3.2 Integración de Conversaciones orientada a Conversaciones

La **Selección de operaciones orientada a conversaciones** no ofrece flexibilidad para la composición ya que solo es posible realizarlo de una única manera, sin embargo, la denominada **Integración de Conversaciones orientada a Conversaciones** ofrece cuatro dimensiones de configuración, las cuales permiten representar la flexibilidad del mecanismo de composición, consiguiendo de esta manera variantes de composición diferentes en términos de rendimiento y resultados ofrecidos (ver Ilustración 16).

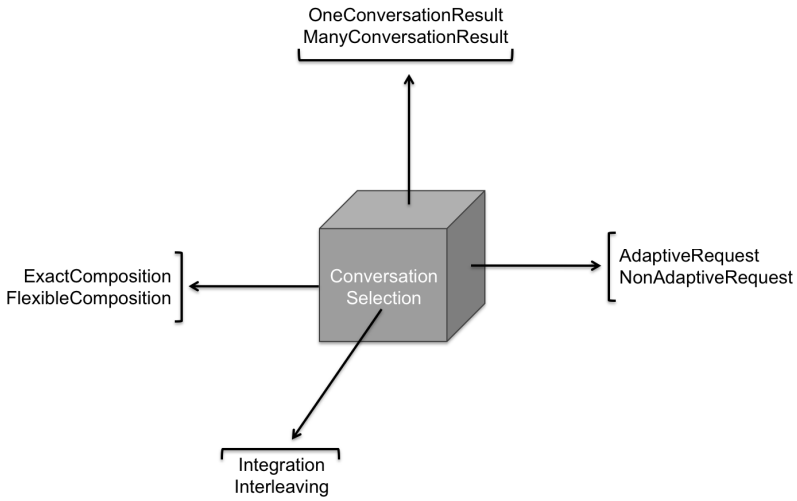


Ilustración 16. Dimensiones para configurar las variantes de la composición basada en conversaciones.

Como se verá a continuación estas cuatro dimensiones afectan a todos los aspectos de la composición, comenzando desde la flexibilidad de la petición (*AdaptivityMode*), técnica empleada (*CompositionTechnique*) y grado de relajación (*RelaxationMode*) del mecanismo de composición, hasta llegar al tipo de resultado final deseado (*FinalCompositionType*), así *ConversationSelection=<AdaptivityMode*,*CompositionTechnique*,*RelaxationMode*,*Final CompositionType>*. A continuación se ofrece una breve descripción de los diferentes parámetros que componen el elemento *ConversationSelection* además de ejemplos que facilitan su compresión:

a) **Adaptabilidad de la *Conversation* solicitada (*AdaptivityMode*):** Pueden existir casos en los que el $Automaton_{Req}$ pueda ser modificado con el fin de conseguir un emparejamiento más completo. Este tipo de caso se da cuando el orden de las *Capability* que forman parte del $Automaton_{Req}$ puedan ser modificadas sin que ello afecte al $ContextFlow_{Req}$ y al $DataFlow_{Req}$ del mismo. De esta manera al variar las posibles combinaciones del $Automaton_{Req}$ se amplía el espectro de posibles $Automaton_{res}$ que emparejan con lo solicitado. Por ejemplo, en el caso en el que se solicite un $Automaton_{Req}$ que incremente la intensidad de la luz y que después ponga la música de un estilo en concreto, en caso de que estas dos funcionalidades no tengan relación entre ellas (no exista $DataFlow_{Req}$ o $ContextFlow_{Req}$ que las relacione directa o

indirectamente), el sistema trataría de buscar a su vez también una combinación de $Automaton_{Req}$ (equivalente pero en el que el orden de las *Capability* de la misma es diferente, siendo el resultado final el mismo) en la que primero se ponga la música de un estilo en concreto y después se incremente la intensidad de la luz. De esta manera, existen dos variantes del sistema, donde *AdaptivityModen* = {*adaptMode* | *adaptMode* ∈ {*NonAdaptiveRequest*,*AdaptiveRequest*}}:

- ***NonAdaptiveRequest***: Esta opción se emplea cuando no es posible o no se desea adaptar la $Conversation_{Req}$.

- ***AdaptiveRequest***: En cambio en la denominada *AdaptiveRequest* la $Conversation_{Req}$ es analizada con el fin de determinar las posibles combinaciones equivalentes existentes, con el fin de ofrecer una mayor flexibilidad al mecanismo de emparejamiento. En este caso el elemento a modificar es el $Automaton_{Req}$ para así crear nuevas versiones de la misma en la que el orden de las *Capability* es alterado, siendo tanto el *DataFlow* como el *ContextFlow* equivalente, es decir, entre las mismas *Capability* que en la $Conversation_{Req}$ original. Este proceso dará lugar al elemento *AdaptedReqConversationList* donde **¡Error!** y *ADA* representa el número de variantes de $Conversation_{Req}$ que se han generado durante el proceso.

Como se puede ver en la Ilustración 17 en la parte superior se muestra el *AdaptedReqConversationList* el cual está formado exclusivamente por $Automaton_{Req}$, en caso de que AdaptivityMode=NonAdaptiveRequest y en la parte inferior de la se puede apreciar como el elemento AdaptedReqConversationList está formado por las diferentes combinaciones de $Automaton_{Req}$ en el caso de que AdaptivityMode=AdaptiveRequest, donde se muestran las nuevas variantes $Automaton_{ada}$ calculadas, siempre y cuando se respeten el $DataFlow_{Req}$ y el $ContextFlow_{Req}$. en la parte superior se muestra el *AdaptedReqConversationList* el cual está formado exclusivamente por $Automaton_{Req}$, en caso de que *AdaptivityMode=NonAdaptiveRequest* y en la parte inferior de la se puede apreciar como el elemento

AdaptedReqConversationList está formado por las diferentes combinaciones de $Automaton_{Req}$ en el caso de que AdaptivityMode=AdaptiveRequest, donde se muestran las nuevas variantes $Automaton_{ada}$ calculadas, siempre y cuando se respeten el $DataFlow_{Req}$ y el $ContextFlow_{Req}$.

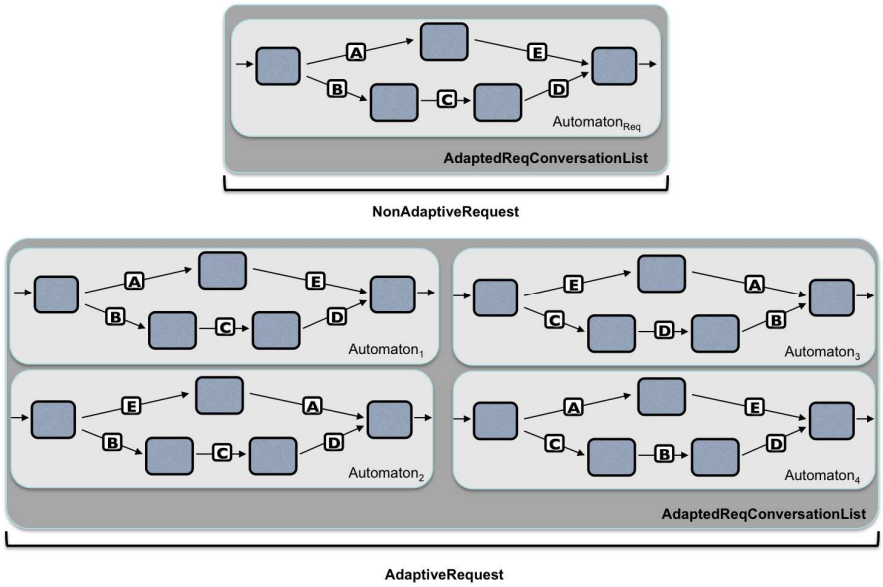


Ilustración 17. Variantes de ConversationReq en base a la adaptabilidad de AdaptivityMode

b) **Técnica de composición (*CompositionTechnique*)**: La técnica de composición define cual será el proceso empleado para construir el $Automaton_{res}$ equivalente al $Automaton_{Req}$[2] (ver Ilustración 18), siendo éste de varios tipos, tal y como se describe a continuación, *CompositionTechnique*={*compTech* | *compTech*∈{*Integration*,*Interleaving*}}:

- ***Integration***: En esta técnica el $Automaton_{res}$ es construido mediante la integración de varios $Automaton_{reg}$ para así conseguir elementos $Automaton_{res}$ válidos y completos. Para ello se concatenan varios

[2]Durante la presentación de las diferentes propiedades y hasta el inicio de la sección **¡Error! No se encuentra el origen de la referencia.** se considerará que *AdaptivityMode=NonAdaptiveRequest*, con el fin de facilitar la comprensión de la misma.

Automaton$_{reg}$ (concatenando el estado final de un *Automaton*$_{reg}$ con el estado inicial de otro *Automaton*$_{reg}$) uno detrás de otro hasta conseguir un comportamiento equivalente al solicitado.

- ***Interleaving***: Esta técnica emplea un enfoque más flexible que el de *Integration* en el que la composición del *Automaton*$_{res}$ es llevado a cabo entrelazando varios *Automaton*$_{reg}$. Es decir, se emplea un mecanismo más flexible que el anterior en el que es posible entrelazar partes de *Automaton*$_{reg}$ para conseguir un *Automaton*$_{res}$ equivalente. En este caso el sistema que invoque la *Conversation*$_{res}$ tendrá que ser capaz de gestionar las múltiples conexiones que tendrá que mantener abiertas debido al entrelazado. Cabe destacar que *Integration*$\subseteq$*Interleaving*.

En la Ilustración 18 se puede observar como en el el tipo de composición *Integration* el servicio es compuesto integrando, es decir, integrando consecutivamente los tres *Automaton*$_{reg}$ para así conseguir un *Automaton*$_{res}$ equivalente al solicitado. Sin embargo, en el caso de *Interleaving* pese a que a primera vista la concatenación de las *Automaton*$_{reg}$ no sigue el mismo orden de ejecución (concretamente en el caso de la rama de ejecución *B*–*C*–*D* solicitada la *Conversation*$_{res}$ dispone de la rama *B*–*D*–*C*) de las *Capability* de *Automaton*$_{Req}$ la composición es considerada válida. Esto es debido a que el mecanismo soporta el entrelazado entre conversaciones. Gracias a ello es posible comenzar con un *Automaton* y antes de que finalice éste, parar su ejecución para comenzar con otro *Automaton* para así volver al anterior una vez que haya finalizado el segundo y después seguir con la ejecución del primero hasta que finalice éste. Así en el caso de *B*–*D*–*C*, en el momento de ejecución de la composición, el mecanismo que invoca a las *Capability* tendrá que comenzar con el *Automaton*$_{4}$ y tras invocar la *Capability B* parará la ejecución de dicho *Automaton* y comenzará con la ejecución del *Automaton*$_{6}$ para así invocar la *Capability C* y posteriormente volver a continuar con la invocación de la *Capability D* y así llegar al estado final de cada uno de los *Automaton*$_{reg}$. Todo esto será posible solo si el motor de composición es capaz de soportar el entrelazado mediante la gestión de sesiones.

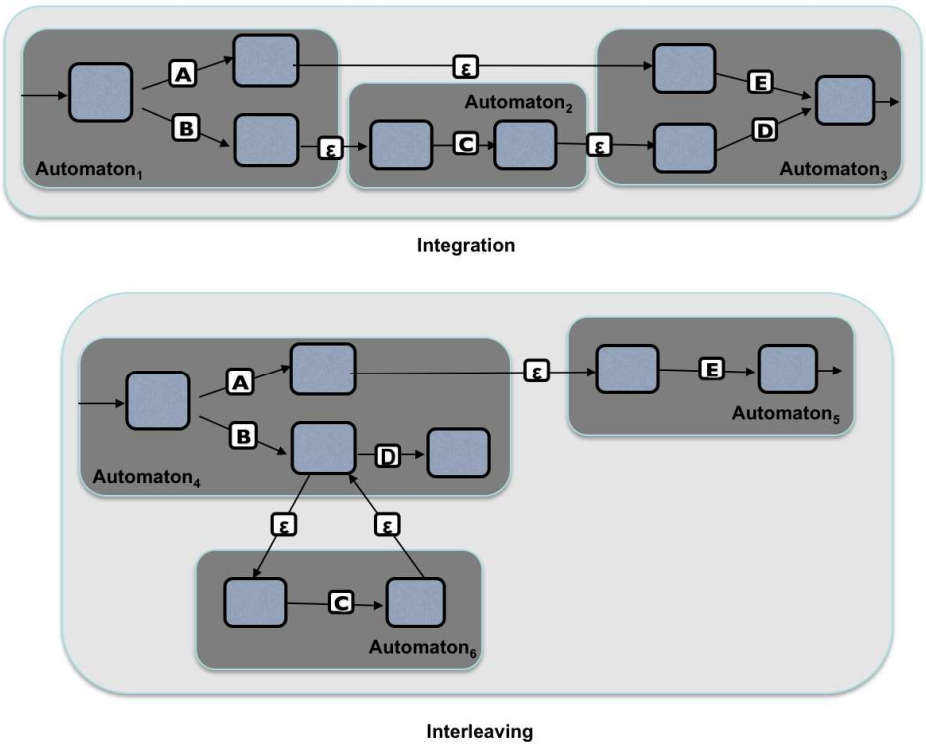


Ilustración 18. Técnicas de composición soportadas.

c) **Grado de relajación soportada en la composición (*RelaxationMode*):** Existen dos variantes del mecanismo en función del grado de relajación soportado para el resultado final (ver Ilustración 19), donde *RelaxationMode* = {*relaxMode* | *relaxMode*∈{*ExactComposition*,*FlexibleComposition*}}:

- ***ExactComposition***: Esta técnica presenta un enfoque en el que el $Automaton_{res}$ tiene que ser estructuralmente equivalente al $Automaton_{Req}$. Es decir, tiene que tener el mismo número de elementos *Capability*.

- ***FlexibleComposition***: Esta técnica presenta un enfoque más relajado que el anterior ya que permite que el $Automaton_{res}$ tenga en su conjunto más elementos *Capability* que las solicitadas en $Automaton_{Req}$, siempre y cuando éstas *Capability* extra no afecten al $DataFlow_{res}$, $ContextFlow_{res}$ ni a la solicitud global de la *CompositeCapability*, es decir, no produzca efectos indeseados. Cabe destacar que *ExactComposition* ⊆ *FlexibleComposition*.

Como se puede apreciar en la Ilustración 19, en el caso de *ExactComposition* la composición es exacta ya que dispone del mismo número de elementos *Capability* que la solicitud de la Ilustración 14. Sin embargo, en el caso de *RelaxedComposition* la composición resultante ofrece más elementos *Capability* que los solicitados pero como en global la composición resultante tiene el mismo comportamiento que el solicitado (no produce más *Effect* que los deseados ni necesita más elementos *Parameter* que los de $Conversation_{Req}$), éste es considerado válido, previa comprobación mediante el proceso ***resultChecking(*** $Conversation_{ada}$***,*** $Conversation_{toCheck}$***)***, descrito en 3.3.4.

d) **Tipo de *Conversation* resultante (*FinalCompositionType*):** Dependiendo de como se construye la *Conversation* resultante existen dos variantes (ver Ilustración 20), así *FinalCompositionType*={*finalCompType* | *finalCompType*∈{*OneConversationResult*,*ManyConversationResult*}}:

- **$Automaton_{res}$ compuesto exclusivamente por un único $Automaton_{reg}$ (*OneConversationResult*)**: En este caso la técnica exclusivamente trata de emparejar el $Automaton_{Req}$ con un único $Automaton_{reg}$ del repositorio. Es decir, el $Automaton_{res}$ estará compuesto exclusivamente por un único $Automaton_{reg}$. Esta técnica tiene un rendimiento alto, pero al contrario solo determina un subconjunto de los posibles resultados.

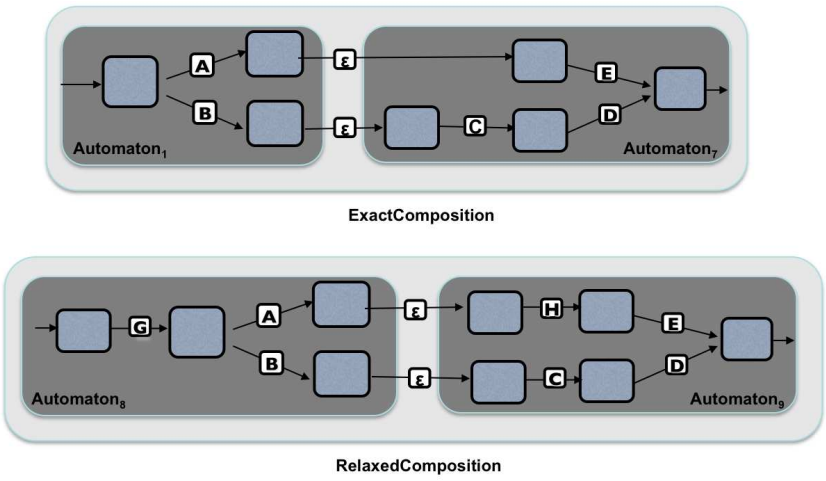


Ilustración 19. Flexibilidad en los resultados en función del grado de relajación.

- **Automaton$_{res}$ compuesto por al menos un Automaton$_{reg}$ (*ManyConversationResult*)**: En este caso la técnica trata de emparejar el *Automaton$_{Req}$* tanto con uno como con varios *Automaton$_{reg}$*. Es decir, el *Automaton$_{res}$* estará compuesto mediante la integración de varios *Automaton$_{reg}$*. Esta técnica tiene un rendimiento inferior al anterior, pero por el contrario ofrece todos los resultados posibles además de integrar a la anterior técnica, de esta manera *OneConversationResult*⊆*ManyConversationResult*.

Como se puede apreciar en la Ilustración 20 el tipo *OneConversationResult* no es en esencia una técnica de composición ya que lo único que se en carga es de comprobar que existe una *Conversation$_{reg}$* que es equivalente al solicitado, es decir, que comenzando en un estado inicial llegue a un estado final empleando elementos *Capability* equivalentes. El segundo en cambio, *ManyConversationResult* construye el resultado final combinando dos *Automaton$_{reg}$*.

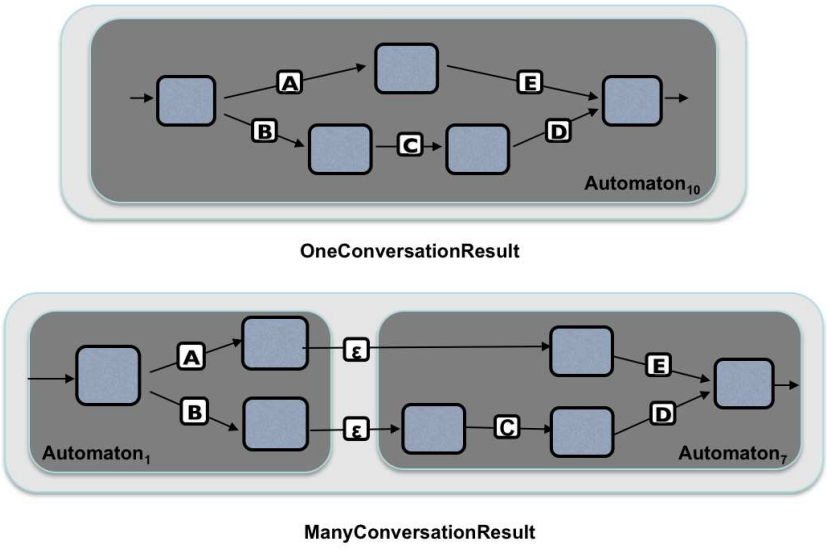


Ilustración 20. Tipo de composición en función del tipo de resultado deseado.

Tomando como base las cuatro dimensiones previamente definidas, es posible representar las diferentes variables que definen todos los tipos de *ConversationSelection* que se van a analizar en el presente trabajo. Sin embargo, también es posible representar el mecanismo *CapabilitySelection*, definiendo para ello

una *ConversationSelection* en la que *ConversationSelection* = <*NonAdaptiveRequest,Integration,*ExactComposition,ManyConversationResult> y empleando un subconjunto de Registry. Este subconjunto, denominado $Registry_{OnlySimpleCapability}$[3], estará constituido exclusivamente por aquellas $Conversation_{reg}$ que representan una Conversation de tipo simple, es decir, estará únicamente compuesto por aquellas SimpleCapability visibles publicadas en el *Registry*. Por defecto y en caso de que no se defina lo contrario en la petición de descubrimiento el valor asignado y que establece la flexibilidad de la composición será la composición de tipo *CapabilitySelection*.

[3]En adelante $Registry_{OSC}$.

2.4 Módulo de descubrimiento

2.4.1 Consideraciones

Como se ha comentado en epígrafes anteriores el módulo de descubrimiento basa su funcionamiento en el API que ofrece el módulo de modelado y coordinación semántica. El módulo de descubrimiento da soporte a la necesidad de conocer qué dispositivos y servicios asociados están disponibles en ecosistema de ISMED.

En la etapa actual nos centraremos en el descubrimiento de los dispositivos existentes en el entorno, dejando para posteriores revisiones el apartado de descubrimiento de los servicios exportados por dichos dispositivos.

2.4.2 Primitivas a implementar

Tras la evaluación del escenario y del problema, se ha visto la necesidad de dar respuesta a las siguientes necesidades: descubrimiento de dispositivos y servicios, descubrimiento automático y sistema de suscripción. A continuación se detallarán cada una de las áreas involucradas en módulo de descubrimiento y las primitivas que ofrecerá.

2.4.2.1 Descubrimiento de dispositivos y servicios

Cada uno de los nodos tendrá la capacidad de conocer al resto de dispositivos conectados en su mismo espacio y por extensión el descubrimiento de sus servicios asociados. La búsqueda o descubrimiento se subdivide en dos posibles escenarios:

2.4.2.1.1 Descubrimiento

Para esta primera hipótesis se ha de considerar la necesidad de un dispositivo, en un instante dado, de determinar que otros dispositivos están presentes en el sistema. El objetivo por tanto será conocer al resto de integrantes de la red. Para ello haciendo uso de la primitiva *query* (ofrecida por el módulo de modelado), se lanzará una consulta al espacio de tripletas por cada tipo de dispositivo conocido. El resultado de dicha query será transformado o mapeado a objetos que serán almacenados en una caché, con el objetivo de minimizar el lanzamiento de consultas al espacio. La primitiva

a implementar es:

- *discoverDevices.*

En un futuro se incorporarán a este módulo nuevas primitivas que aporten la funcionalidad requerida para el descubrimiento de servicios.

2.4.2.1.2 Descubrimiento automático

En un intento de automatizar el descubrimiento de dispositivos se ha modelado un protocolo de comunicación. Mediante la adopción de este protocolo un dispositivo será capaz de determinar la entrada o salida de otras unidades en la red de ISMED. El protocolo establecido es el siguiente:

1) Cuando un dispositivo se integra en la red realizará una suscripción (por cada tipo de dispositivo) para recibir las posibles notificaciones de los nuevos dispositivos. En este caso la primitiva a invocar será: subscribe (módulo de modelado). El template especificado para la suscripción será del tipo:

   ? <rdf:type> <ismed:DeviceType>

2) Cuando un dispositivo se integra en la red, notificará al resto de dispositivos su llegada invocando la primitiva *advertise* (módulo de modelado). De igual modo, en el momento de abandonar el sistema, avisará a sus vecinos mediante el servicio *advertise*. En ambos casos el template especificado será del tipo:

   <prefix:deviceName> <rdf:type> <ismed:DeviceType>

3) Si un dispositivo presente en la red recibe una notificación procedente de otro nodo, comprobará si se trata de una notificación de entrada o salida.

Para satisfacer el protocolo adoptado, se implementarán las siguientes primitivas que serán ofrecidas por el módulo de descubrimiento:

- **start**: Suscripción para recibir notificaciones de nuevos dispositivos

- **stop**: Elimina la suscripción del dispositivo.

Si bien estas primitivas serán utilizadas en el sistema final, quizás deban ser modificadas, adaptadas o ampliadas para contemplar el descubrimiento de servicios mediante el sistema de notificaciones.

Cabe destacar en el caso del descubrimiento automático, que el rendimiento o efectividad del mismo está ligada a la eficacia del sistema de notificación en que se basa TripCom y a la existencia o no de nodos rendezvous. Por tanto el rendimiento de este mecanismo deberá ser evaluado para determinar su viabilidad.

2.4.2.2.3. Sistema de suscripción

Siguiendo la idea propuesta en apartado de Descubrimiento automático, el módulo de descubrimiento requerirá la utilización del sistema de suscripción / notificación provisto por el API del módulo de modelado. Este sistema será ampliado para contemplar la suscripción a eventos como la llegada de nuevos servicios a la red de ISMED.

Por otra parte gracias este sistema, un dispositivo será capaz de suscribirse para recibir notificaciones a cerca de determinados eventos como por ejemplo las diferentes mediciones que puedan realizar los sensores de un nodo SunSpot.

2.4.3 Diagrama de clases

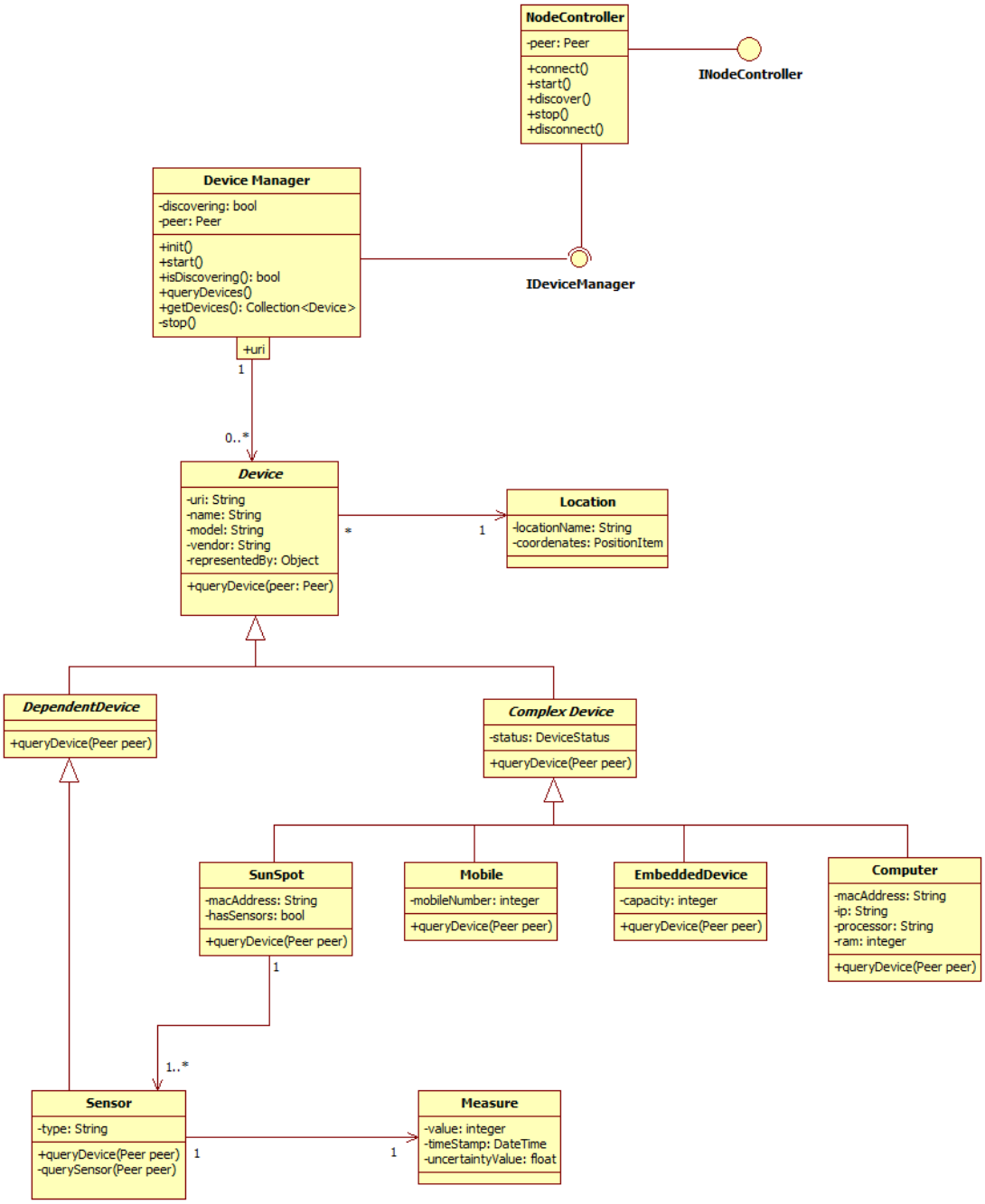


Ilustración 21. Diagrama de clases del módulo de descubrimiento.

2.5 Módulo de razonamiento

El problema del razonamiento distribuido ha demostrado ser un problema de una entidad tal que requeriría un esfuerzo y unos recursos que exceden claramente a los que el proyecto ISMED puede aportar.

Es por ello, que el razonamiento que se realizará, al igual que en TripCom, será a nivel de cada nodo. A continuación se especifica el tipo de razonamiento que se puede lograr en cada nodo que use el framework desarrollado para ISMED.

- Sesame [16]. Pese a que el tipo de repositorio que se configure no tiene porque permitir inferencia (como es ocurre en el caso de tsc++), Sesame ofrece un motor de inferencia sobre RDFS.

- Owlim [17]. Tiene un motor de inferencias que permite razonar sobre RDFS (existe una optimización a costa de limitar su expresividad) y sobre dialecto de OWL cercado a OWLite (que ignora cardinalidad para valores mayores de 1).

- Dispositivos móviles.

  Se han encontrado serias dificultades para encontrar un motor de inferencia que pueda ser ejecutado en un dispositivo móvil. A lo largo de este curso, se publicó un artículo en el que se mencionaba un esperanzador motor de inferencia para dispositivos móviles JavaME con el perfil CDC llamado µOR [18]. µOR hace entailments para un subconjunto de OWL Lite.

  Desafortunadamente a día de hoy ese motor no es ni público, ni está preparado para la versión CLDC de JavaME (aunque según indicaban sus creadores se encontraban trabajando en ello).

  Mientras se busca una alternativa viable con la cual solucionar dicho problema, en tscME no existirá inferencia alguna. Las clases RecordStoreDataAccess y MemoryDataAccess almacenarán tripletas agrupándolas en espacios y grafos análogamente a Sesame u Owlim, y mediante MicroJena se podrá consultar cuando una tripleta cumple un patrón concreto, pero nunca inferir nada que no haya sido explícitamente escrito previamente.

3 IMPLEMENTACIÓN

3.1 Módulo de modelado y coordinación semántica

El módulo de modelado y coordinación semántica ofrece una infraestructura para permitir la comunicación entre los distintos módulos que hay en cada nodo de la red. Es decir, no hace nada por sí sólo.

Es por ello que pese a que se han construido multitud de aplicaciones apoyándose en el mismo, ninguna muestra mejor la esencia de las primitivas implementadas que aquella aplicación construidas para probar la correcta integración entre los distintas versiones del middleware del módulo de coordinación.

Dichas aplicaciones se han hecho tanto para los nodos modificados de tsc++, como para los de tscME.

3.1.1 tsc++ modificado

Esta aplicación permite probar cada una de las primitivas mediante 4 pestañas:

- Connection. En la que se pueden especificar parámetros de configuración de tsc++ tales como dónde se guardarán los archivos de log, el nombre del peer, etc.

- Spaces. Desde aquí se podrá llevar a cabo la gestión de los espacios, y se podrán crear, unirnos a ellos o abandonarlos.

- Write. Desde aquí se puede llamar a la primitiva *write* seleccionando un fichero en el que haya escritas ciertas tripletas (pudiendo ser el formato rdf u turtle) y volcándolas en un espacio determinado.

- Query. En esta pestaña se llamarán las consultas sobre el espacio basadas en tripletas *query*, *take* o *read*. El resultado podrá ser consultado después en View>Results, o compararlo directamente con las tripletas que se espera que devuelva especificadas en un archivo.

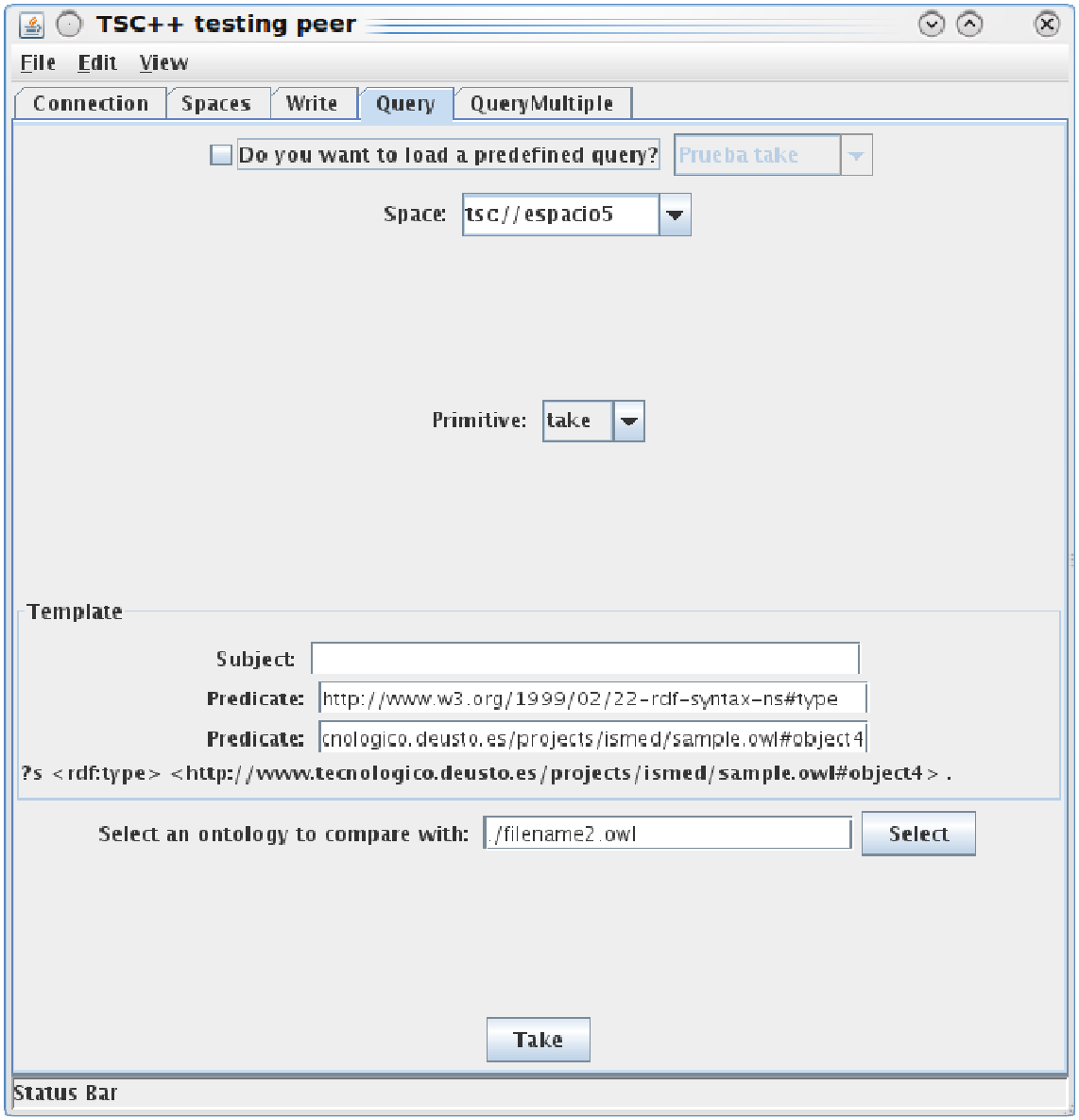


Ilustración 22. Take sobre el espacio *tsc://espacio5*, de la que se esperan los resultados definidos en el archivo *filename2.owl*.

- QueryMultiple. Esta pestaña permite definir consultas SPARQL de una forma bastante básica y lanzar las consultas sobre el espacio.

3.1.2 tscME

Salvando las distancias surgidas por las limitaciones a la hora de definir una interfaz de usuario rica en dispositivos móviles, la aplicación desarrollada es análoga en comportamiento a la aplicación descrita en el anterior epígrafe, pero apoyándose en la librería tscME.

Ilustración 23. Capturas de pantalla de la aplicación de testeo de tscME.

El diagrama mostrado en la Ilustración 24 explica el funcionamiento de la aplicación. Primero se especifican los datos del *Rendezvous* al que un peer móvil necesita conectarse, después se permite gestionar espacios, y sobre espacios ya creados se pueden llevar a cabo primitivas básicas como *query*, *read*, *take* o *write*.

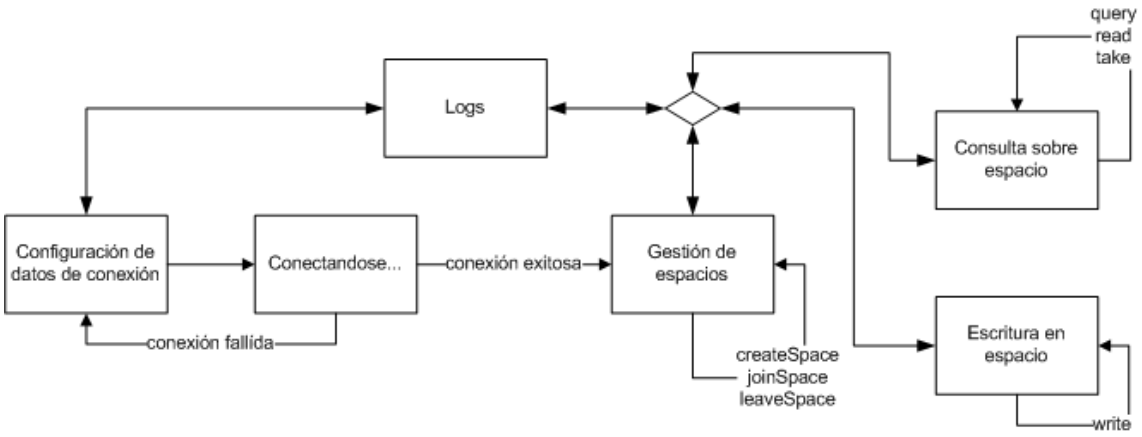


Ilustración 24 . Esquema de la aplicación de prueba de integración de tscME.

3.1.3 Ejemplo de funcionamiento

Las aplicaciones descritas permiten probar otras aplicaciones y gestionar los espacios creados. Además, permitirían construir escenarios tan complejos como quisiésemos, logrando una mejor comprensión del funcionamiento de Triple Space. Un escenario podría ser el mostrado en la Ilustración 25.

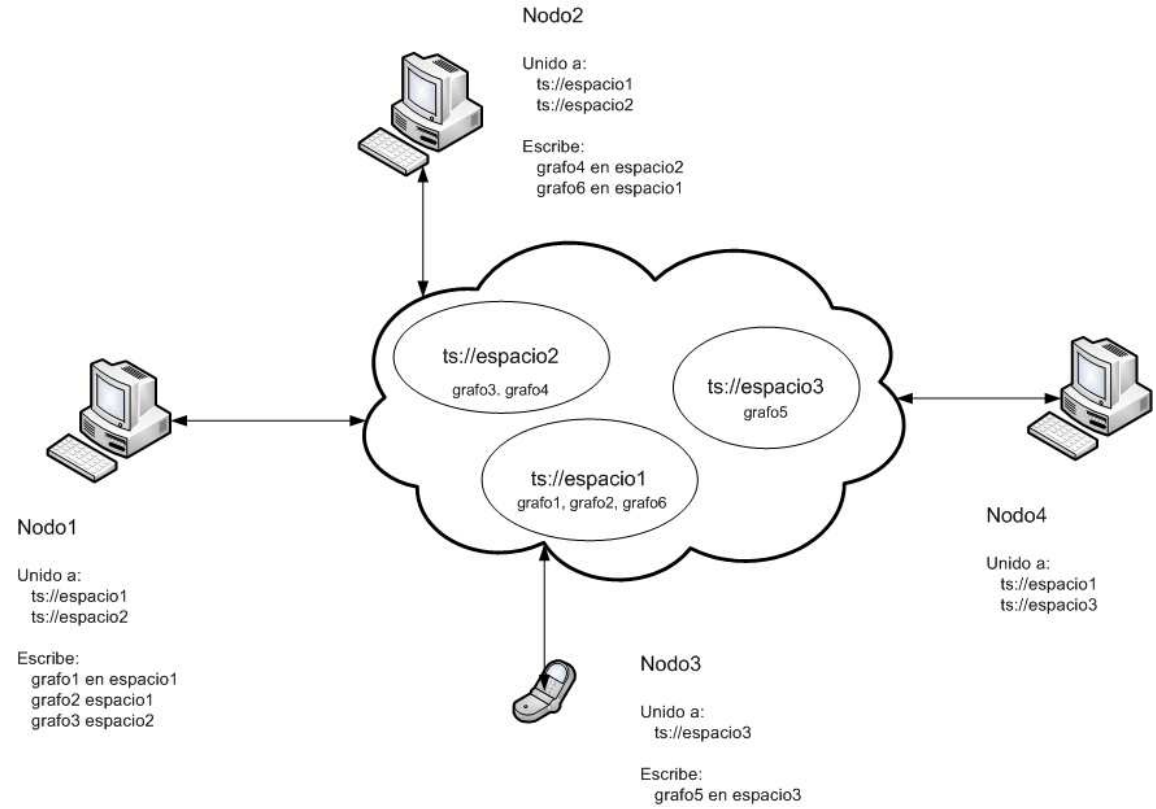


Ilustración 25. Situación de partida para el ejemplo explicado en este epígrafe.

En él se puede apreciar mejor como si el nodo4 realizase una consulta al espacio3 preguntando por información contenida en el grafo3, nadie le respondería nada porque esa información la tendría el nodo1 asociada al espacio2.

De la misma forma, si el nodo1 realizase una consulta *query* sin especificar sujeto y predicado, pero indicando el objeto “http://perro” (“?s ?p <http://perro> .”) sobre el espacio2, recibiría una respuesta del nodo2 con todas aquellas tripletas del grafo4 que tuviesen “http://perro” en el objeto y las uniría con aquellas de su repositorio local que cumpliesen con dicho patrón (las correspondientes al grafo3).

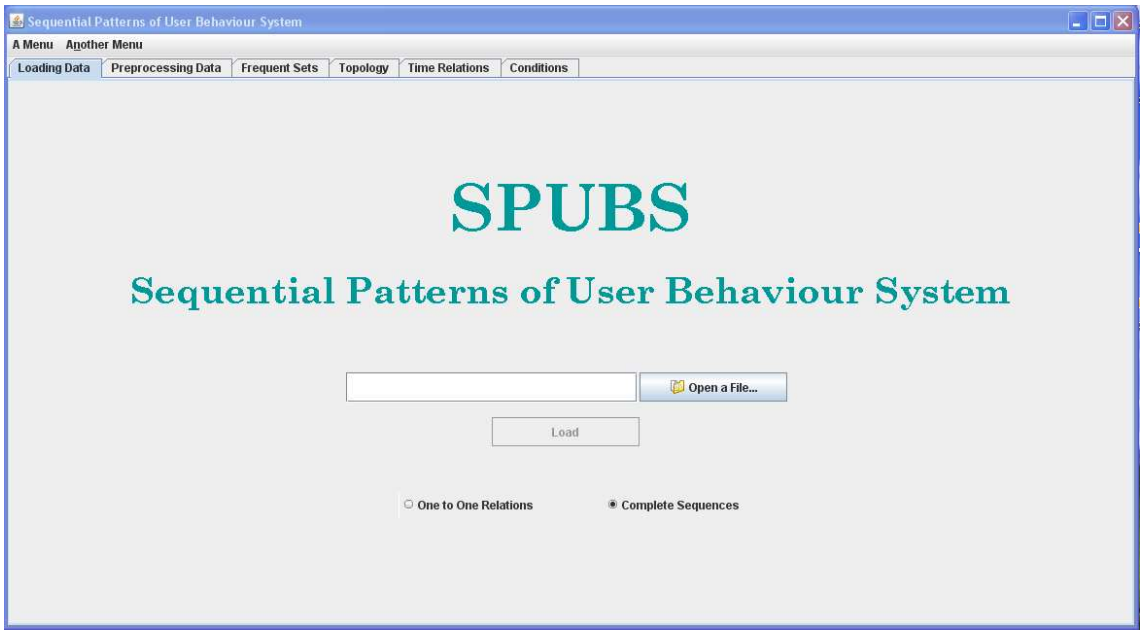
Si el nodo4 realizase una *take* sin especificar sujeto, predicado u objeto sobre el espacio3, recibiría el grafo5 del nodo3 la primera vez, pero nada la segunda, porque el nodo3 habría eliminado ese grafo de su repositorio la primera vez.

Finalmente, si el nodo2 hiciese una *read* sobre el espacio1 preguntando por una información contenida en el grafo1, el grafo2 y el grafo4, recibiría el grafo1 o el grafo2 desde el nodo1, dado que el grafo4 estaría en el espacio2 y un mismo nodo sólo puede devolver cómo máximo un grafo ante una *take* o una *query*.

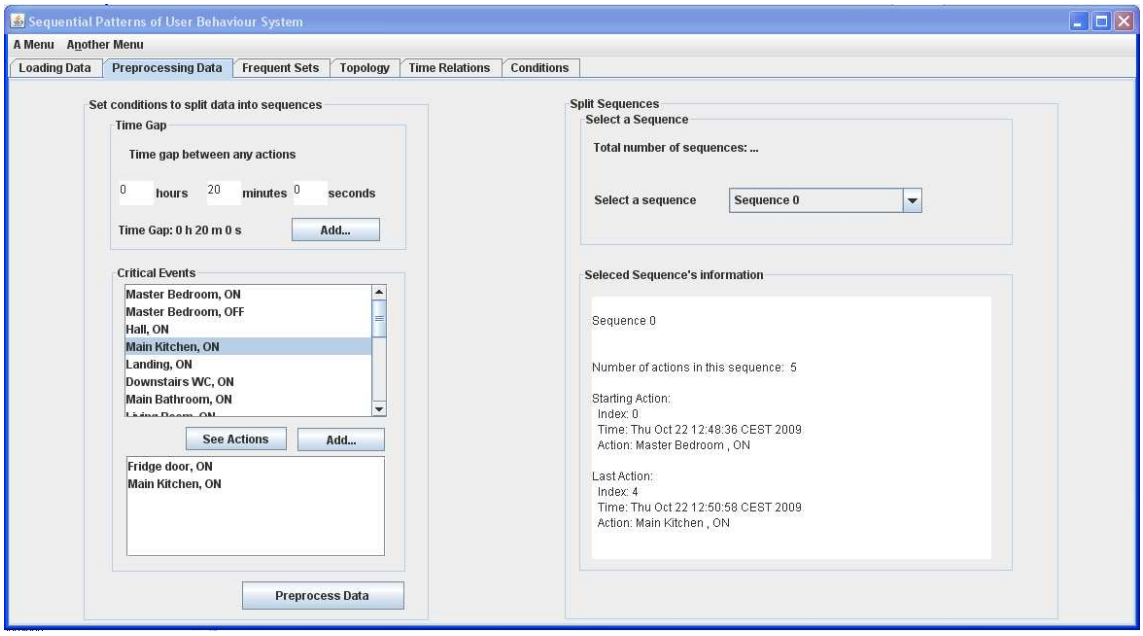
3.2 Módulo de aprendizaje

El módulo de aprendizaje se ha desarrollado, para facilitar la integración con los otros módulos, en Java. Además se ha desarrollado un interfaz gráfico que facilita la interacción del usuario con el sistema.

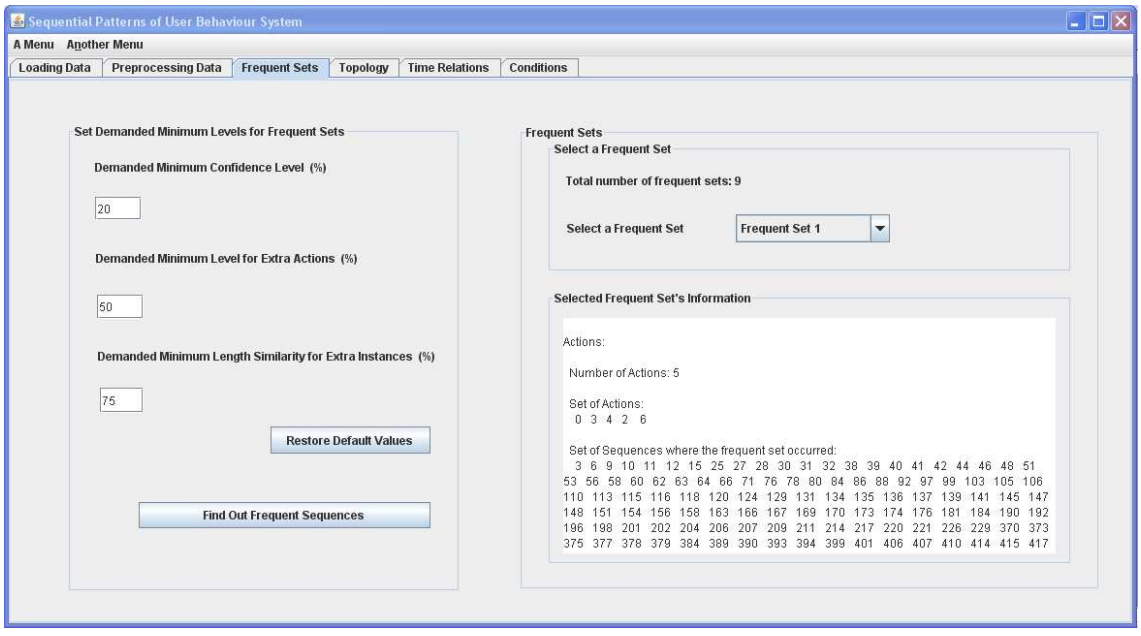
A continuación, utilizando el interfaz gráfico, se detallarán los pormenores de la implementación, así como las diferentes opciones que posibilita dicha implementación. La pantalla inicial permite seleccionar el conjunto de datos a utilizar en el proceso de aprendizaje.



Como se ha detallado en el apartado de diseño, el primer paso es la transformación del conjunto de datos. Para ello, se permiten utilizar tanto información temporal como información acerca de las acciones. El resultado de este paso es el conjunto de datos transformado.

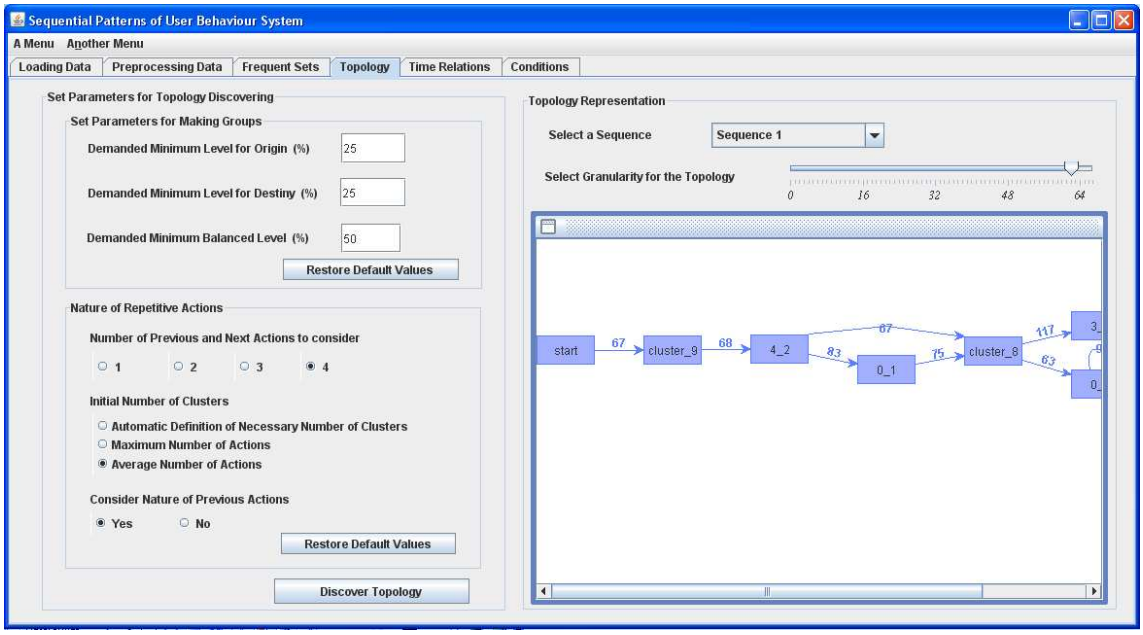


Una vez que los datos hayan sido transformados, el módulo de aprendizaje permite llevar a cabo todos los pasos definidos en el diseño. El primero de ellos es el de descubrir los conjuntos de acciones que son frecuentes. Para ello, se pueden definir varios tresholds que definirán el treshold para considerar un conjunto como frecuente o no.

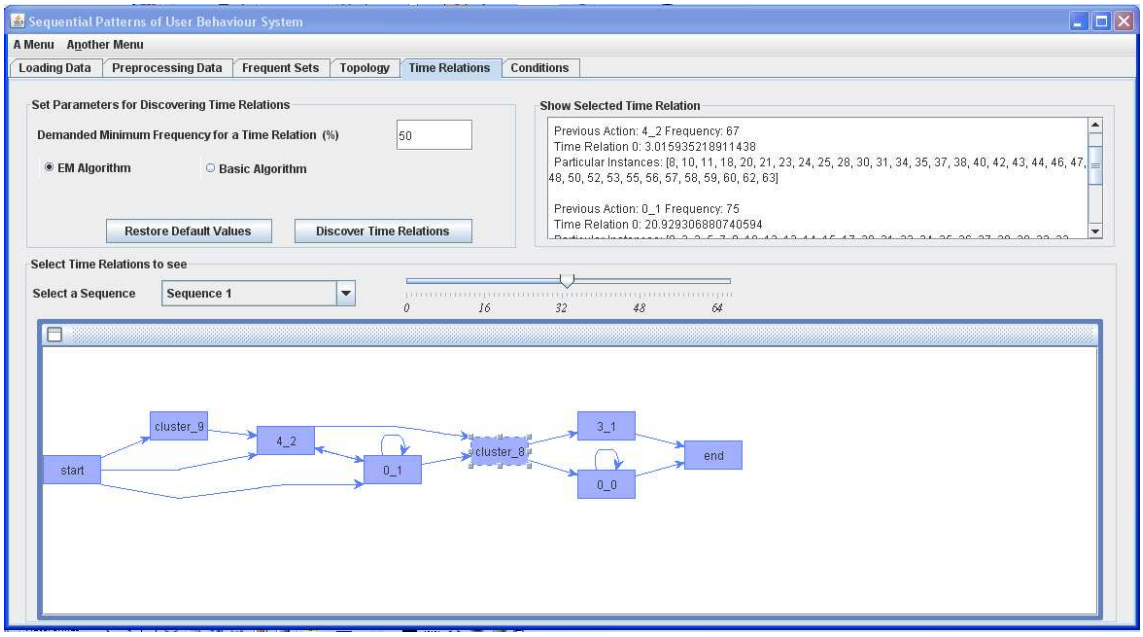


El siguiente paso es descubrir la topología de los diferentes conjuntos de datos y darles el sentido de una secuencia. Para ello, el interfaz permite definir parámetros que deciden si un sub-conjunto de acciones es no-ordenado o no. También permite

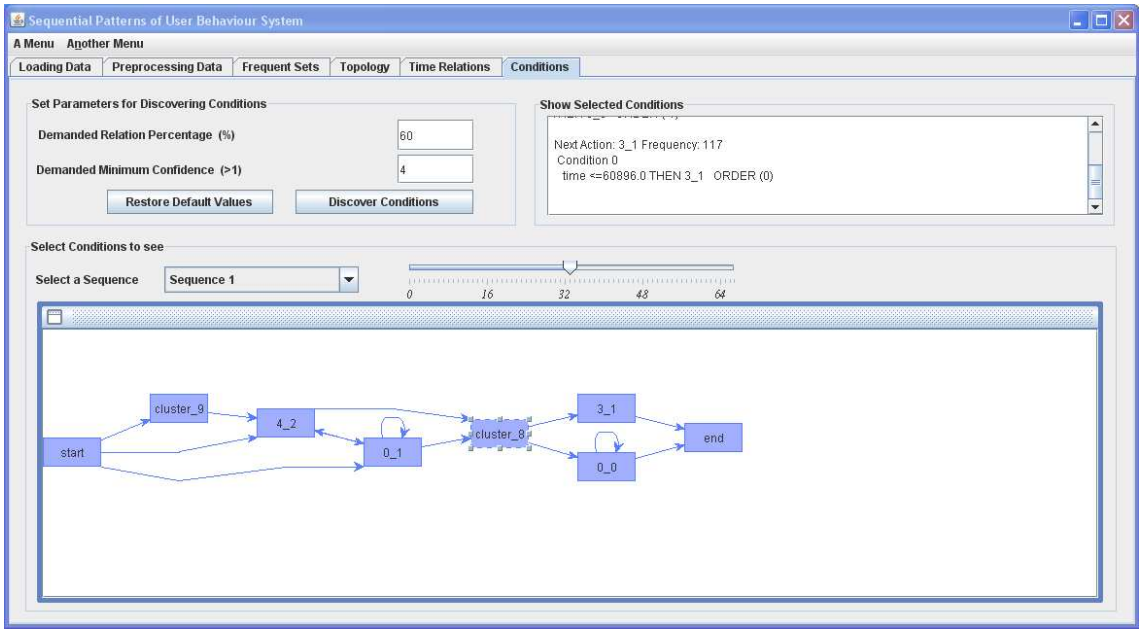
definir parámetros para acciones repetitivas.



Después de la topología vienen las relaciones temporales. Para ello, se han implementado dos técnicas que pueden ser seleccionadas mediante el interfaz.



Finalmente, debido a que existen diferentes opciones a partir de una acción, el módulo de aprendizaje lleva a cabo el descubrimiento de las condiciones.



3.2.1 Integración del módulo de aprendizaje con el módulo de coordinación semántica

Como se ha explicitado en el apartado de diseño, el descubrimiento de las condiciones se lleva a cabo generando dos tablas donde se explicitan las instancias cubiertas y no-cubiertas. A continuación, mediante técnicas de clasificación, se extraen las condiciones que diferencian esas dos tablas. Para dicha separación se almacenan en dichas tablas la información proveniente de los sensores de contexto.

Al utilizar toda la información del contexto (temperatura, humedad, luminosidad,…) se dan casos donde la condición que mejor define la relación no tiene significado. Por ejemplo, si la relación es entre las acciones “entrar oficina” y “encender la luz”, es lógica que dichas acciones estén relacionadas por la luminosidad de la oficina, es decir, que el usuario encienda la luz si la luminosidad de la oficina no es suficiente.

En una primera aproximación se utilizaba toda la información del contexto para definir las condiciones, obteniendo a veces condiciones no muy lógicas que relacionaban las acciones con variables del entorno de diferente naturalidad. Por ejemplo, siguiendo con el ejemplo anterior, se podía llegar a definir una condición que decía que el usuario encienda la luz de la oficina cuando la humedad de la misma sea menor que x.

Así, se decidió que integrando el módulo de aprendizaje con el módulo de coordinación semántica se podría utilizar la información semántica del entorno para

llevar a cabo un descubrimiento de condiciones mucho más eficiente. Para ello, se decidió utilizar dos parámetros que definen la naturalidad de las acciones: la ubicación de la acción y el tipo de información del contexto (temperatura, humedad, luminosidad,…).

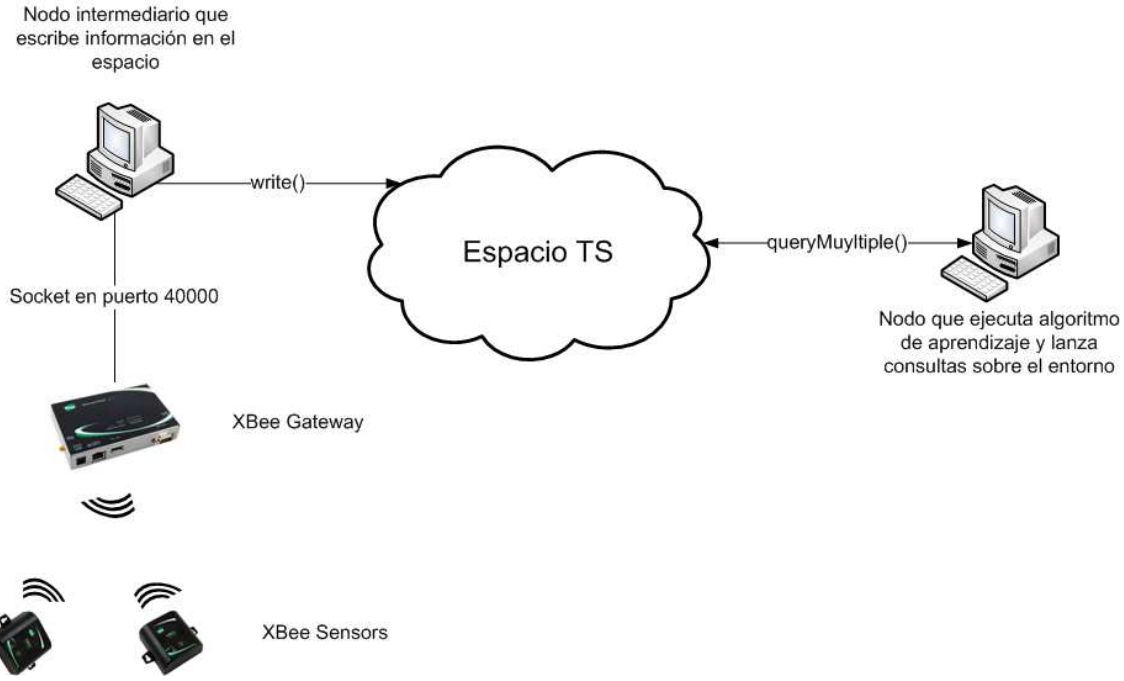


Ilustración 26. Esquema de la aplicación de integración desarrollada.

Como se puede apreciar en el diagrama, para dar solución a dicha problemática, se definieron dos nodos que trabajan sobre un mismo espacio: un nodo que escribiese y otro que consultase la información escrita por él.

El nodo escritor, introduce la información de dónde se encuentran qué dispositivos, y los sensores que componen estos dispositivos (por ejemplo, de humedad o de luminosidad).

La particularidad de este nodo es que al estar implementado en Java, no se puede cargar en el Gateway XBee [10], que sólo permite cargar scripts desarrollados en Python. Por ello, el Gateway se comunicará mediante un socket con un servidor hecho en Java que será quien a su vez escriba las tripletas que recibe por ese socket en el espacio TS.

El nodo consultor es utilizado por el algoritmo de aprendizaje, que es el encargado de consultar que sensores son de una naturaleza concreta y están ubicados en un lugar

concreto, utilizando la primitiva *queryMultiple* con la consulta mostrada en la Tabla 4.

Tabla 4 . Consulta SPARQL que devuelve los dispositivos que tienen sensores de una natuzaleza concreta (donde TYPE es la uri de un tipo de sensor) y se encuentran en el lugar especificado (donde PLACE es la uri de un lugar).

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ismed: <http://www.tecnologico.deusto.es/projects/ismed/ismed.owl#>
CONSTRUCT {
  ?mota ismed:name ?name .
}
WHERE {
  ?mota rdf:type ismed:XBee .
  ?mota ismed:name ?name .
  ?mota ismed:madeUpOf ?sensor .
  ?sensor rdf:type <TYPE> .
  ?mota ismed:locatedIn <PLACE> .
}
```

Como se puede apreciar, en esta primera iteración, sólo se escribe información relativa a los dispositivos que existen en el entorno y no las capturas que estos mismos van realizando a lo largo del tiempo. El siguiente paso comprendería escribir dichas medidas y mantenerlas actualizadas, y sería muy sencillo de implementar siguiendo la filosofía de la aplicación descrita en este epígrafe.

3.3 Módulo de composición de servicios

Tras describir las diferentes combinaciones soportadas por el sistema de composición, el siguiente paso consiste en describir el algoritmo base empleado para el emparejamiento de las *Conversation*. Este algoritmo soporta la flexibilidad del elemento *ConversationSelectio*n y de esta es capaz de dar soporte a los diferentes tipos de composición propuestos y a sus combinaciones. El objetivo del mecanismo de emparejamiento de *Conversatio*n es comparar la estructura de la $Conversation_{Req}$ solicitada respecto a la estructura de las $Conversation_{reg}$ del repositorio, teniendo en consideración las estructuras de control de las mismas y así calcular las posibles combinaciones de $Conversation_{reg}$ que pueden ser equivalentes al $Conversation_{Req}$. En

la Ilustración 27 se muestra el proceso a seguir para llegar al resultado final, el cual consta de cuatro procesos principales:

- Por una parte se separan los procesos relacionados con el *Registry*, el cual está dividido en dos procesos:

    - Inicialmente es necesario seleccionar aquellas $Conversation_{reg}$ que son candidatos (generando así la lista *CandidateConversationList*) para emplearlos en el proceso de composición mediante la función ***candidateSelection(Registry*, *$Conversation_{Req}$*, *FinalCompositionType*, *RelaxationMode)***. Donde *CandidateConversationLis*t = $\{Conversation_{cand}\}_{cand=1,...,CAND}$ y $1 \leq cand \leq CAND$ siendo *CAN*D el número total de elementos candidatos seleccionados, donde $CAND \leq REG$. Esta selección variará en funcióndelavariable F*inalComposition*T *yp*e ylavariable *RelaxationMod*e ytomarácomo base el conjunto de las $Conversation_{reg}$ del *Registr*y además de la $Conversation_{Req}$.

    - Una vez obtenida la lista *CandidateConversationLis*t del proceso anterior, el siguiente paso será construir el denominado $Conversation_{CandidateRegistry}$[4] mediante la función ***getCandidateRegistryConversation(CandidateConversationList*, *CompositionTechnique*, *RelaxationMode*, *FinalCompositionType)***. El elemento $Conversation_{CR}$ estará formado por el elemento $Automaton_{CandidateRegistry}$[5] el cual representa el *Automato*n global del conjunto de conversaciones que son candidatas así como todos los *DataFlow* y *ContextFlow* de las *Conversatio*n que forman parte de $Automaton_{CR}$. Las transformaciones para la generación del $Automaton_{CR}$ variarán en función de las variables *CompositionTechnique*, *RelaxationMode* y *FinalCompositionType*, tal y como se verá más adelante.

- Y por otra el denominado ***getAdaptations(Conversation$_{Req}$*, *AdaptivityMode*) el cual está relacionado con la solicitud (*Conversation$_{Req}$*)** que evalúa las

[4] En adelante $Conversation_{CR}$

[5] En adelante $Automaton_{CR}$

posibles adaptaciones de la *Conversation*$_{Req}$ en caso de que así se establezca en *AdaptivityMode*. De esta manera se obtendrá el conjunto de *AdaptedReqConversationLis*t que será empleado en el proceso posterior de composición.

- Por último está el proceso ***conversationMatch(AdaptedReqConversationList*, *Conversation*$_{CR}$)** toma como base las estructuras creadas en las fases anteriores y lleva a cabo el algoritmo de composición que determina el conjunto de *Conversation*$_{res}$ que emparejan correctamente con lo solicitado y que forman el elemento *ResultConversationList*.

Todos las variantes de composición descritas en la sección 2.3.1 emplean la misma base para la composición, es decir, emplean el mismo algoritmo ***conversationMatch(AdaptedReqConversationList*, *Conversation*$_{CR}$)** para calcular las diferentes composiciones posibles. La única diferencia entre estos mecanismos estriba en la estructura del *Conversation*$_{CR}$ empleado y en la *AdaptedReqConversationLis*t solicitada, generadas a partir del tipo de composición *CompositionType* que se desee realizar, teniendo además en consideración las cuatro dimensiones anteriormente definidas en caso de que *CompositionType* = *ConversationSelection*. Así, a continuación se procederá a describir como se lleva a cabo cada uno de los procesos mencionados anteriormente, detallando como son calculados en función de los parámetros definidos para la composición.

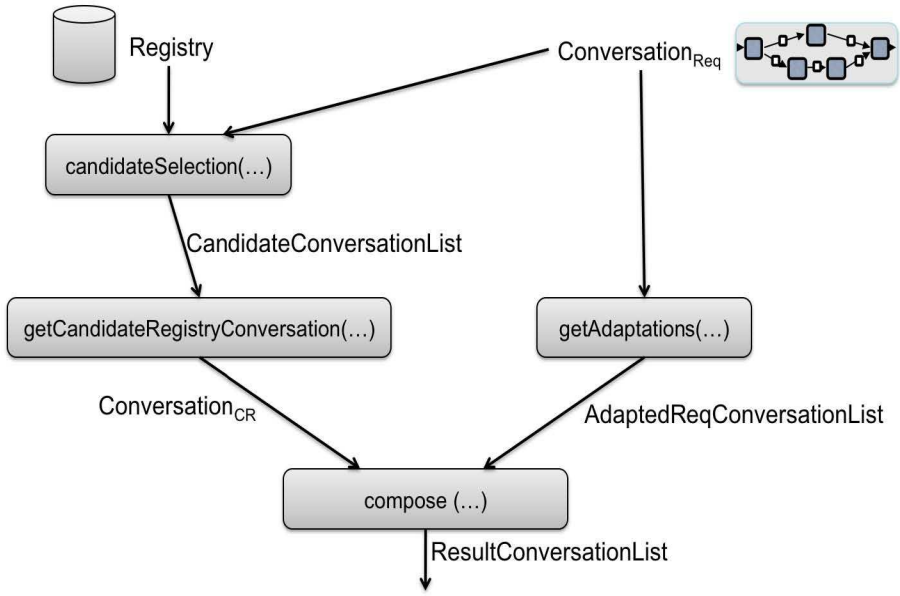


Ilustración 27. Proceso global de emparejamiento de *Conversation*.

3.3.1 Selección de candidatos

La selección de candidatos ***candidateSelection(Registry*, *Conversation$_{Req}$*, *FinalCompositionType*, *RelaxationMode*)** consiste en el proceso en el que se toma como base todo el *Registry* (o bien *Registry$_{OSC}$* en caso de que el tipo de composición a realizar sea *CapabilitySelection*) al completo y se realiza una selección de aquellas *Conversation$_{reg}$* que son candidatas para la composición final respecto a la *Conversation$_{Req}$*. Este proceso está estrechamente ligado a las dimensiones *FinalCompositionType* y *RelaxationMode*, las cuales dan lugar a cuatro posibles combinaciones que se corresponden con tres funciones (ver Ecuación 3), siendo el valor final de *CandidateConversationLis*t devuelto por la función definido de la siguiente manera:

Ecuación 3

$$CandidateConversationList = \begin{cases} ALOMatch(Registry, Conversation_{Req}) & (a) \\ FullMatch(Registry, Conversation_{Req}) & (b) \\ FullMatchRelaxed(Registry, Conversation_{Req}) & (c) \end{cases}$$

(a) $ManyConversationResult$

(b) $(ExactComposition \ \wedge OneConversationResult)$

(c) $(FlexibleComposition \ \wedge OneConversationResult)$

Las funciones arriba mencionadas definen el conjunto de *Conversation*$_{reg}$ candidatos de la siguiente manera[6]:

- ***ALOMatch(Registry*, *Conversation*$_{Req}$*)***: Esta función crea la lista de candidatos tomando en consideración la premisa en la que la *Conversation*$_{reg}$ tiene que existir al menos una *SimpleCapabilit*y (*SCapablity*') que empareje con al menos una de las *SimpleCapabilit*y (*SCapablity*'') que componen la *Conversation*$_{Req}$. En caso de que exista emparejamiento, la *Conversation*$_{reg}$ será añadida a la lista *CandidateConversationLis*t como *Conversation*$_{cand}$, en caso contrario dicha *Conversation*$_{reg}$ será descartada.

Ecuación 4

$$\text{Si } \exists\ SCapability' \in Conversation_{reg}, \exists\ SCapability'' \in Conversation_{Req} : SimpleCapability_{Match}(SCapability'', SCapability') \Rightarrow Conversation_{cand} = Conversation_{reg}$$

- ***FullMatch(Registry*, *Conversation*$_{Req}$*)***: Esta función considera como válidas sólo aquellas *Conversation*$_{reg}$ en la que para cada una de las *SimpleCapabilit*y (*SCapability''*) de *Conversation*$_{Req}$ existe al menos una *SimpleCapabilit*y (*SCapability'*) equivalente en *Conversation*$_{reg}$ y viceversa (equivalente a un grafo bipartito). Es decir, para que una *Conversation*$_{reg}$ sea candidata para *CandidateConversationList*, todas sus *SimpleCapabilit*y tienen que emparejar

[6] La selección considerará el tipo de solicitud requerido, empleando diferentes tipos de emparejamiento, como son *SimpleMethod*$_{Match}$, *SimpleCapability*$_{Match}$ o bien la combinación de ambas.

positivamente en *Conversation*$_{Req}$ y todas las *SimpleCapability* de *Conversation*$_{Req}$ tienen que emparejar con al menos una *SimpleCapability* de *Conversation*$_{reg}$. Además el número de *SimpleCapability* de *Conversation*$_{reg}$ y el número de *SimpleCapability* de *Conversation*$_{Req}$ tiene que ser el mismo ya que si es menor o mayor éste no será considerado como candidato. Donde |*Conversation*| representa de manera genérica el número de *SimpleCapability* de una *Conversation*.

Ecuación 5

$$\forall\ SCapability' \in Conversation_{reg},\ \forall\ SCapability'' \in Conversation_{Req}$$
$$\text{si } \exists\ SimpleCapability_{Match}(SCapability'', SCapability')$$
$$\wedge |Conversation_{reg}| = |Conversation_{Req}| \Rightarrow Conversation_{cand} = Conversation_{reg}$$

- ***FullMatchRelaxed(Registry, Conversation$_{Req}$)***: Esta función es equivalente a ***FullMatch(Registry, Conversation$_{Req}$)***, sin embargo en este caso, el número de *SimpleCapability* (*SCapability'*) de *Conversation*$_{reg}$ tiene que ser mayor o igual al número de *SimpleCapability* (*SCapability''*) de *Conversation*$_{Req}$ para que sea considerado candidato. Sin embargo, el presente caso no será equivalente a un grafo bipartito, ya que no todos los *SimpleCapability* de *Conversation*$_{reg}$ emparejarán positivamente con al menos una *SimpleCapability* de *Conversation*$_{Req}$, pero si todas las *SimpleCapability* que componen *Conversation*$_{Req}$ tendrán al menos un emparejamiento positivo con una *SimpleCapability* de *Conversation*$_{reg}$.

Ecuación 6

$$\forall\ SCapability' \in Conversation_{reg},\ \forall\ SCapability'' \in Conversation_{Req}$$
$$si\ \exists\ SimpleCapability_{Match}(SCapability'', SCapability')$$
$$\wedge\ |Conversation_{reg}| \geq |Conversation_{Req}| \Rightarrow Conversation_{cand} = Conversation_{reg}$$

El objetivo de esta fase es doble, ya que además de determinar las *Conversation*$_{reg}$ candidatas, la función ***candidateSelection(Registry, Conversation$_{Req}$, FinalCompositionType, RelaxationMode)*** calcula la similitud entre cada una de las *Transition* de *Automaton*$_{Req}$ y cada una de las *Transition* de *Automaton*$_{reg}$ candidatas (comparando aquellos *Symbol* que representan a las *Capability* por una parte y los que

representan las *Condition* por otra). Así todos los procesos que requieren del cálculo de similitud entre elementos *Capability* o elementos *Condition* es realizada previamente, de esta manera la función ***conversationMatch(AdaptedReqConversationList*, *Conversation*$_{CR}$**)** solo se encargará de realizar el emparejamiento de los *Automaton* y simplemente empleará la lista de la similitud de las *Transition* para así determinar el grado de relación de las mismas de manera rápida en el proceso de composición.

3.3.2 Generación de registro de conversaciones candidatas

El proceso **getCandidateRegistryConversation(CandidateConversationList, CompositionTechnique, RelaxationMode, FinalCompositionType)** de generación de *Conversation*$_{CR}$ presenta varias variantes en función del tipo de composición que se desea realizar. Inicialmente se describirá el proceso a seguir para las *ConversationSelection*, el cual depende de tres de las cuatro dimensiones descritas anteriormente (*CompositionTechnique*, *RelaxationMode* y *FinalCompositionType*). Las cuales tienen que ser procesadas en un determinado orden con el fin de crear correctamente la *Conversation*$_{CR}$ final. El orden correcto comienza con la denominada *CompositionTechnique*, para después continuar con *RelaxationMode* y finalizar con *FinalCompositionType*. Por último, tras describir como crear la *Conversation*$_{CR}$ para *CompositionSelection* se describirá el caso especial de como se construirá el *Conversation*$_{CR}$ para la composición *CapabilitySelection*.

CompositionTechnique

La técnica de composición representa la dimensión más importante ya que es fundamental para la creación de las composiciones equivalentes. En esta dimensión existen dos técnicas, *Integration* e *Interleaving*, las cuales serán descritas a continuación describiendo como se crea el *Automaton*$_{CR}$ en cada uno de los casos, mostrando además como quedaría el conjunto de candidatos de la Ilustración 28 en dicho *Automaton*.

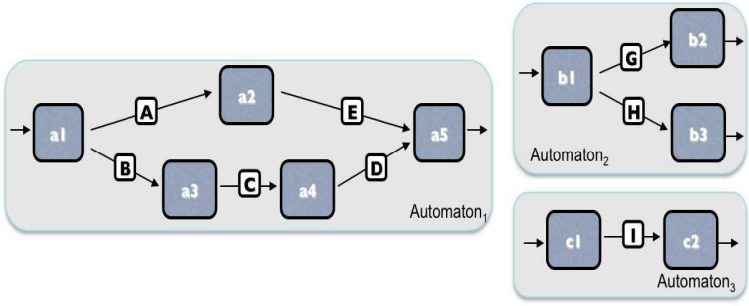


Ilustración 28. Conjunto de elementos *CandidateConversationList*

***Integration*:** Para construir el *Automaton*$_{CR}$ que soporte integración el proceso a seguir consiste en integrar toda y cada una de los *Automaton*$_{cand}$ del repositorio (o bien del conjunto de los *Servic*e con el cual el solicitante desea realizar el emparejamiento) en un único elemento denominado *Automaton*$_{CR}$. Para conseguir este requisito el proceso a seguir consiste en generar dos nuevos estados denominados *State*$_{Init_{CR}}$ que representa el estado inicial y el estado *State*$_{Final_{CR}}$ que representa el estado final. Una vez creados los estados el siguiente paso consiste en asociar una *Transition* vacía (mediante el elemento E que representa una transición vacía) que va desde *State*$_{Init_{CR}}$ hasta *State*$_{0_{cand}}$, donde *can*d representa la *Capabilit*y candidata del repositorio. Y después, asociar una T *ransitio*n vacía que va desde cada uno de los *F*$_{cand}$ hasta el estado final (*State*$_{Final_{CR}}$) del *Automaton*$_{CR}$. Así la formalización del *Automaton*$_{CR}$ =<Q_{CR}, Σ_{CR},δ_{CR}, *State*$_{0_{CR}}$,*F*$_{CR}$ >, donde *Automaton*$_{cand}$ = <Q_{cand}, Σ_{cand},δ_{cand}, *State*$_{0_{cand}}$,*F*$_{cand}$> representa el conjunto de *Automaton*$_{cand}$ del registro queda de la siguiente manera:

- $$Q_{CR}=\left(\bigcup_{cand=1}^{CAND} Q_{cand}\right) \cup \{State_{Init_{CR}}\}$$
- $$\Sigma_{CR}=\left(\bigcup_{cand=1}^{CAND} \Sigma_{cand}\right) \cup \{\epsilon\}$$
- $$\delta_{CR}: \bigcup_{cand=1}^{CAND} (Q_{cand} \times \Sigma_{cand}) \rightarrow \bigcup_{cand=1}^{CAND} Q_{cand}$$
  $$(State, Symbol) \mapsto \delta_{CR}(State, Symbol) = \delta_{cand}(State, Symbol) \ cuando \ (State, Symbol) \in Q_{cand} \times \Sigma_{cand},$$
  $$\delta_{CR}(State, Symbol) = State_{0_{cand}} \ cuando \ State = State_{Init_{CR}} \wedge \ Symbol = \epsilon,$$
  $$\delta_{CR}(State, Symbol) = State_{Final_{CR}} \ cuando \ State \in F_{cand} \wedge \ Symbol = \epsilon$$
- *State*$_{0CR}$ =*State*$_{InitCR}$

- *FCR* ={*State*$_{FinalCR}$}

De esta manera se consigue que todas las *Automaton*$_{cand}$ candidatas (ver Ilustración 28) estén unidas en un único *Automaton*$_{CR}$, como se puede apreciar en la Ilustración 29.

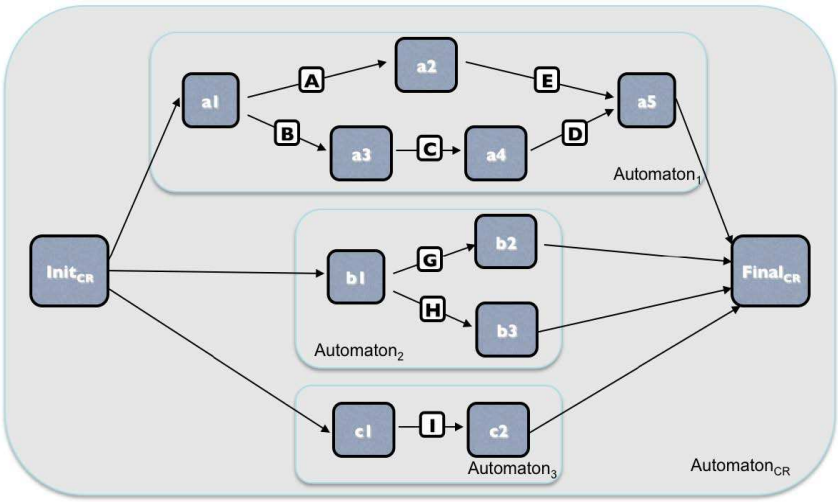


Ilustración 29. Integración de todos los *Automaton* en un único *Automaton*$_{CR}$.

***Interleaving*:** Para construir el *Automaton*$_{CR}$ que soporte el entrelazado de elementos *Automaton*$_{cand}$ se ha de llevar a cabo el denominado *Producto Asíncrono de Automata*[7]. Este proceso, que será descrito a continuación, tiene como base el proceso que genera el denominado Automaton$_{Product}$ que calcula el producto de varios *Automaton*$_{cand}$ el cual es empleado para generar el *Automaton*$_{CR}$ global que soporte entrelazado, es decir, más adelante se describirá cual es la relación entre Automaton$_{Product}$ y *Automaton*$_{CR}$. Por lo tanto Automaton$_{Product}$ se define de la siguiente manera (donde *State*$_{cand}$ = {*st* | *st* ∈ Q_{cand}} y State′$_{cand}$= {st′ | st′ ∈ Q_{cand}}):

- $$Q_{Product} = \prod_{cand=1}^{CAND} Q_{cand}$$
- $$\Sigma_{Product} = \bigcup_{cand=1}^{CAND} \Sigma_{cand}$$

[7] *Asynchronous Product Automaton*

- $\delta_{Product}: \prod_{cand=1}^{CAND} Q_{cand} \times \bigcup_{cand=1}^{CAND} \Sigma_{cand} \rightarrow \prod_{cand=1}^{CAND} Q_{cand}$ $(<State_1, State_2, \ldots, State_{CAND}>, Symbol) \rightarrow \delta_{Product}(<State_1, State_2, \ldots, State_{CAND}>, Symbol) = <State'_1, State'_2, \ldots, State'_{CAND}>$ $si \ \exists \ cand1, cand2 \in \{1 \ldots CAND\}$ donde: $State'_{cand2} = State_{cand2} \ \forall \ cand2 \neq cand1 \ \wedge \ \delta_{cand1}(State_{cand1}, Symbol) = State'_{cand1}$

- $State_{0Product} =< State_{01}, State_{02}, \ldots, State_{0CAND} >$

- $F_{Product} =\{< State_1, State_2, \ldots, State_{CAND} > \in Q_{Product}\} : State_1 \in F_1 \wedge State_2 \in F_2 \wedge \ldots \wedge State_{CAND} \in F_{CAND}$

Sin embargo, aplicando el proceso *Interleavin*g a todos los $Automaton_{cand}$ en conjunto y definiendo que $Automaton_{CR} = Automaton_{Product}$, el $Automaton_{CR}$ resultante representa un *Automato*n en el que solo es posible llegar desde el estado inicial a uno final invocando todos y cada uno de los $Automaton_{cand}$ candidatos. Esto representa que si existen únicamente dos *Automato*n candidatos, por ejemplo el $Automaton_1$ y el $Automaton_3$ de la Ilustración 28, el *AutomatonProduct* generado (ver Ilustración 30) solo ofrecerá matching positivo para aquellos $Automaton_{Req}$ que realicen interleaving de esos dos *Automaton*. Es decir, no existirá matching en el caso en el que se solicite exclusivamente el $Automaton_1$ o el $Automaton_3$, sino exclusivamente en el caso en el que se solicite un *Automato*n que los entrelace. Por lo tanto, no es posible aplicar que $Automaton_{CR} = Automaton_{Product}$.

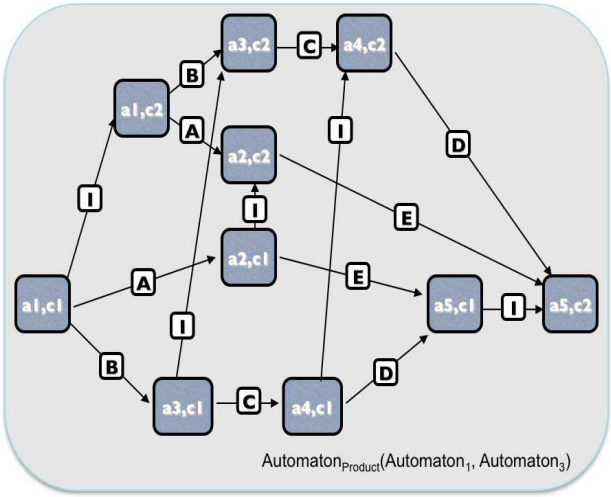


Ilustración 30. Entrelazado de$Automaton_1$ y $Automaton_3$.

Por ello, el proceso de *Interleavin*g y por consiguiente el $Automaton_{CR}$ generado tiene que dar soporte desde el entrelazado entre dos $Automaton_{cand}$ hasta el entralazado del

conjunto de candidatos, teniendo en cuenta en cada uno de los casos todas y cada una de las posibles combinaciones. De esta manera, para conseguir lo anteriormente expresado se proponen dos soluciones:

- La primera técnica consiste en generar tantos elementos $Automaton_{Product}$ como posibles combinaciones de $Automaton_{cand}$ desde un único elemento hasta el conjunto total de candidatos. Por ejemplo, en caso de que existan cuatro candidatos habría que generar un total de 15 $Automaton_{Product}$, 4 (1,2,3,4) para las combinaciones de un elemento, 6 (1-2,1-3,1-4,2-3,2-4,34) para las combinaciones de 2 elementos, 4 (1-2-3,1-3-4,1-3-4,2-3-4) para las combinaciones de 3 elementos y uno para las combinaciones de 4 elementos (1-2-3-4). Generalizando la expresión al número de candidatos *CAND*, el número de combinaciones total de $Automaton_{Product}$ será el siguiente:

Ecuación 7

$$InterleavingCombinations = \sum_{cand=1}^{CAND} \binom{CAND}{cand}$$

- El segunda técnica consiste en emplear un único $Automaton_{Product}$ que entrelaza todos y cada uno de los $Automaton_{cand}$ candidatos y posteriormente realizar ciertas adaptaciones. Estas adaptaciones consistirán en modificar varios estados de $Automaton_{Product}$ para así convertirlos en estados finales. Para que un estado pueda ser convertido al menos uno de los subestados que lo componen tiene que ser final (en el $Automaton_{cand}$ correspondiente) y el resto de sus subestados, estados iniciales (en sus respectivos $Automaton_{cand}$), es decir: $F_{Product} = \{< State_1, State_2, \ldots, State_{CAND} > \in Q_{Product}\} : \forall\ State_{cand} = F_{cand} \vee State_{0cand}$. Posteriormente una vez realizadas dichas adaptaciones $Automaton_{CR} = Automaton_{Product}$.

Tanto en la primera como en la segunda técnica los DataFlow y *ContextFlow* seguirán siendo los mismos, es decir, no variará el *Symbol* de origen y destino de cada uno de ellos, pese a que existan nuevos estados. En ambos casos, tras el proceso de entrelazado final, el siguiente paso consiste en aplicar el proceso *Integrati*on anteriormente definido, para así combinar en un único *Automato*n cada uno de los $Automaton_{Product}$ generados. Para ello cada $Automaton_{ProductinterleavingCombinations}$ será tratado como un $Automaton_{cand}$ para así ser integrado, donde 1 ≤

interleavingCombinations ≤ *InterleavingCombinations*. En el segundo caso como solo existirá un único *Automaton*P *roduct* se establece que $Automaton_{CR} = Automaton_{Product}$. Para el presente caso se ha decidido emplear el segundo enfoque ya que solo es necesario crear un único *Automato*n siendo necesario un proceso extra para anotar aquellos nuevos estados que pasan a ser finales. Sin embargo, en el primer caso, hay que construir un número muy elevado (ver Ecuación 7) de $Automaton_{ProductinterleavingCombinations}$ el cual supone un coste computacional muy elevado en comparación con el proceso seleccionado, como puede apreciarse en la Ilustración 31 donde el eje horizontal representa el número de elementos $Conversation_{cand}$ y el eje vertical representa el tiempo (en nanosegundos) necesario para crear el entrelazado de los *Automaton*. En la figura se puede ver que el rendimiento de las técnicas es similar para los casos en los que el número de elementos donde *CAN*D es bajo, pero a medida que este aumenta la diferencia de rendimiento es sustancial llegando a ser del 50% para *CAN*D =6.

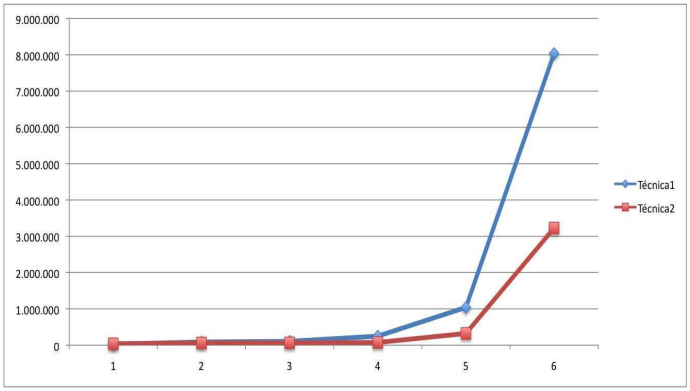


Ilustración 31. Rendimiento en las dos técnicas de entrelazado

RelaxationMode

En caso de que el tipo de composición sea *ExactCompositio*n no será necesario realizar ningún tipo de adaptación en $Automaton_{CR}$, sin embargo, en caso de que sea *FlexibleComposition* serán necesarias las adaptaciones descritas a continuación, quedando el $Automaton_{CR}$ final de la siguiente manera[8] a (ver Ilustración 32):

[8] El ejemplo empleado en la Ilustración 32 está basado en un conjunto de $Automaton_{cand}$ que ha sufrido un proceso previo de *Integration*, sin embargo, en caso de que dicho conjunto hubiera sufrido un proceso de *Interleaving* el proceso de

- Para todos los estados que no sean el inicial o el final del *Automaton*$_{CR}$ (*State*0_{CR} o F_{CR})y no sean el inicial o el final de cada una de los *Automaton*$_{cand}$ (*State*0_{cand} o F_{cand}) se creará una transición vacía desde *State*0_{CR} a dicho estado, si es que no existe previamente y se creará una transición vacía desde dicho estado a F_{CR}, si es que no existe previamente:

Ecuación 8

$$\delta_{CR} = \delta_{CR} \cup < State_{0_{CR}}, \epsilon, State > \cup < State, \epsilon, F_{CR} >,$$
$$\forall\ State \in Q_{CR} - \{\{State_{0_{cand}}\}_{cand=1}^{CAND}, \{F_{cand}\}_{cand=1}^{CAND}, State_{0_{CR}}, F_{CR}\}$$

- Para cada uno de los *Automaton*$_{cand}$ y más concretamente para cada una de las ramas de ejecución[9] de dicho *Automaton*$_{cand}$, se creará una transición vacía desde cada uno de los estados (*State*) a todos los estados (*State'*) que vienen a continuación de éste, y así sucesivamente hasta llegar a F_{cand}, siempre que dicha transición no exista previamente. Quedan excluidos del proceso el estado inicial y final del *Automaton*$_{CR}$ (*State*0_{CR} y F_{CR}) y los estados iniciales y finales de cada uno de los *Automaton*$_{cand}$ (*State*0_{cand} y F_{cand}).

---

transformación del *Automaton*$_{CR}$ sería el mismo. Es decir, tanto si previamente se ha realizado el proceso de *Interleaving* como el de *Integration* el proceso posterior relacionado con *RelaxationMode* es el mismo en ambos casos.

[9] Se denomina rama de ejecución a cada uno de los posibles caminos del *Automaton* que comienzan en el estado inicial del *Automaton* y llegan hasta el estado final del *Automaton*. A su vez, una rama de ejecución puede estar compuesta por varios uno o varios bloques ejecución. Cada bloque de ejecución vienen definido por el intervalo donde el comienzo de éste está definido mediante el estado inicial del *Automaton* (en caso de que se trate de del bloque inicial del *Automaton*) o un estado al que llega una *Transition* representada mediante una *Condition*. El final del bloque está definido bien por el estado final del *Automaton*, bien por un estado al que llegan varias *Transition* vacías o por un estado del cual salen nuevas Transition representadas mediante elementos *Condition*. Como puede apreciarse en la Ilustración 36 se puede apreciar como el *AutomatonReq* mostrado en Ilustración 34 tiene cinco bloques de los cuales derivan 3 ramas de ejecución.

Ecuación 9

$$\delta_{CR} = \delta_{CR} \cup < State, \epsilon, State' >$$
$$\forall\ State, State' \in Q_{CR} - \{\{State_{0_{cand}}\}_{cand=1}^{CAND}, \{F_{cand}\}_{cand=1}^{CAND}, State_{0_{CR}}, F_{CR}\}$$
$$: State \neq State' \wedge\ sameBranch(State, State') \wedge\ preState(State, State')$$

Donde la función *sameBranch*(*State*, *State'*) representa que ambos estados están en la misma rama de ejecución y *preState*(*State*, *State'*) representa que *State* se encuentra en un estado previo a *State'* en la rama de ejecución.

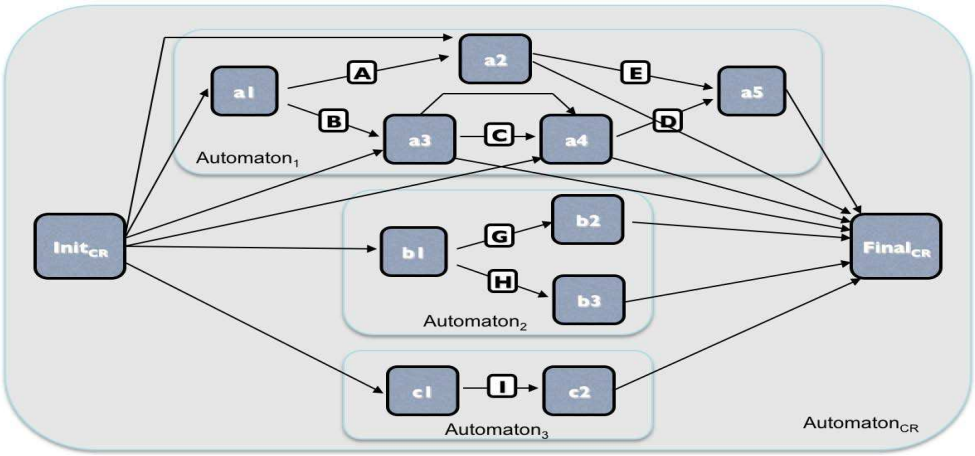


Ilustración 32. *Automaton*$_{CR}$ obtenido tras adaptación para soportar *FlexibleComposition*.

Gracias a estas adaptaciones el *Automaton*$_{CR}$ ofrece soporte para realizar un tipo de composición más relajado en el que no es necesario que todas las *Conversation*$_{cand}$ que forman parte de la *Conversation*$_{res}$ tengan que tener un equivalente en *Conversation*$_{Req}$. El hecho anteriormente descrito puede hacer que el *Conversation*$_{res}$ resultante produzca más elementos *ReturnValue* y *Effect* que los deseados y necesite de más elementos *Parameter* y *Precondition* que los solicitados. Sin embargo, la *Conversation*$_{res}$ en su conjunto tiene que ser equivalente a *Conversation*$_{Req}$ siendo necesario un proceso extra que evalúe si las *Conversation*$_{res}$ calculadas en el proceso de composición son compatibles con *Conversation*$_{Req}$. Este proceso es llevado a cabo una vez calculados todos los *Conversation*$_{res}$ equivalentes, es decir dentro del mecanismo ***conversationMatch(AdaptedReqConversationList*, *Conversation*$_{CR}$)** y más concretamente dentro del subproceso ***resultChecking(Conversation*$_{ada}$, *Conversation*$_{toCheck}$)** descrito en la sección 3.3.4.

FinalCompositionType

En caso de que *FinalCompositionType* = *OneConversationResult* el $Automaton_{CR}$ no sufrirá cambio alguno, pero en caso de que *FinalCompositionType* = *ManyConversationResul*t el $Automaton_{CR}$ tiene que ser modificado para que de soporte a la combinación de varias *Conversation*. Este cambio es un cambio muy simple, ya que el cambio que hay que hacer es definir que el estado final F_{CR} será equivalente al $State0_{CR}$, es decir $F_{CR} = \{State0_{CR}\}$.

CapabilitySelection

En caso de que el tipo de composición sea de tipo *CapabilitySelectio*n se aplicarán los procesos anteriormente descritos (empleando la siguiente combinación de dimensiones: <*ManyConversationResult*, *ExactComposition*, *Integration*, *NonAdaptiveRequest*>) para la generación del $Conversation_{CR}$, sin embargo como base se empleará la estructura $Registry_{OSC}$, la cual tras el proceso **getCandidateRegistryConversation(CandidateConversationList, CompositionTechnique, RelaxationMode, FinalCompositionType)** dará como resultado el $Automaton_{CR}$ equivalente al de la Ilustración 33, tomando como base las *SimpleCapabilit*y que componen las *Conversatio*n de la Ilustración 28.

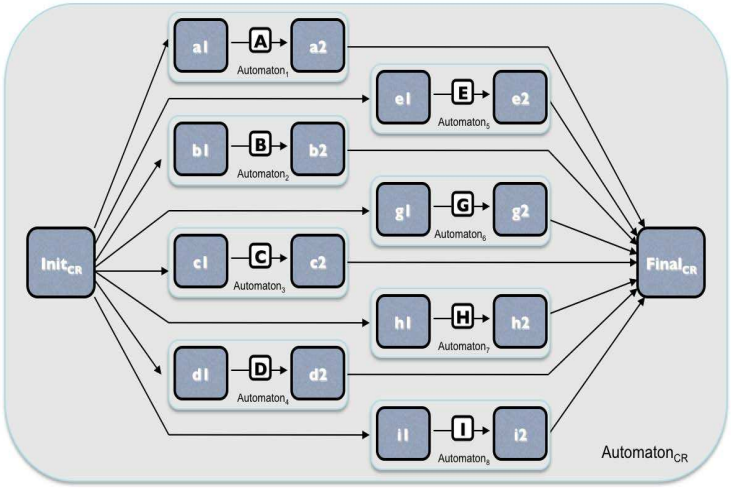


Ilustración 33. $Automaton_{CR}$ para CapabilitySelection

3.3.3 Adaptación de la conversación solicitada

Este proceso está ligado a la dimensión *AdaptivityMod*e el cual puede tener dos valores posibles, *NonAdaptiveReques*t y *AdaptiveRequest*. En caso de *AdaptivityMod*e = *NonAdaptiveReques*t el proceso ***getAdaptations(Conversation$_{Req}$, AdaptivityMode)*** no realizará transformación alguna en *Conversation$_{Req}$*, por lo tanto *AdaptedReqConversationLis*t = {*Conversation$_{Req}$*} en caso contrario se llevará a cabo el proceso ***calculateAdaptations(Conversation$_{Req}$)*** de análisis y adaptación de *Conversation$_{Req}$* con el fin de determinar otras posibles combinaciones que tengan el mismo comportamiento que el solicitado.

Ecuación 10

$$AdaptedReqConversationList = \begin{cases} Conversation_{Req} & (a) \\ calculateAdaptations(Conversation_{Req}) & (b) \end{cases}$$

(a) $NonAdaptiveRequest$

(b) $AdaptiveRequest$

El proceso ***calculateAdaptations(Conversation$_{Req}$)*** tiene como objetivo crear diferentes variantes de *Conversation$_{Req}$* que sean equivalentes a esta para así ampliar el espectro de posibles combinaciones y tener una mayor probabilidad de encontrar emparejamientos positivos [10]. Para ello el proceso tiene que cumplir con los siguientes requisitos:

- Las *Capability* de *ConversationRe*q que no están relacionadas con ningún *ContexFlow* ni *DataFlow* pueden ser movidas a lo largo de la estructura del *Automato*n y pueden ser situadas en cualquier lugar ya que no tienen relación con ninguna otra *Capability*, por lo tanto no tienen dependencias, ni nadie depende de ellas. Sin embargo, existen ciertas limitaciones a la afirmación anterior, ya que las *Capabilit*y podrán ser movidas dentro del *Automato*n

[10] El sistema no calcula todas las combinaciones posibles para el AutomatonReq solicitado, sino que ofrece una heurística que calcula un conjunto considerado como suficiente para conseguir un emparejamiento válido. Esta decisión ha sido tomada debido a que el proceso completo de cálculo de combinaciones supone un proceso muy costoso en términos computacionales, siendo la heurística empleada una aproximación mucho más eficiente.

siempre y cuando sean movidas dentro del bloque de ejecución en el que se encuentran. De esta manera se respetan las restricciones definidas en la *Conversation*$_{Req}$ gracias a las *Condition* asociadas a las *Transition*, es decir que la *Capability* es movida dentro de los rangos que han sido establecidos. Como se puede apreciar en la Ilustración 34 la *Capability D* (que no tiene *Flow* asociado) puede ser movida a lo largo del *Automaton*, sin embargo, como se encuentra dentro de una bloque de ejecución concreto solo puede moverse dentro de dicho bloque de ejecución. Así la *Capability* D solo puede ser movida para situarse antes que *C* o tras *E*.

- Aquellas *Capability* que forman parte al menos de un *DataFlow* o de un *ContexFlow* tendrán las siguientes características en función de si son productoras o consumidoras de dicho *Flow*:

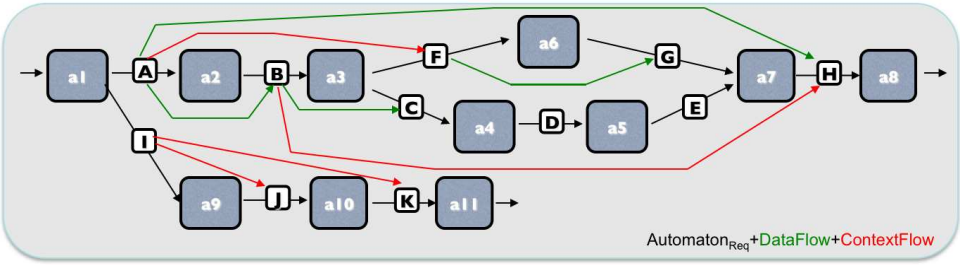


Ilustración 34. Automaton$_{Req}$ a adaptar.

  - En caso de que la *Capability* sea el origen del *Flow* (mediante *ReturnValue* o *Effect*), ésta podrá ser movida a lo largo del *Automaton* y podrá estar en cualquier posición siempre y cuando no sea después de la *Capability* que consume el *Flow* creado por éste.

  - En caso de que la *Capability* sea la consumidora del *Flow* creado (mediante *Parameter*, *PreCondition*, *IndirectCondition* o *DirectCondition*), ésta podrá ser movida a lo largo del *Automaton* y podrá estar en cualquier posición siempre y cuando no esté antes de la *Capability* que produce el *Flow* consumido por ésta.

Además de las restricciones específicas que tiene cada una de las dos variantes descritas anteriormente, también se aplican a estos casos las restricciones definidas para las *Capability* que no están relacionados con ningún *ContextFlow* ni *DataFlow*.

- Los bloques de ejecución no pueden ser movidos a lo largo del *Automaton*, ya que éstos han sido definidos por el usuario para representar restricciones (mediante las *Condition*) en la ejecución del *Automaton*.

De esta manera, el proceso que sigue la función ***calculateAdaptations(Conversation$_{Req}$)*** se constituye de los siguientes pasos:

a) Calcular el grafo *FlowDependencyGraph$_{Req}$*, que representa un grafo en el que se muestra el flujo global del *Conversation$_{Req}$*, el cual incluye sólo aquellas *Capability* que bien son consumidoras o generadoras del *Flow* (*DataFlow* y *ContextFlow*). Esta estructura es la que define las restricciones relativas a las *Capability* que forman parte de algún *Flow* y todas las variantes de *Conversation$_{Req}$* creadas tendrán que respetar esta estructura (ver Ilustración 35).

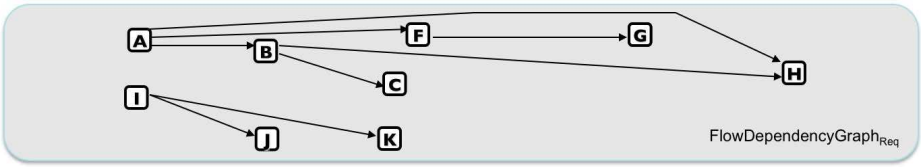


Ilustración 35. AutomatonReq junto con el grafo FlowDependencyGraph.

b) Calcular el grafo *BlockDependencyGraph$_{Req}$* que representa un grafo en el que se muestran agrupadas las *Capability* que forman el *Automaton$_{Req}$* en función al bloque de ejecución al que pertenecen, de esta manera se establecen las restricciones relativas a los diferentes bloques de ejecución definidas, en el cual todas las variantes de *Conversation$_{Req}$* creadas tendrán que respetar esta esctructura (ver Ilustración 36).

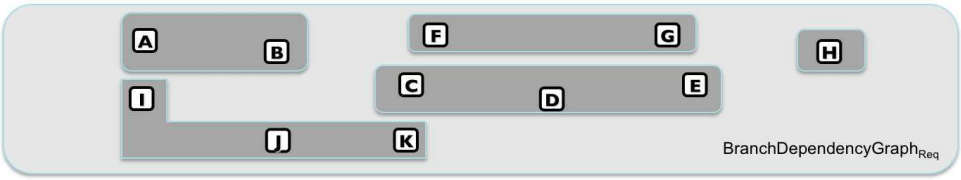


Ilustración 36. Grafo BlockDependency Graph

c) Una vez se tienen los dos grafos anteriores, el siguiente proceso consiste en integrarlos en único grafo (*IntegratedDependencyGraph$_{Req}$*) (ver Ilustración 37),

el cual ofrece una visión global de los dos tipos de restricciones existentes: las relativas al *Flow* y los relativos a los bloques de ejecución.

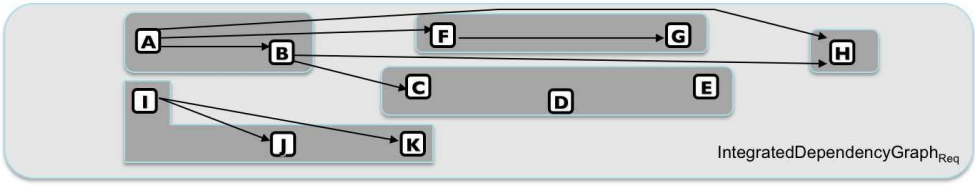


Ilustración 37. *IntegratedDependencyGraph*.

d) El siguiente paso consiste en emplear como base el grafo *IntegratedDependencyGraph*$_{Req}$ creado anteriormente y eliminar aquellos *Flow* que no tienen inicio y fin en el mismo bloque de ejecución, ya que el hecho de que estén separados en bloques de ejecución reduce el propio ámbito de posibles movimientos. Esto se realiza debido a que la propia separación entre bloques de ejecución es más restrictivo que el *Flow*. De esta manera aquellas *Capabilit*y que previamente tenían al menos un *Flow* que tuviera fin en un bloque de ejecución diferente al de comienzo del *Flow*, pasan a ser elementos *Capabilit*y que pueden ser movidos a cualquier lugar del bloque de ejecución en la que se encuentran, a menos que tengan al menos otro *Flow* que tenga inicio y fin en el mismo bloque de ejecución, en dicho caso se aplicarían las restricciones definidas para los *Flow*. Por ejemplo, en el caso de la Ilustración 37 se puede apreciar que las *Capabilit*y A y B además del *Flow* entre ellas, también tiene *Flow* con las *Capabilit*y F y C respectivamente. Sin embargo, estos dos *Flow* tienen inicio y fin en diferentes bloques de ejecución, por lo tanto, tienen que ser eliminados, ya que el ámbito de aplicación de los F *lo*w se restringe al propio bloque de ejecución de inicio y fin (lo mismo ocurre con la relación entre las *Capabilit*y A y B respecto a *H*). Esta serie de adaptaciones, dan lugar al denominado *FinalDependencyGraph*$_{Req}$ (ver Ilustración 38).

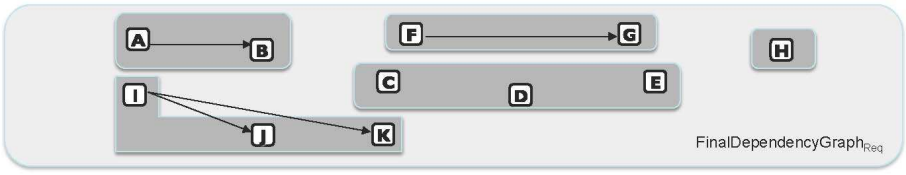


Ilustración 38. FinalDependencyGraph.

e) El paso final consiste en crear las diferentes combinaciones de subautómatas

para cada uno de los bloques de ejecución (ver Ilustración 39). En el presente caso los únicos bloques de ejecución que soportan combinaciones son la de I − J − K que da lugar a dos posibles combinaciones (I − J − K y I − K − J) y la de C − D − E que da lugar a 6 posibles variantes (C − D − E, C − E − D, D − C − E, D − E − C, E − C − D y E − D − C).

Para después combinar las diferentes versiones de las bloques de ejecución con el fin de generar todas las posibles variantes de $Conversation_{Req}$, las cuales darán lugar al conjunto *AdaptedReqConversationLis*t (ver Ilustración 40).

3.3.4 Composición de conversaciones

Una vez calculados tanto $Conversation_{CR}$ así como *AdaptedReqConversationLis*t el siguiente paso consiste en la invocación del algoritmo de emparejamiento ***conversationMatch(AdaptedReqConversationList*, *$Conversation_{CR}$)***, descrito en la Tabla 5, el cual tiene dos funciones principales:

- Por cada uno de los $Conversation_{ada}$ se lleva a cabo el proceso ***getMatchingCompositions(*$Conversation_{ada}$*, *$Conversation_{CR}$)***, que emparejara resultado el conjunto de elementos *Conversation* que empareja con dicha $Conversation_{ada}$, donde **getMatchingCompositions(*$Conversation_{ada}$*, *$Conversation_{CR}$*)** = *AdaConversationCompatibleList* = $\{Conversation_{adacalc}\}^{CALC}_{calc=1}$ y *CALC* representa el número total de elementos Conversation que emparejan con un $Conversation_{ada}$ en concreto. Después, como dichos elementos son válidos son añadidos a la lista *ResultConversationList*, como: $\{Conversation_{res}\}^{RES}_{res=1}$ = $\{\{Conversation_{adacalc}\}^{CALC}_{calc=1}\}^{ADA}_{ada=1}$, siendo *RES* la suma de cada uno de los *CALC* de cada $Conversation_{ada}$. De esta manera se tiene completado parte del elemento *ResultConversationList*, el cual es completado en su totalidad en la siguiente fase.

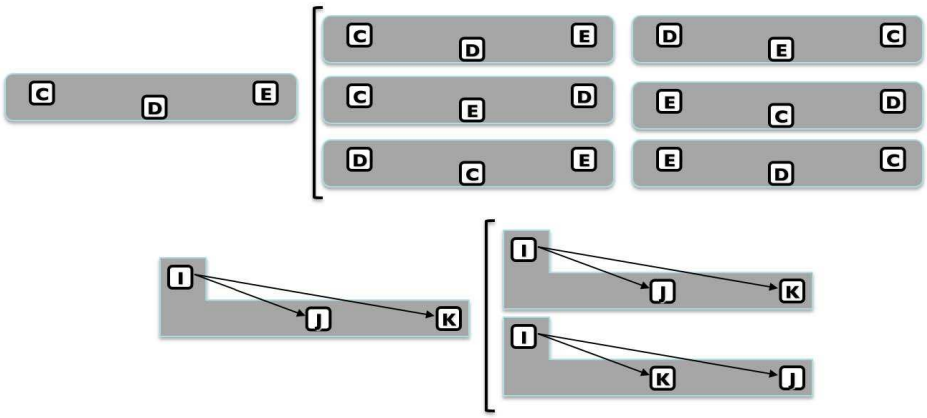


Ilustración 39. Combinaciones posibles de los bloques de ejecución *Conversation*$_{Req}$

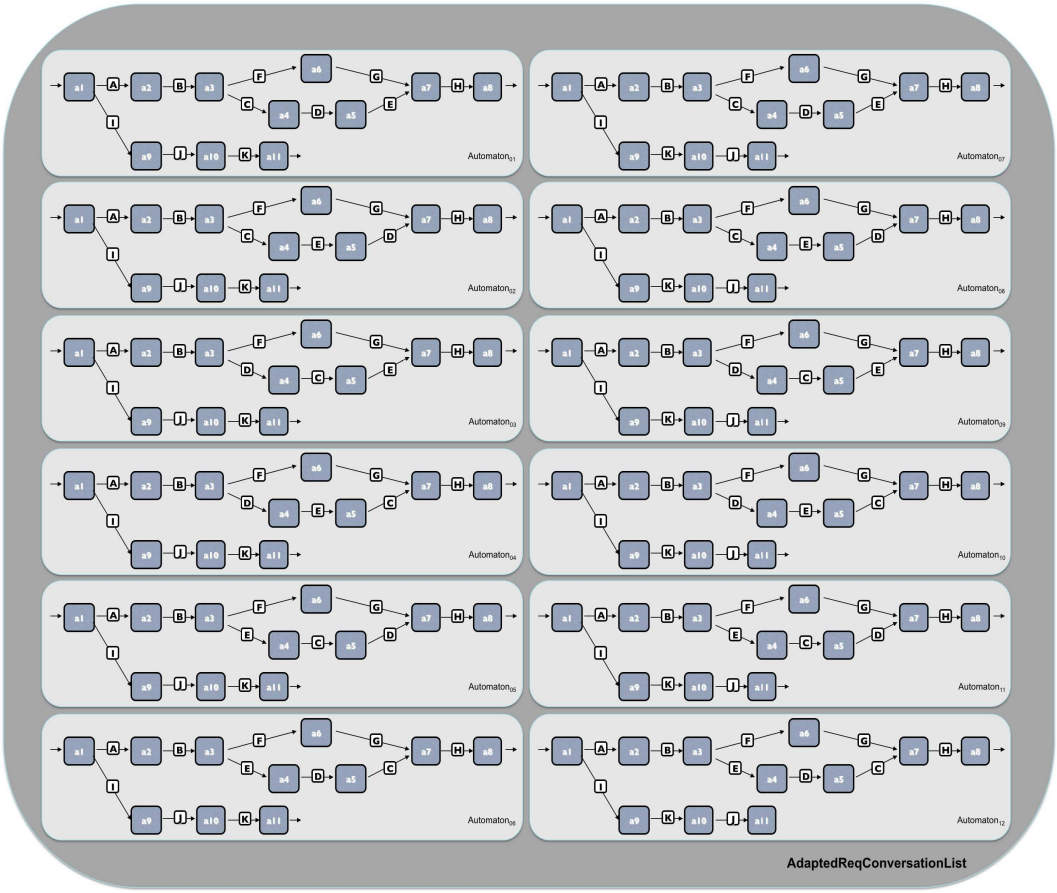


Ilustración 40. Conjunto de *Conversation*$_{Req}$ equivalentes

- Una vez que se disponga del conjunto *ResultConversationList*, el sistema calcula la distancia entre *Conversation*$_{Req}$ y cada uno de los *Conversation*$_{res}$ mediante la función ***calculateConversationDistance(Conversation*$_{Req}$*, Conversation*$_{res}$*)*** representada en la Ecuación 12 finalizando de esta manera el

proceso de composición.

De esta manera, tras aplicar los dos procesos anteriores, el proceso global devolverá el conjunto *ResultConversationLis*t que está compuesto por una lista en la que cada uno de los elementos del mismo está formado por una *Conversation* y el *MatchResult* correspondiente, donde el número total de elementos estará formado por el conjunto de resultados derivados de cada uno de los $Conversation_{ada}$, siendo *RES* el número total de resultados positivos.

Tabla 5. Algoritmo conversationMatch

```
Input: List<Conversation>adaptedReqConversationList, Conversation conversationCR
List<MatchResult>resultConversationList = new <Hashtable<MatchResult>();
Conversation conversationReq = adaptedReqConversationList.getFirst();
foreach conversationAda in adaptedReqConversationList do
        resultConversationList.addAll(getMatchingCompositions(conversationAda, conversationCR));
end
foreach conversationRes in resultConversationList do
        conversationRes.hasMatchValue(calculateConversationDistance(conversationReq, conversationRes));
end
Output: List<MatchResult>
return resultConversationList;
```

getMatchingCompositions(Conversation$_{ada}$, Conversation$_{CR}$)

Una vez descrito como es de manera general el proceso de composición, el siguiente paso consiste describir como es el proceso ***getMatchingCompositions(Conversation$_{ada}$, Conversation$_{CR}$)*** (ver Tabla 6), el cual invoca el proceso ***automatonMatch(State0_{ada}, State0_{CR}, TFunctionList1, TFunctionList2)*** que extrae el un conjunto de ramas de ejecución donde cada una de las ramas describe que conjunto de transiciones componen la *Conversatio*n que empareja con $Conversation_{ada}$, del cual posteriormente se extraen las *Conversatio*n equivalentes (mediante la función ***extractConversationsFromSavedPaths()***), la cuales son añadidas al elemento *ConversationLis*t (que es una lista de elementos *Conversation*). Tras ello cada elemento de *ConversationLis*t es validado mediante la función ***resultChecking(Conversation$_{ada}$, Conversation$_{toCheck}$)*** (descrita más adelante), donde $Conversation_{toCheck}$ representa a una de las *Conversatio*n de *ConversationLis*t y que tiene que ser validado antes de pasar a ser un $Conversation_{ada_{calc}}$ y ser devuelto en la lista *AdaConversationCompatibleList*.

Tabla 6. Algorimo getMatchingCompositions.

```
Input: Conversation conversationAda, Conversation conversationCR
List<Conversation>conversationList = new LinkedList<Conversation>();
reqConv = conversationAda; //used in the automatonMatch method
regConv = conversationCR; //used in the automatonMatch method
if automatonMatch(conversationAda.getAutomaton().getInitialState(), conversationCR.getAutomaton().getInitialState(),
new LinkedList<TransitionFunction>(), new LinkedList<TransitionFunction>()) then
        conversationList = extractConversationsFromSavedPaths();
end
List<MatchResult>AdaConversationCompatibleList = new LinkedList<MatchResult>();
foreach conversationT oCheck in conversationList do
        if resultChecking(conversationAda, conversationT oCheck) then
                AdaConversationCompatibleList.add(new MatchResult(conversationToCheck, 1);
        end
end
Output: List<MatchResult>
return AdaConversationCompatibleList;
```

automatonMatch($State0_{ada}$, $State0_{CR}$, *TFunctionList1*, *TFunctionList2*) es la función que representa la base del proceso de composición, el cual como premisa se basa en el cálculo de subautómatas de $Conversation_{CR}$ que emparejancorrectamente con $Conversation_{ada}$. Para ello son integrados y/o entrelazados varios de los *Automaton* que componen $Automaton_{CR}$ paraasíconseguir el conjunto de $Conversation_{ada_{calc}}$ completo y válido. De esta manera manera $Automaton_{ada_{calc}} \subseteq Automaton_{CR}$, es decir $\boldsymbol{subAutomaton(Automaton_{ada_{calc}}, Automaton_{CR})}$. Donde la relación de subautómatas se define de la siguiente manera, siendo $Automaton_{ada_{calc}} = <Q_{ada_{calc}}, \Sigma_{ada_{calc}}, State0_{ada_{calc}}, \delta_{ada_{calc}}, F_{ada_{calc}}>$ y $Automaton_{CR} = < Q_{CR}, \Sigma_{CR}, State0_{CR}, \delta_{CR}, F_{CR} >$:

- $Q_{ada_{calc}} \subseteq Q_{CR}$
- $\Sigma_{ada_{calc}} \subseteq \Sigma_{CR}$
- $\delta_{ada_{calc}} : Q_{ada_{calc}} \times \Sigma_{ada_{calc}} \longrightarrow Q_{ada_{calc}}$

  $(State, Symbol) \longrightarrow \delta_{ada_{calc}}(State, Symbol) = \delta_{CR}(State, Symbol)$
- $\forall State \in Q_{ada_{calc}}, \forall Symbol \in \Sigma_{ada_{calc}} : \delta_{ada_{calc}} = \varnothing \Rightarrow State \in F_{ada_{calc}}$
- $State0_{ada_{calc}} = State0_{CR}$
- $F_{ada_{calc}} \subseteq \Sigma_{CR}$

Para conseguir lo anterior, inicialmente el proceso parseará cada uno de los estados del autómata *Automaton*$_{ada}$ comenzando desde el *InitialStat*e (*State*0_{ada}) continuando con las transiciones de dicho automata. Simultaneamente, realizará el parseo del *Automaton*$_{CR}$ con el fin de encontrar un estado equivalente al estado que se está parseando en *Automaton*$_{ada}$ a cada paso del proceso de parseo. La equivalencia entre un estado de *Automaton*$_{ada}$ y *Automaton*$_{CR}$ se da cuando para cada elemento *Symbo*l de entrada del estado de *Automaton*$_{ada}$ existe al menos un elemento *Symbo*l semánticamente equivalente en *Automaton*$_{CR}$. Cada uno de los estados de *Automaton*$_{ada}$ es parseado una única vez. Todo este proceso ha sido plasmado en un algoritmo (ver Tabla 7) que calcula aquellos subautómatas de *Conversation*$_{CR}$ que se comportan como el autómata de *Conversation*$_{ada}$, siendo cada una de las ramas de ejecución devueltas como referencia. Este algoritmo hace uso de las variables globales *reqCon*v y *regCon*v definidas en la Tabla 6.

Tabla 7. Algoritmo automatonMatch

```
Input: State reqState, State regState, LinkedList<TransitionFunction> regPath, LinkedList<TransitionFunction>
reqPath
Output: boolean
LinkedList<TransitionFunction> tfStartingInReqState = getTransitionsStartingFrom(reqConv, reqState);
LinkedList<TransitionFunction> tfStartingInRegState = getValidTransitionsStartingFrom(regConv, regState);
if (reqState isFinal in reqConv) and (regState isFinal in regConv) and tfStartingInReqState isEmpty then
        savePathMatch(reqP ath, regP ath);
        return true;
end
if (reqState isFinal in reqConv) and (regState isNotFinal in regConv) then
        return false;
end
if (tfStartingInRegState.getFirst().getFinalState() equalsNot regConv.getAutomaton.getRegFinalState()) and
tfStartingInRegState containsAllNot tfStartingInReqState then
        return false;
end
boolean match = false;
foreach currentReqT F in tfStartingInReqState do
        reqPath.add(currentReqTF );
        foreach currentRegT F in tfStartingInRegState do
                regPath.add(currentRegT F );
                if currentRegT F isInitial in regConversation then
                        foreach nextRegT F in getValidTransitionsStartingFrom(regConv,
currentRegTF.getFinalState()) do
                                regPath.add(nextRegTF);
                                if automatonMatch(currentReqTF.getFinalState(), nextRegTF.getFinalState(),
regPath, reqPath) then
                                        match = true;
```

```
                                    end
                                    regPath.remove(nextRegT F );
                            end
                    end
                    else
                            if transitionFunctionMatch(currentReqTF,currentRegTF)=_0 then
                                    if automatonMatch(currentReqTF.getFinalState(), nextRegTF.getFinalState(),
regPath, reqPath) then
                                            match = true;
                                    end
                            end
                    end
                    regPath.remove(currentRegTF);
            end
            reqPath.remove(currentReqTF);
end
return match;
```

resultChecking(*Conversation*$_{ada}$, *Conversation*$_{toCheck}$) es llevada a cabo por cada una de las *Conversation* (denominada *Conversation*$_{toCheck}$) equivalente a *Conversation*$_{ada}$ (ver Tabla 7) de la lista *conversationList* y toma en consideración la tipología de *MatchingType* que se ha solicitado al realizar el *DiscoveryRequest*[11]. Este proceso consiste en verificar si las *ConversationtoCheck* equivalentes obtenidas son válidas respecto a la signatura y la especificación general de *Conversation*$_{ada}$, siendo almacenada como elemento *Conversation*$_{ada_{calc}}$ en caso de que dicha *Conversation* seaválida.Sinembargo,esteprocesonohayquellevarloacabosiempre,sinoque tiene que ser aplicado únicamente cuando la dimensión *RelaxationMode* = *FlexibleComposition* y además en aquellos casos en los que el *Conversation*$_{toCheck}$ tenga más elementos *Transition* que los solicitados, en caso contrario no se realiza la validación ya que se considera que la *Conversation*$_{toCheck}$ es correcta (es decir la *Conversation*$_{toCheck}$ es considerada valida y por lo tanto pasa a ser una *Conversation*$_{ada_{calc}}$). Con tal fin se ha

[11] El mecanismo *Conversation*$_{Match}$ es empleado tanto para *PEConversation*$_{Match}$, *IOConversation*$_{Match}$ e *IOPEConversation*$_{Match}$, es decir, en todos los casos los procesos planteados serán los mismos, sin embargo existirán dos partes del proceso: ***candidateSelection(Registry, Conversation*$_{Req}$*, FinalCompositionType, RelaxationMode)*** y ***resultChecking(Conversation*$_{ada}$*, Conversation*$_{toCheck}$*)***, en donde se empleará el algoritmo de matching correspondiente (SimpleMethod$_{Match}$, SimpleCapability$_{Match}$, o SimpleCapability$_{Match}$+SimpleMethodMatch para IOPEConversation$_{Match}$).

definido la función denominada ***resultChecking(Conversation$_{ada}$, Conversation$_{toCheck}$)***, la cual tiene dos subprocesos descritos a continuación y que se aplican en función del tipo de *DiscoveryReques*t realizado:

IOResultChecking(Conversation$_{ada}$, Conversation$_{toCheck}$): Este es el proceso inicial que evalúa si la *Conversation$_{toCheck}$* resultante requiere más elementos *Parameter* que los requeridos y/o si ofrece más elementos *ReturnValue* que los requeridos:

- Respecto a los elementos *ReturnValue*, el hecho de que la *Conversation$_{toCheck}$* ofrezcamás elementos *ReturnValue* no afecta en ningún caso al resultado final, ya que al menos aquellos que han sido solicitados están en *Conversation$_{toCheck}$* (tal y como es validado en el proceso ***candidateSelection(Registry, Conversation$_{Req}$, FinalCompositionType, RelaxationMode)*** y en ***conversationMatch(AdaptedReqConversationList, Conversation$_{CR}$)***), por lo tanto la *Conversation$_{toCheck}$* puede ser invocada ya que se obtienen al menos aquellos *ReturnValue* que son solicitados, siendo el resto no considerados. Por lo tanto, no es necesario realizar ninguna comprobación para los elementos *ReturnValue*.

En cambio, en el caso de los elementos *Parameter* el hecho de que existan más elementos *Parameter* que los solicitados significa que para invocar correctamente dicha *Conversation$_{toCheck}$* es necesario proveer todos los P *arameter* requeridos por ésta. Si esto sucede, la *Conversation$_{toCheck}$* no puede ser invocada correctamente debido a la carencia de dichas *Parameter*. Para ello se evaluará cada uno de los *Parameter* de las *Capability* de *Conversation$_{toCheck}$* que no forman parte del *Conversation$_{ada}$* (ya que el resto de las *Capability* ha sido correctamente emparejada previamente) siguiendo los siguientes pasos:

a) Se emparejarán las *Parameter* con cada una de las *Parameter* de la *Conversation$_{ada}$* conel fin de determinar si empareja positivamente con alguna delas *Parameter* requerida en *Conversation$_{ada}$*.

b) Si el paso anterior no determina emparejamiento positivo se determinará si existe algún *ReturnValue* equivalente (que empareje positivamente con el *Parameter*) producida por alguna de las *Capability* de *Conversation$_{toCheck}$* que la preceden en el orden de ejecución del *Automaton*, para así determinar un posible nuevo *DataFlow*.

Así el único proceso a llevar a cabo en ***IOResultChecking(Conversation$_{ada}$, Conversation$_{toCheck}$)*** será comprobar que la *Conversation$_{toCheck}$* no requiera de elementos *Parameter* que no se encuentran en la solicitud. En caso de que la afirmación anterior no se cumpla la *Conversation$_{toCheck}$* no será considerada válida y será eliminada.

PEResultChecking(Conversation$_{ada}$, Conversation$_{toCheck}$): El segundo proceso es más complejo que el anterior, ya que en el presente caso el hecho de que existan elementos *Transtion* extra hace que puedan surgir elementos *Effec*t no deseados y que alteren la información contextual del entorno, haciendo así imposible la obtención del resultado deseado. Para ello es necesario comprobar los dos aspectos que se describen a continuación para cada una de las ramas de ejecución:

- Hay que comprobar la información contextual previa a la invocación de cada una de las *Capability*, con el fin de determinar que las *Condition* requeridas por ésta se cumplen. Esta comprobación se realizará para todas y cada una de las *Transition* de la *Conversation$_{toCheck}$* resultante. Y el proceso a seguir es el siguiente:

  a) Las *Condition* de la *Capability* que contiene la *Conversation$_{toCheck}$* que está siendo analizada se cumplen, por lo que lo podemos tomar como información contextual inicial.

  b) La siguiente comprobación sigue el orden de invocación de las *Capability* definida por la rama de ejecución, y este proceso será repetido por cada una de ellas hasta llegar al final de la rama de ejecución:

     - Por cada *Transition* que contiene la *Conversation$_{toCheck}$* se debe comprobar que las *Condition* que ésta requiere son cumplidas por la información contextual que existe en el estado inicial de la *Transition*.

     - En el estado final de la *Transition* de la *Capability* analizada, la información con-textual será modificada de la manera que describa el conjunto de *Effec*t de ésta.

- Tras finalizar la comprobación de que las *Capability* son invocables (tanto para las *Transition* no solicitadas como para las solicitadas), el siguiente paso consiste en comprobar que el conjunto de elementos *Effec*t obtenidos a la

finalización de la *Conversation*$_{toCheck}$ son equivalentes a la *Conversation* solicitada en *Conversation*$_{ada}$.

calculateConversationDistance(*Conversation*$_{Req}$, *Conversation*$_{res}$)

Una vez que se tiene la lista *resultConversationList* de *Conversation*$_{res}$ el sistema calcula la distancia entre la *Conversation*$_{Req}$ requerida y la *Conversation*$_{res}$ ofrecida mediante la función ***calculateConversationDistance(Conversation*$_{Req}$, *Conversation*$_{res}$)**, siendo para ello necesario tener en consideración varios aspectos:

- Número de *Automaton*$_{cand}$ combinadas para conseguir el *Automaton*$_{res}$ equivalente al *Automaton*$_{Req}$ solicitado, es decir, la granularidad del *Automaton*$_{res}$ calculado respecto al solicitado. Siendo en el presente caso valorado positivamente aquellos *Automaton*$_{res}$ constituidos por un número menor de elementos *Automaton*$_{cand}$, siendo *AutomatonsInResult*$_{res}$ el valor que representa la cantidad de elementos *Automaton*$_{cand}$ combinados para conseguir *Automaton*$_{res}$.

- Número de elementos *SimpleCapability* extra que se encuentran en *Conversation*$_{res}$. Este valor será considerado en el grado de emparejamiento de las *Transition*.

- Grado de emparejamiento entre las *Transition* solicitadas y las ofrecidas. Como las *Transition* están descritas mediante elementos *Symbol* que pueden tener bien una *SimpleCapability* y/o una *Condition* el grado de emparejamiento de las *Transition* tiene que considerar ambas estructuras. Para ello se define la siguiente Ecuación 11 que además tiene en consideración el número de *Transition* extra existentes derivadas de la flexibilidad soportada mediante la dimensión *RelaxationMode*, donde T*RANS*$_{Req}$ representa el número de elementos *Transition* solicitados en la *Conversation*$_{Req}$ y T*RANS*$_{res}$ representa el número de elementos T *ransition* del *Conversation*$_{res}$ y $1 \leq trans_{Req} \leq TRANS_{Req}$:

Ecuación 11

$$TransitionList_{Match_{res}} = \sum_{trans_{Req}=1}^{TRANS_{Req}} \frac{Transition^{res}_{Match_{trans_{Req}}}}{TRANS_{res}}$$

Siendo $Transition^{res}_{Match_{trans_{Req}}}$ definido de la siguiente manera: $Transition^{res}_{Match_{trans_{Req}}} = SimpleCapability^{res}_{Match_{trans_{Req}}} * Condition^{res}_{Match_{trans_{Req}}}$ y representa al grado de emparejamiento de una $Transition$ concreta que se encuentra en una de las adaptaciones $trans_{Req}$ y mediante res se representa a la $Transition$ de $Conversation_{res}$ que es equivalente a la de $Conversation_{Req}$.

> Siendo Transition$^{res}_{Match_{trans_{Req}}}$ definido de la siguiente manera: *Transition*$^{res}_{Match_{trans_{Req}}}$ = *SimpleCapability*$^{res}_{Match_{trans_{Req}}}$ $*$ *Condition*$^{res}_{Match_{trans_{Req}}}$ y representa al grado de emparejamiento de una *Transition* concreta que se encuentra en una de las adaptaciones *trans*$_{Req}$ y mediante *re*s se representa a la *Transition* de *Conversation*$_{res}$ que es equivalente a la de *Conversation*$_{Req}$.

De esta manera combinando las ecuaciones anteriores el cálculo final de *Conversation*$_{Match_{res}}$ queda de la siguiente manera, siendo este valor calculado para cada uno de los *Conversation*$_{res}$ obtenidos en la fase anterior, siendo este el valor devuelto por la función ***calculateConversationDistance(Conversation*$_{Req}$*, Conversation*$_{res}$*)***:

Ecuación 12

$$Conversation_{Match_{res}} = \frac{TransitionList_{Match_{res}}}{AutomatonsInResult_{res}}$$

Una vez realizadas estas operaciones el sistema devolverá al solicitante las descripciones ejecutables, es decir, todas las *Conversation*$_{res}$ equivalentes a la *Conversation*$_{Req}$, las cuales incluyen referencias a las *Conversatio*n ofrecidas por los *Servic*e del entorno. Posteriormente será el solicitante quien determine cual de las posibles soluciones desea enviar al mecanismo de ejecución, el cual ejecutará la descripción invocando así las correspondientes *Capabilit*y necesarias para conseguir el comportamiento requerido.

3.4 Módulo de descubrimiento

A continuación se detallan aspectos relativos a la implementación del módulo de descubrimiento, que suponen una continuación natural de lo explicado en el epígrafe 2.4.

3.4.1 Consideraciones sobre la implementación

Cabe destacar que, si bien el kernel de ISMED se ejecutará en los diferentes dispositivos mencionados en el epígrafe 2.1.2, para el caso que nos ocupa y con el objetivo de clarificar la dimensión y el contenido del módulo de descubrimiento, el despliegue de dicho módulo se ha realizado sobre un nodo cuyo cometido es monitorizar el sistema ISMED.

3.4.2 Estructura del módulo

Con el fin de satisfacer las necesidades del módulo de descubrimiento y tal como ilustra el diagrama de clases presentado en el apartado de diseño (ver el epígrafe 2.4.3), se ha dividido la estructura del módulo de descubrimiento en dos bloques funcionales diferenciados:

- Controlador de Nodo (Node Controller): La misión principal del controlador es gestionar y coordinar la conexión y desconexión de los dispositivos al espacio de tripletas. El controlador proveerá de dos servicios: *connect* y *disconnect* para este fin. Por otra parte hará uso de una serie de primitivas ofrecidas por el API que nos provee el módulo de coordinación semántica:

    - *Conexión* (**connect**): Primitiva ofrecida por el módulo de descubrimiento para establecer el vínculo entre un dispositivo y el Triple Space. El proceso de conexión requiere efectuar los siguientes pasos: Creación del espacio asociado, unión a dicho espacio y arranque del kernel del Triple Space. Durante el proceso de conexión se invocarán las siguientes funciones del API ofrecido por el módulo de modelado:

        - ✓ *create(space)*

- ✓ *join(space)*

- ✓ *startup()*

  - Desconexión (**disconnect**): Primitiva ofrecida por este módulo para interrumpir el enlace entre un dispositivo del ecosistema y el espacio asociado. Para a la desconexión se hace uso de la primitiva:

    - ✓ *shutdown()*

- Gestor de dispositivos (Device Manager): Este bloque funcional da respuesta, ofreciendo una serie de primitivas, al resto de necesidades: Descubrimiento de dispositivos y sistema de suscripción. Destacar que con el objetivo de minimizar el lanzamiento de consultas sobre el espacio, el gestor de dispositivos mantendrá una caché de dispositivos activos y sus sensores asociados, en caso de poseerlos.

  - Sistema de suscripción: Tal y como se ha comentado en el apartado de diseño referente a este módulo, se ha implementado un protocolo de comunicación entre dispositivos, con el objetivo de conseguir que una entidad reconozca la entrada / salida de dispositivos en la red. Se ha hecho uso de las primitivas PUSH que orece el módulo de modelado, para satisfacer dicho protocolo. Primitivas ofrecidas:

    - ✓ **start:** Se invoca en el momento de ingresar en la red, para notificar al resto de nodos su llegada (*advertise*) y realizar la suscripción (*subscribe*) al espacio con el objetivo de recibir las notificaciones producidas por la llegada de nuevos nodos al sistema [1].

    - ✓ **stop:** Primitiva invocada cuando un dispositivo abandona la red. El objetivo es cancelar la suscripción realizada al ingresar (*unsuscribe*) y notificar la salida al resto de nodos (*advertise*)

    [1] Al recibir una notificación por parte de un nuevo dispositivo se procederá a su descubrimiento e inclusión en la caché. Este mecanismo es idéntico al descrito en el apartado de descubrimiento de dispositivos.

- Descubrimiento de dispositivos: Gracias a la primitiva **discover_devices** un nodo será capaz de conocer al resto de dispositivos que conforman la red de ISMED. Como el resto de funciones, se apoya en el módulo de modelado y coordinación semántica. Concretamente se hará uso de la primitiva *query* provista por el API de dicha capa.

  - ✓ Query: Se lanzará, al espacio de tripletas, una consulta por cada tipo de dispositivo contemplado en la ontología de ISMED. Esta consulta se realizará en forma de "template", compuesto por un sujeto, predicado y objeto. En este caso el sujeto es el componente a localizar, siendo el predicado la constante RDF *type* y el objeto el tipo de dispositivo definido en la ontología.

    - Ejemplo de "template" para obtener todos los dispositivos SunSpot: ? <rdf:type> <ismed:SunSpot>

    De esta forma se determinará el número e identidad de los dispositivos presentes en la red. Por cada entidad encontrada se lanzarán determinadas query's para obtener toda la información (detallada en el diagrama de clases) relevante de cada dispositivo.

3.4.3 Esquema de arquitectura y funcionamiento

3.4.3.1 Conexión

Tal y como ilustra la siguiente figura, el módulo de descubrimiento ofrece la primitiva **connect** para establecer la unión con el espacio de tripletas.

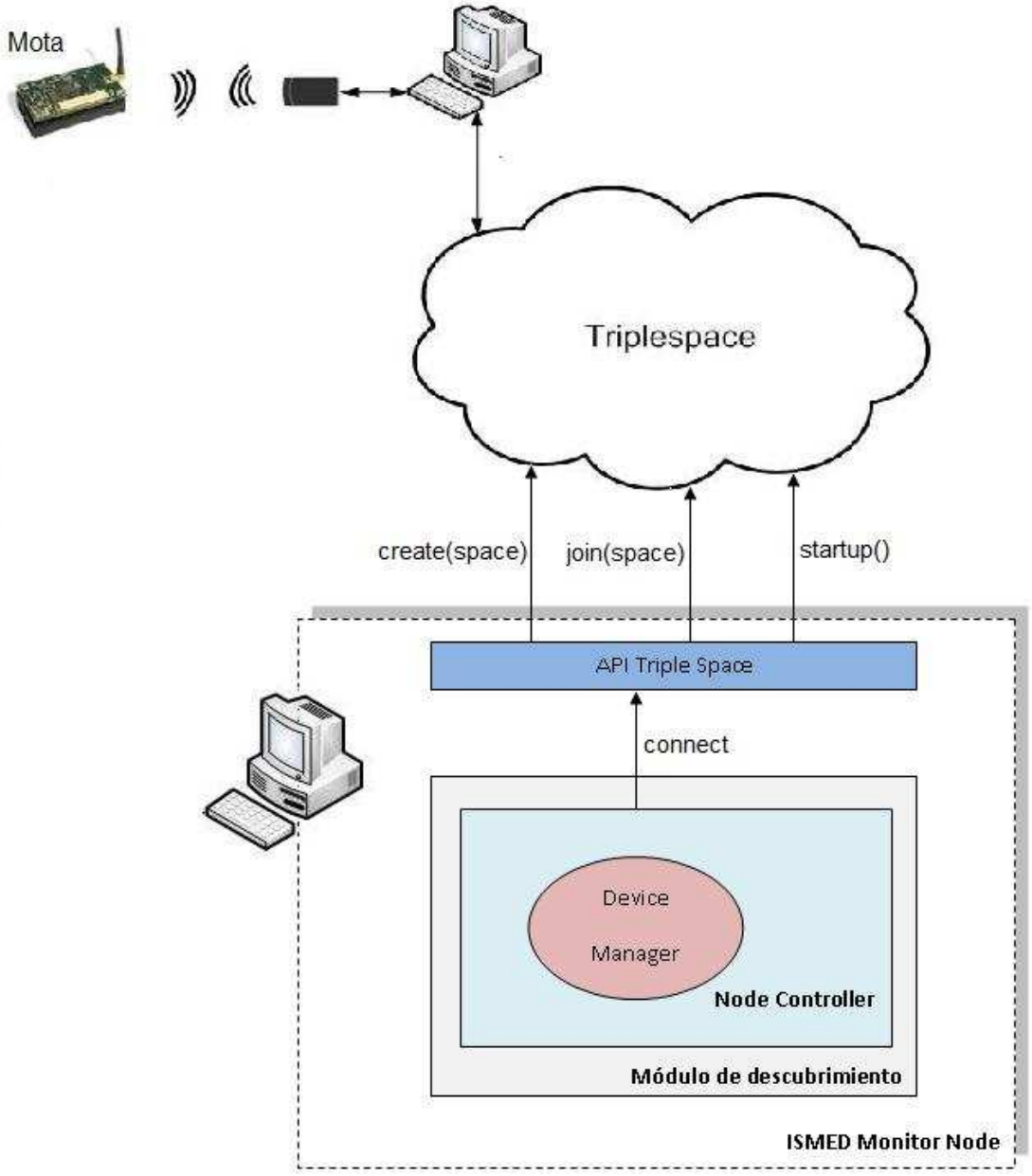


Ilustración 41: Entrada de un nuevo dispositivo - conexión.

3.4.3.2 Suscripción

Con la llegada de un nuevo dispositivo la se realiza una suscripción para recibir notificaciones de otros nodos y notificar al resto de miembros de la red la presencia del mismo.

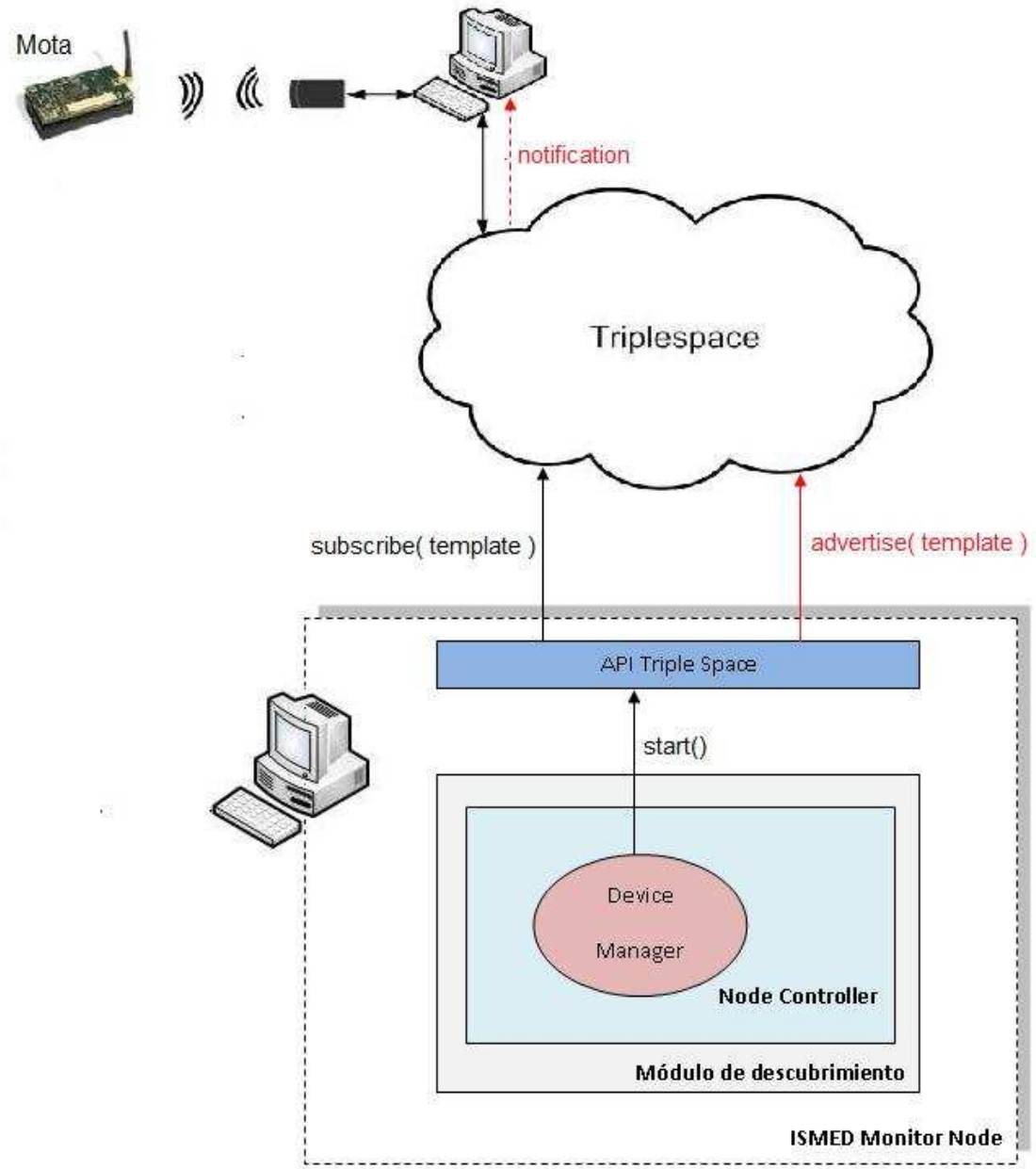


Ilustración 42: Suscripción y notificación de llegada.

3.4.3.3 Descubrimiento

El descubrimiento de dispositivos puede producirse bien por necesidad del nodo en cuestión o por la recepción de una notificación. En el primer caso se analizará la red para detectar la presencia de nuevos dispositivos, mediante querys, sobre el espacio de tripletas. En el segundo caso solo se realizará el descubrimiento del dispositivo que lanzó el aviso, pero de manera análoga al primer caso.

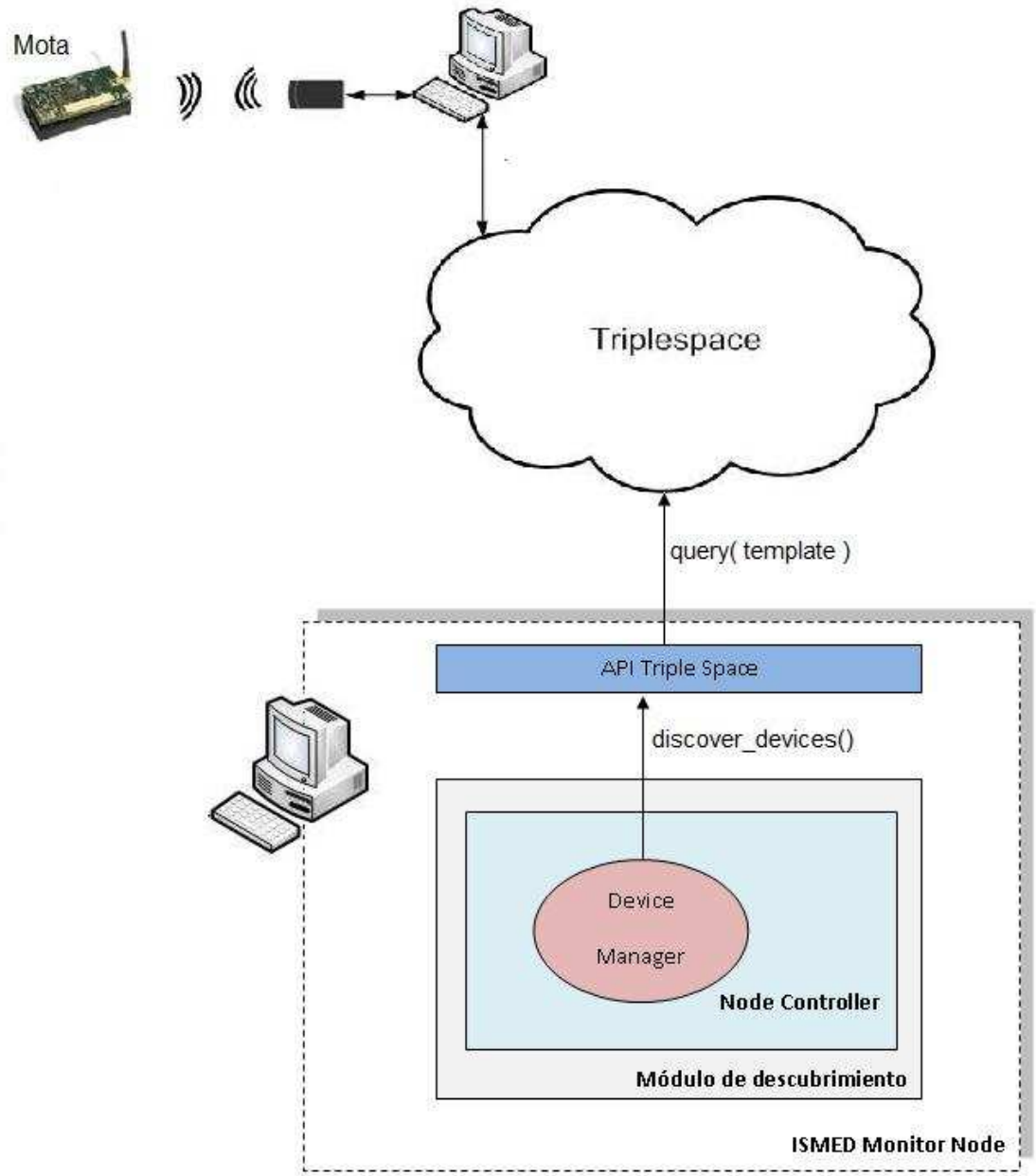


Ilustración 43: Descubrimiento de dispositivos presentes en la red.

3.4.3.4 Des-suscripción

Este escenario se producirá cuando un dispositivo abandone la red de ISMED. En primer lugar el nodo cancelará la suscripción realizada en el momento de su llegada y en segundo lugar, con el objetivo de notificar al resto de miembros su salida, lanzará un aviso mediante la primitiva *advertise*.

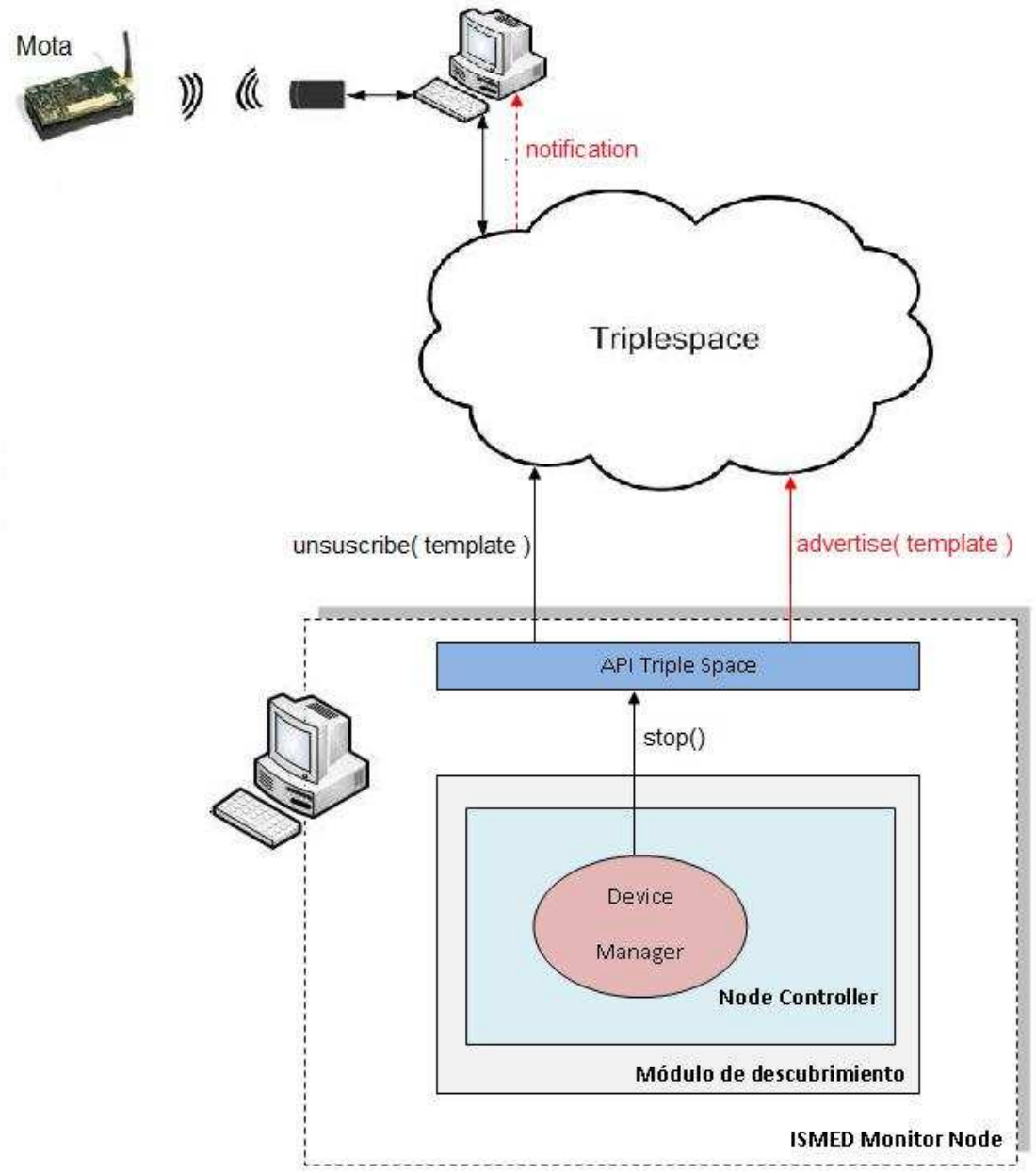


Ilustración 44: Salida de un dispositivo de la red

3.4.3.5 Desconexión

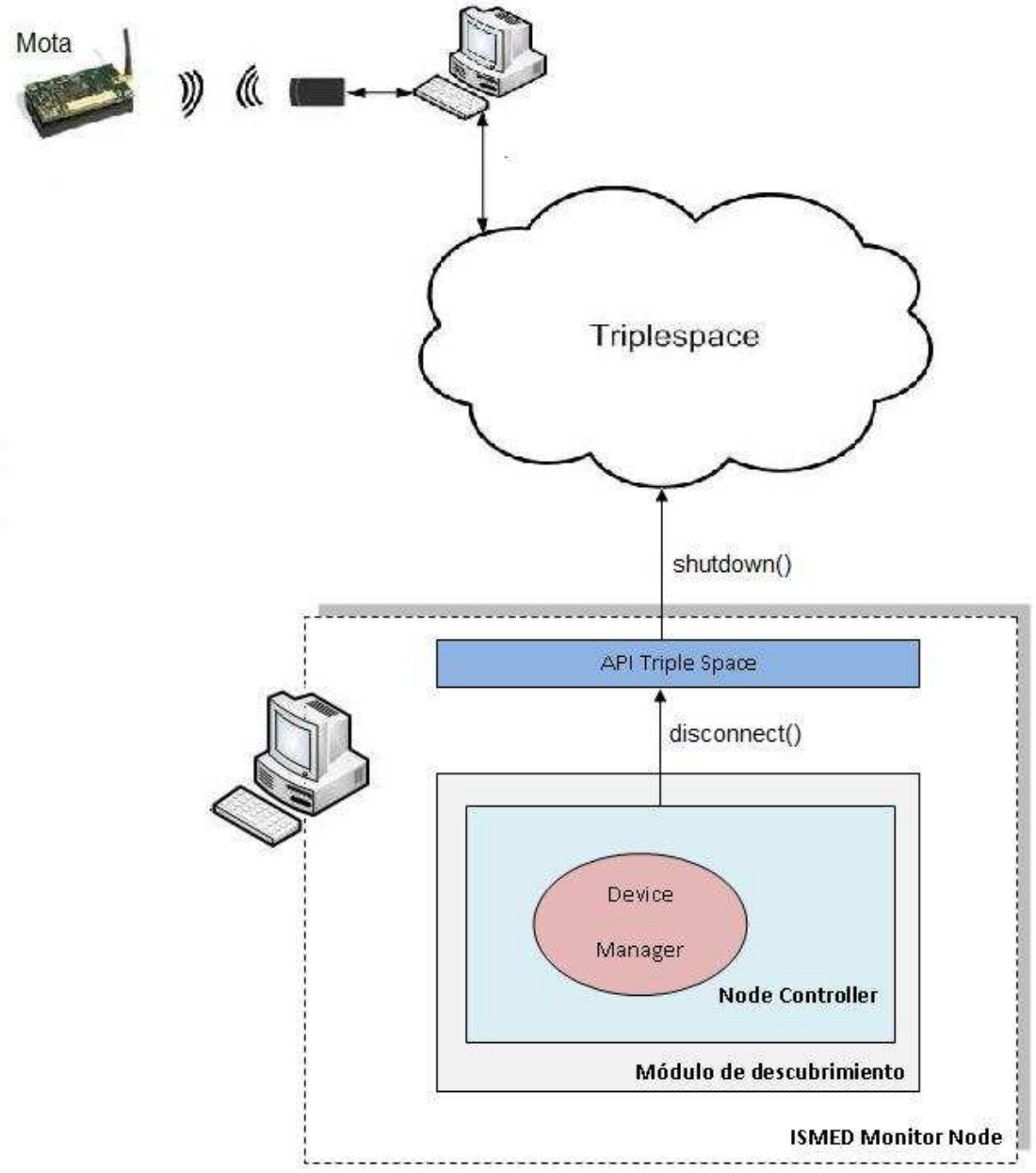


Ilustración 45: Cierre de la conexión

3.4.4 Capturas de pantalla de la aplicación

A continuación se presentan, en forma de capturas de pantalla, las representaciones gráficas de cada escenario propuesto en el epígrafe 3.4.3. Cabe destacar que el despliegue se ha realizado sobre un nodo que pretende realizar las funciones de monitorización de la red ISMED y por tanto la interfaz ha sido diseñada exclusivamente para el caso en cuestión.

3.4.4.1 Nodo sin conexión

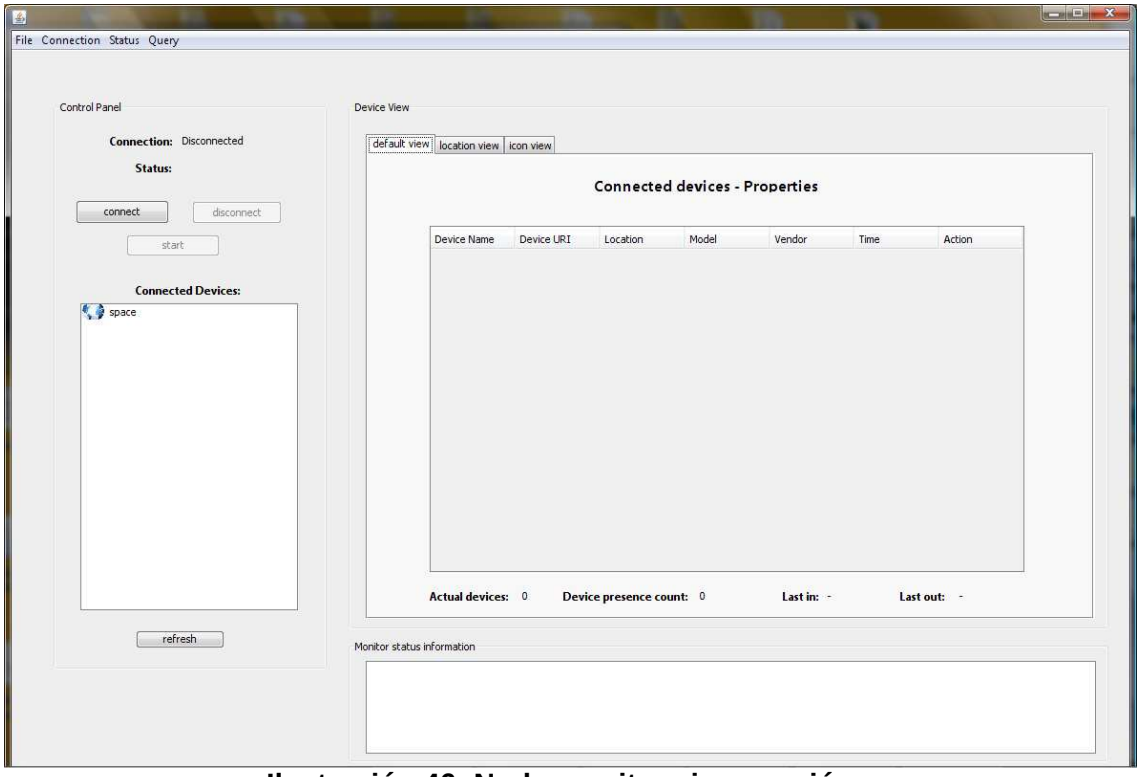
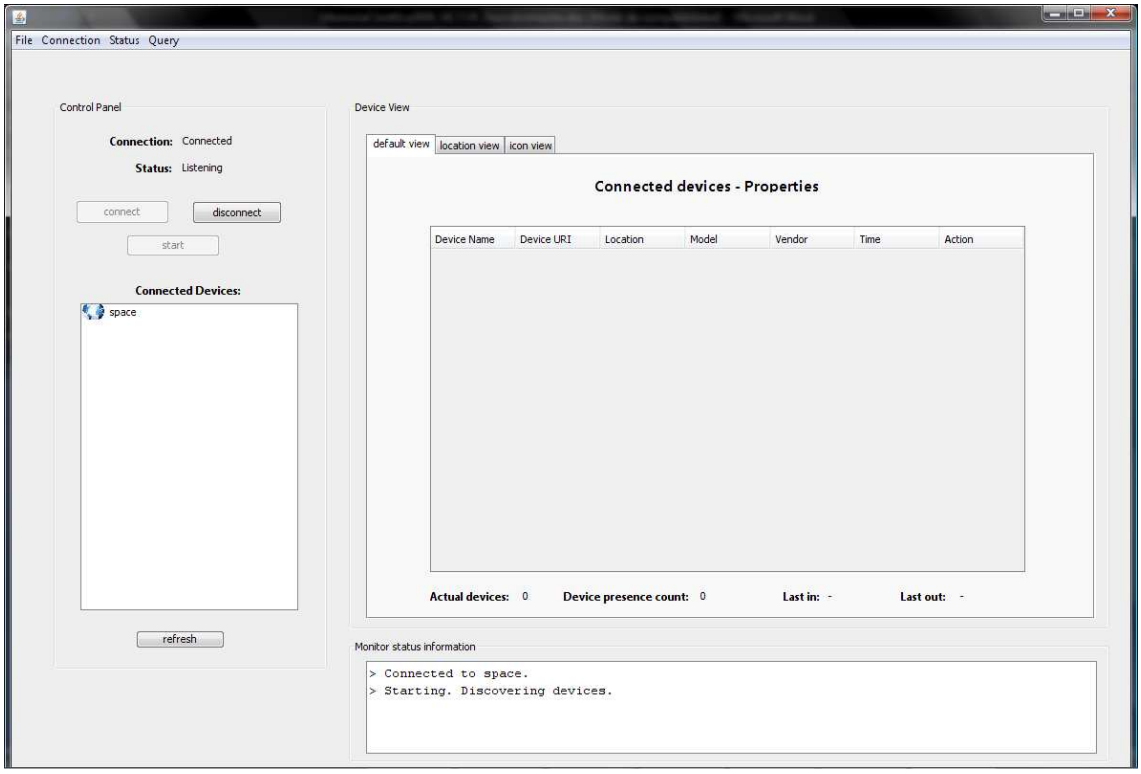
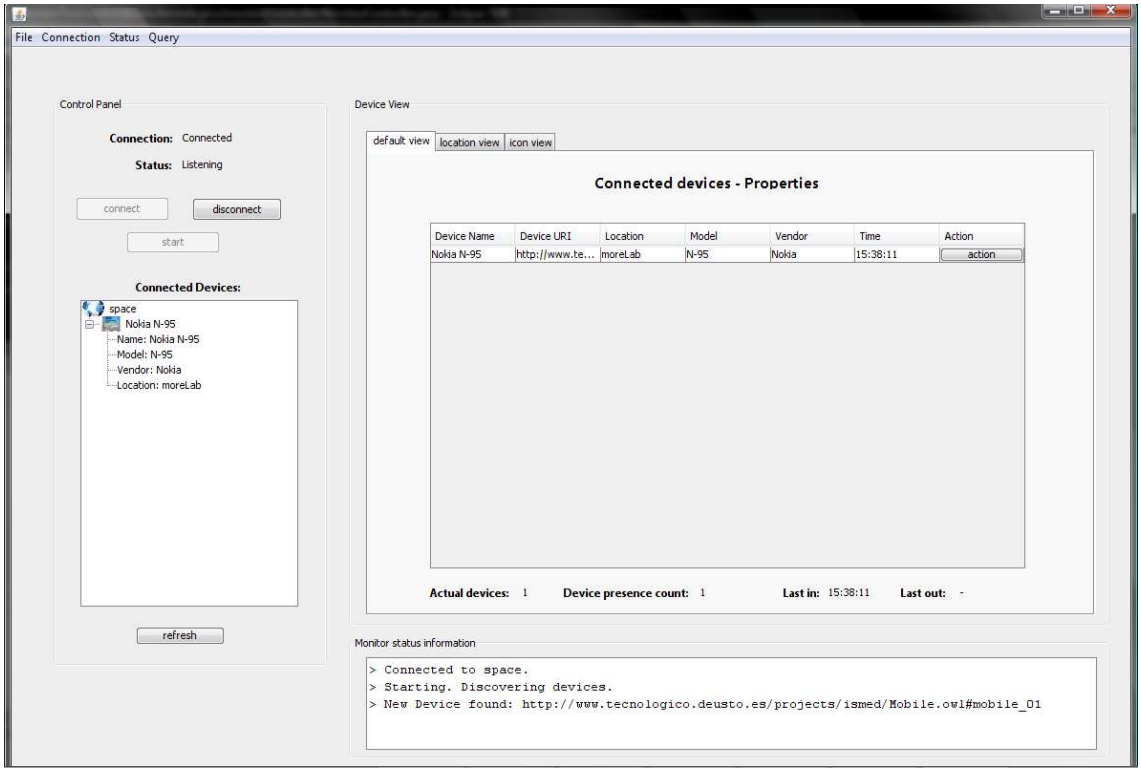


Ilustración 46: Nodo monitor sin conexión

3.4.4.2 Nodo conectado sin dispositivos activos



3.4.4.3 Conexión Nokia-N95



3.4.4.4 Conexión SunSpot

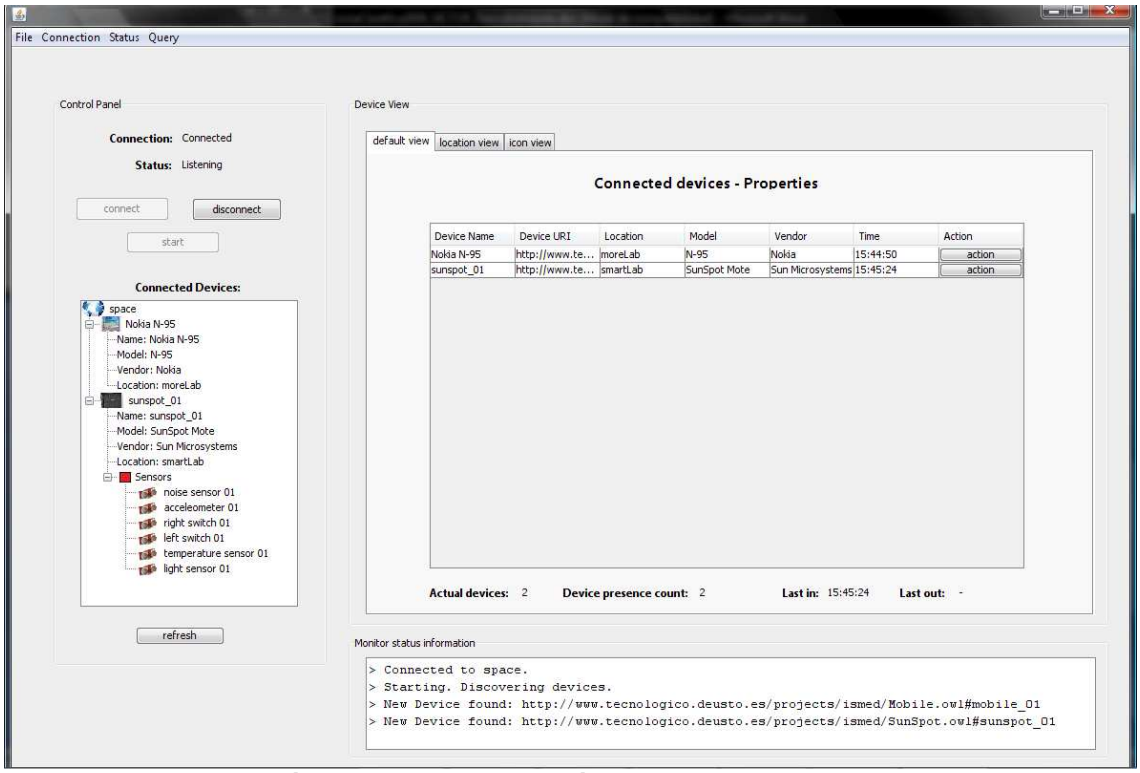


Ilustración 47: Dispositivo móvil y SunSpot conectados.

3.4.4.5 Desconexión SunSpot

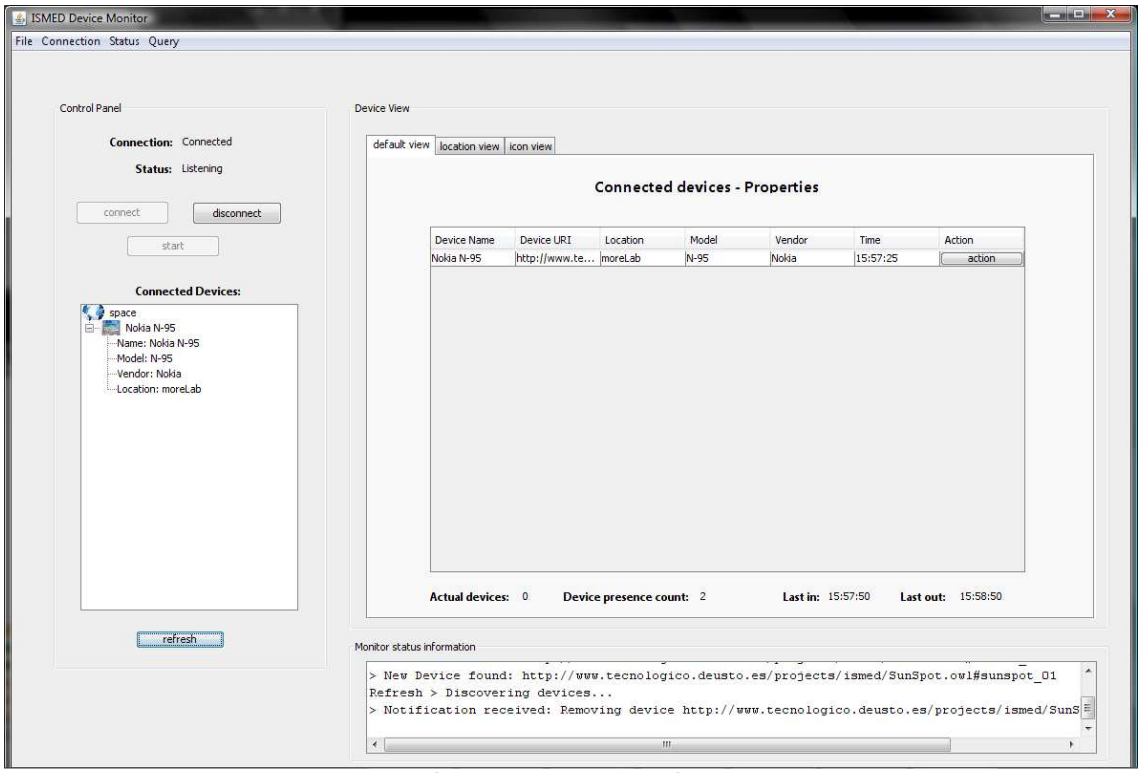


Ilustración 48: Desconexión SunSpot

3.5 Módulo de razonamiento

El **razonamiento distribuido** es un objetivo muy difícilmente alcanzable, por lo que toda inferencia que se pueda realizar en el módulo de coordinación, deberá ser a nivel local.

Una estrategia posible sería consultar a distintos nodos y luego **razonar sobre los resultados obtenidos**. Esto es lo que hace la primitiva *queryMultiple* en esencia tras recibir las respuestas de cada una de los templates básicos que se extraen de la consulta SPARQL.

En la aplicación realizada para probar consultas SPARQL, se ve mejor explicado este enfoque.

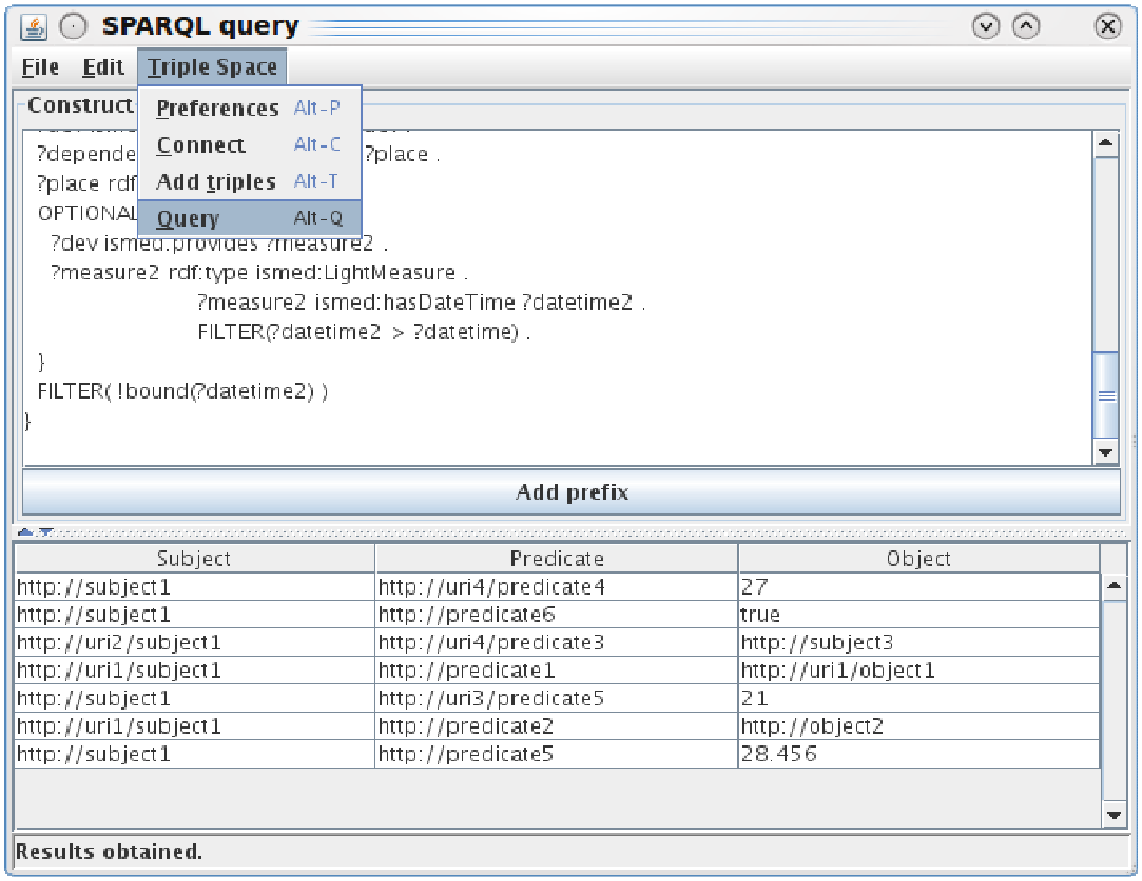


Ilustración 49 . Ventana de consulta desde la que se pueden escribir tripletas y realizar consultas de tipo queryMultiple.

Un caso práctico sería uno en el que se iniciasen tres nodos en el mismo espacio. Dos de ellos escribirían información sobre dos dispositivos distintos ubicados en dos lugares distintos en el mismo espacio (ver *Ilustración 50*).

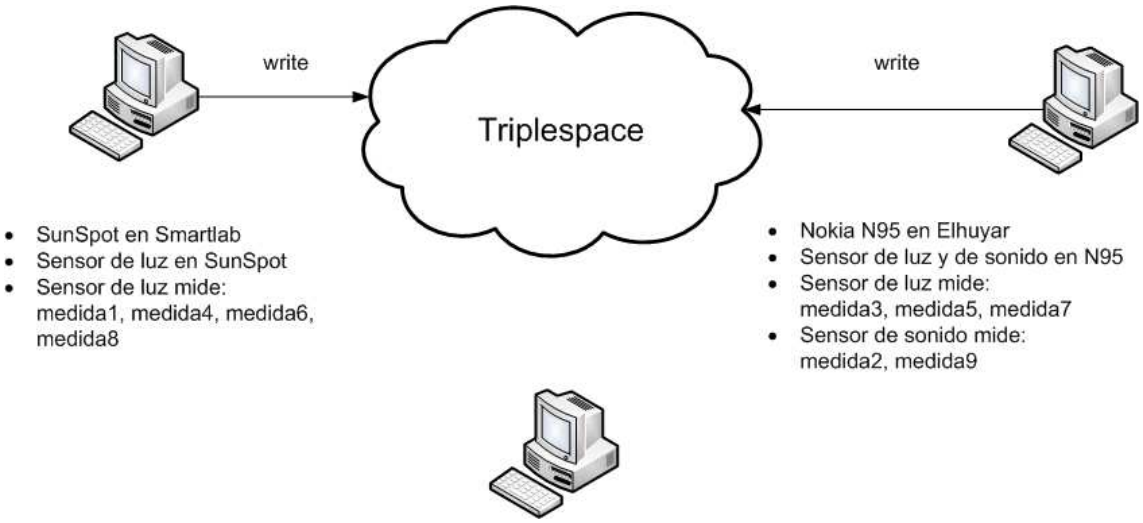


Ilustración 50. Dos nodos escribiendo información en el mismo espacio.

Cuando el tercer nodo realizase una consulta como la que se muestra en la *Tabla* 8, sobre el espacio, esta consulta se descompondría en templates básicos (ver *Tabla 9*), que serían enviados a la vez por la red. Las respuestas serían devueltas de forma individual, y el tercer nodo esperaría un tiempo determinado recogiendo y agrupando cada una de las respuestas recibidas (ver *Ilustración 51*).

Tabla 8. Consulta SPARQL que muestra la última medida de luminosidad tomada por cualquier dispositivo ubicado en una habitación cualquiera.

```
CONSTRUCT {
        ?measure rdf:type ismed:LightMeasure .
        ?measure ismed:hasValue ?value .
        ?measure ismed:hasDateTime ?datetime .
}
WHERE {
        ?measure rdf:type ismed:LightMeasure .
        ?measure ismed:hasValue ?value .
        ?measure ismed:hasDateTime ?datetime .
        ?dev rdf:type ismed:SimpleDevice .
        ?dev ismed:provides ?measure .
        ?dev ismed:isPartOf ?dependentdev .
        ?dependentdev ismed:locatedIn ?place .
        ?place rdf:type ismed:Room .
        OPTIONAL {
                ?dev ismed:provides ?measure2 .
                ?measure2 rdf:type ismed:LightMeasure .
                ?measure2 ismed:hasDateTime ?datetime2 .
                FILTER(?datetime2 > ?datetime) .
        }
        FILTER( !bound(?datetime2) )
}
```

Tabla 9. Templates básicos obtenidos al descomponer la consulta de la *Tabla 8*.

```
?s rdf:type ismed:LightMeasure .
?s ismed:hasValue ?o .
?s ismed:hasDateTime ?o .
?s rdf:type ismed:SimpleDevice .
?s ismed:provides ?o .
?s ismed:isPartOf ?o .
?s ismed:locatedIn ?o .
?s rdf:type ismed:Room .
```

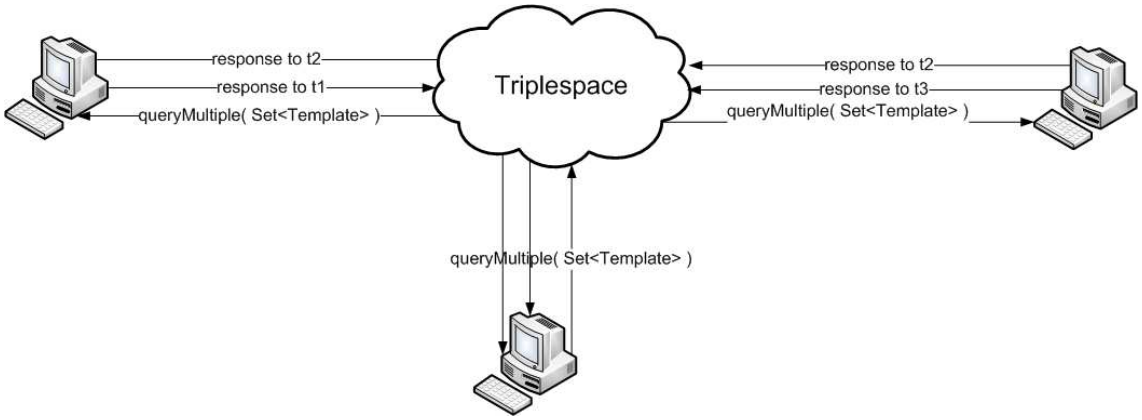


Ilustración 51. El tercer nodo hace una consulta de tipo *queryMultiple* sobre el espacio y los dos nodos restantes le contestan.

Sobre ese conjunto de respuestas recibidas, se aplicaría de nuevo la consulta original a modo de filtro, devolviendo las respuestas concretas a la consulta original. En este último "filtro" se podría realizar inferencia sin mayor complejidad.

Este enfoque, podría usarse incluso a nivel externo del módulo de coordinación, siguiendo el esquema de componentes explicado en el epígrafe 2.1.3.

Otra mejora posible es hacer **inferencia en cada uno de los nodos al responder a una primitiva**. Tsc++ no hace inferencia a nivel de nodo, pero los repositorios que utiliza sí que lo permiten. Aunque no de forma definitiva, se han realizado versiones de SesameDataAccess y OwlimDataAccess que razonaban a nivel local.

Tabla 10. Extracto de código modificado para permitir que Sesame infiera al realizar consultas.

```
File dataDir = new File( storageFolder + "/spaces" );
 // spacesRepository = new SailRepository(new NativeStore(dataDir));
spacesRepository = new SailRepository(
                                new ForwardChainingRDFSInferencer(new NativeStore(dataDir))
                            );
spacesRepository.initialize();
```

Desgraciadamente, Sesame (y consecuentemente Owlim) no son capaces de inferir tripletas en base a un espacio determinado (a nivel de repositorio, el espacio y el grafo

se traducen en recursos, que son agrupaciones jerárquicas lógicas de tripletas[12]) y usan como base para dicha inferencia el total de las tripletas albergadas en dicho repositorio. Para solucionar eso, en las versiones preliminares realizadas, se ha creado un nuevo repositorio por cada espacio.

Tabla 11. Extracto de código en el que se inserta un repositorio en el mapa creado a tal efecto.

```
File dataDir = new File( storageFolder + "/spaces" );
spacesRepositories.put (
    spaceuri,
    new SailRepository( new ForwardChainingRDFSInferencer(new NativeStore(dataDir)))
);
spacesRepositories.get(spaceuri).initialize();
```

[12] Así, la primitiva de org.openrdf.repository.RepositoryConnection.add() escritura en Sesame es:

```
void add(Iterable<Statement> arg0, Resource... arg1) throws RepositoryException
```

Y en SesameDataAccess se traduce a:

```
con.add(triples, contextSpaceURI, contextGraphURI);
```

4 CONCLUSIONES

A lo largo del año 2009 se ha avanzado en los distintos objetivos para cada uno de los módulos que componen ISMED de forma desigual, pero notable en todos ellos. Se han resuelto interesantes problemáticas, y las primeras aplicaciones han dejado entrever la potencia y flexibilidad que el paradigma Triple Space puede ofrecer en entornos de computación ubicua.

Pese a ello, como se ha descrito en el informe, durante el año 2009 se han encontrado diversos aspectos técnicos cuya complejidad ha resultado ser mayor de lo que se pudo esperar en un primer momento. Estos problemas han hecho que la planificación general se retrase y que haya habido que modificar la definición de determinados aspectos. Todo ello, eso sí, sin afectar ni un ápice al grado de interés y utilidad del proyecto.

Durante el año 2010, se finalizará la integración de los distintos módulos, ofreciendo escenarios en los que la utilidad del middleware desarrollado quede patente. Todo ello, sin perder de vista que el software del que se dispone a día de hoy se puede refinar mucho más obteniendo una versión realmente estable y útil incluso para otros proyectos ajenos a ISMED.

5 BIBLIOGRAFÍA

[1] M. Murth*, et al.*, "D6.2 Triple Space reference architecture.," TripCom2008.

[2] L. J. Nixon*, et al.*, "D6.3 Platform API Specification for Interaction Between All Components," TripCom2008.

[3] L. J. B. Nixon*, et al.*, "D2.4 Semantic Clustering and Self-Organization in Triple Space," TripCom2008.

[4] L. J. B. Nixon*, et al.*, "D3.3 Semantic matching in distributed spaces," TripCom2008.

[5] L. J. B. Nixon*, et al.*, "D3.4 Distributed Semantic Query Tool for Triple Space," TripCom.

[6] A. Gómez Goiri*, et al.*, "ISMED: Intelligent Semantic Middleware for Embedded Devices, Informe científico anual 2008," unpublished|.

[7] (2008, *Sun SPOT World*. Available: http://www.sunspotworld.com/

[8] Crossbow. (2009, *MICAz 2.4 GHz*. Available: http://www.xbow.com/Products/productdetails.aspx?sid=164

[9] (2008, *Gumstix Web Page*. Available: http://www.gumstix.com/

[10] Digi. *XBee Sensors*. Available: http://www.digi.com/products/wirelessdropinnetworking/sensors/xbee-sensors.jsp#overview

[11] Digi. *ConnectPort X Gateways*. Available: http://www.digi.com/products/wirelessdropinnetworking/connectportxgateways.jsp#overview

[12] R. Krummenacher*, et al.*, "tsc++ user guide," STI Innsbruck2005.

[13] R. Hervás Lucas, "Modelado de contexto para la visualización de información en ambientes inteligentes," 2008.

[14] R. Agrawal and R. Srikant, "Mining Sequential Patterns," pp. 3-14.

[15] A. W. W. van der Aalst, and L. Maruster, "Workflow mining discovering process models from event logs," *IEEE Transactions*

on Knowledge and Data Engineering, vol. 18, pp. 1128–1142.

[16] J. Broekstra*, et al.*, "Sesame: A Generic Architecture for Storing and Querying RDF," presented at the International Semantic Web Conference, Sardinia, Italy.

[17] O. Lab., "Fact Sheet," Sirma Group Corp.2007.

[18] S. Ali and S. Kiefer, "µOR – A Micro OWL DL Reasoner for Ambient Intelligent Devices," presented at the 4th International Conference, GPC 2009, Geneva, Switzerland.

[19] I. Fernández Gorostizaga, *et al.*, “Desarrollo de un sistema P2P en .NET con un motor de búsqueda basado en Web Semántica”, *unpublished.*