UNIÓN INTERNACIONAL DE TELECOMUNICACIONES

UIT-T T.125

SECTOR DE NORMALIZACIÓN
DE LAS TELECOMUNICACIONES
DE LA UIT

(04/94)

SERVICIOS TELEMÁTICOS

EQUIPOS TERMINALES Y PROTOCOLOS PARA LOS SERVICIOS DE TELEMÁTICA

ESPECIFICACIÓN DE PROTOCOLO DEL SERVICIO DE COMUNICACIÓN MULTIPUNTO

Recomendación UIT-T T.125
Reemplazada por una versión más reciente

(Anteriormente «Recomendación del CCITT»)

PREFACIO

El UIT-T (Sector de Normalización de las Telecomunicaciones) es un órgano permanente de la Unión Internacional de Telecomunicaciones (UIT). Este órgano estudia los aspectos técnicos, de explotación y tarifarios y publica Recomendaciones sobre los mismos, con miras a la normalización de las telecomunicaciones en el plano mundial.

La Conferencia Mundial de Normalización de las Telecomunicaciones (CMNT), que se celebra cada cuatro años, establece los temas que han de estudiar las Comisiones de Estudio del UIT-T, que a su vez producen Recomendaciones sobre dichos temas.

La aprobación de Recomendaciones por los Miembros del UIT-T es el objeto del procedimiento establecido en la Resolución N.° 1 de la CMNT (Helsinki, 1 al 12 de marzo de 1993).

La Recomendación UIT-T T.125 ha sido preparada por la Comisión de Estudio 8 del UIT-T y fue aprobada por el procedimiento de la Resolución N.° 1, el 7 de Abril de 1994.

NOTA

En esta Recomendación, la expresión «Administración» se utiliza para designar, en forma abreviada, tanto una administración de telecomunicaciones como una empresa de explotación reconocida de telecomunicaciones.

Reemplazada por una versión más reciente

ÍNDICE

Página

RESUMEN

Esta Recomendación define un protocolo que funciona a través de la jerarquía de un dominio de comunicación multipunto. Especifica el formato de mensajes y procedimientos de protocolo que rigen el intercambio mediante un conjunto de conexiones de transporte. La finalidad del protocolo es realizar el servicio de comunicación multipunto definido en la Recomendación UIT-T T.122.

Recomendación T.125

ESPECIFICACIÓN DE PROTOCOLO DEL SERVICIO DE COMUNICACIÓN MULTIPUNTO

(Ginebra, 1994)

1 Alcance

Esta Recomendación especifica:

a) los procedimientos de un protocolo único para la transferencia de información de datos y de control de un proveedor MCS a otro proveedor MCS par;

b) la estructura y codificación de las unidades de datos de protocolo MCS utilizadas para la transferencia de información de datos y de control.

Los procedimientos se definen en función de:

a) las interacciones entre proveedores MCS pares mediante el intercambio de unidades de datos de protocolo MCS;

b) las interacciones entre un proveedor MCS y usuarios MCS mediante el intercambio de primitivas MCS;

c) las interacciones entre un proveedor MCS y un proveedor del servicio de transporte mediante el intercambio de primitivas del servicio de transporte.

Estos procedimientos son aplicables a ejemplares (dícese instancias) de comunicación multipar entre sistemas que soportan MCS y desean interconectar en un entorno de sistemas abiertos.

2 Referencias

Las siguientes Recomendaciones y otras referencias contienen disposiciones, que mediante su referencia en este texto, constituyen disposiciones de esta Recomendación. Al efectuar esta publicación, estaban en vigor las ediciones indicadas. Todas las Recomendaciones y otras referencias son objeto de revisiones, por lo que se preconiza que los usuarios de la presente Recomendación investiguen la posibilidad de aplicar la edición más reciente de las Recomendaciones y otras referencias indicadas a continuación. Se publica periódicamente una lista de las Recomendaciones del UIT-T actualmente válidas.

– Recomendación UIT-T T.122 (1993), *Servicio de comunicación multipunto para la conferencia audiográfica y la conferencia audiovisual – Definición del servicio*.

– Recomendación UIT-T T.123 (1993), *Pilas de protocolos para aplicaciones de teleconferencia audiográfica y audiovisual*.

– Recomendación CCITT X.200 (1988), *Modelo de referencia de interconexión de sistemas abiertos para aplicaciones del CCITT*.

– Recomendación CCITT X.214 (1988), *Definición del servicio de transporte para la interconexión de sistemas abiertos para aplicaciones del CCITT*.

– Recomendación CCITT X.208 (1988), *Especificación de la notación de sintaxis abstracta uno (ASN.1)*.

– CCITT Recomendación X.209 (1988), *Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*.

– Proyecto de Recomendación UIT-T X.691 | ISO/CEI 8825-2, *Information Technology – Open Systems Interconnection – ASN.1 encoding rules – Specification of Packed Encoding Rules*.

3 Definiciones

NOTA – En estas definiciones se utilizan las abreviaturas definidas en la cláusula 4.

Esta Recomendación se basa en los conceptos elaborados en la Recomendación X.200 del CCITT y emplea los siguientes términos definidos en la misma:

a) control de flujo;

b) reensamblado;

c) recombinación;

d) segmentación;

e) secuenciación;

f) división;

g) sintaxis de transferencia;

h) conexión de transporte;

i) identificación de punto extremo de conexión de transporte;

j) servicio de transporte;

k) punto de acceso al servicio de transporte;

l) dirección de punto de acceso al servicio de transporte;

m) unidad de datos del servicio de transporte.

Esta Recomendación se basa también en los conceptos elaborados en la Recomendación UIT-T T.122 y emplea los siguientes términos definidos en la misma:

a) MCSAP de control;

b) anexión MCS;

c) canal MCS;

d) conexión MCS;

e) dominio MCS;

f) selector de dominio MCS;

g) canal privado MCS;

h) gestor de canal privado MCS;

i) proveedor MCS;

j) punto de acceso al servicio MCS;

k) usuario MCS;

l) identificador de usuario MCS;

m) proveedor MCS superior.

A los fines de esta Recomendación son aplicables las siguientes definiciones.

3.1 unidad de datos del servicio MCS: Una cantidad de datos de usuario MCS cuya identidad queda preservada durante la transferencia de transmisor a receptores. Específicamente, el contenido de una petición MCS-ENVÍO-DATOS o de una petición MCS-ENVÍO-DATOS-UNIFORME.

3.2 unidad de datos de interfaz MCS: La unidad de información transferida a través de un MCSAP entre un usuario MCS y un proveedor MCS en una sola interacción. Cada unidad de datos de interfaz MCS contiene información de control de interfaz y puede también contener una unidad de datos del servicio MCS o parte de ella.

3.3 unidad de datos de protocolo MCS: Una unidad de información intercambiada en el protocolo MCS, constituida por información de control transferida entre proveedores MCS para coordinar su operación combinada y posiblemente datos transferidos a nombre de usuarios MCS a los cuales están prestando servicio.

3.4 prioridad de transferencia de datos MCS: Uno de cuatro niveles: superior, alto, medio, bajo. El valor se comunica sin modificación del transmisor a los receptores. En función de un parámetro de dominio MCS que da el número de prioridades distintas de datos implementadas, dos o más prioridades de las más bajas pueden recibir la misma calidad de servicio.

3.5 **MCSPDU válida**: Una MCSPDU cuya estructura y codificación se ajustan a esta Recomendación.

3.6 **MCSPDU inválida**: Una MCSPDU que no es válida.

3.7 **error de protocolo**: Utilización de una MCSPDU de una manera que no se ajusta a los procedimientos de esta Recomendación.

3.8 **MCSPDU de conexión**: Cualquiera de las siguientes primitivas: **conexión-inicial**, **conexión-respuesta**, **conexión-adicional**, **conexión-resultado**.

3.9 **MCSPDU de dominio**: Toda MCSPDU que no sea una MCSPDU de conexión.

3.10 **MCSPDU de datos**: Cualquiera de las siguientes: **SDrq**, **SDin**, **USrq**, **USin**.

3.11 **MCSPDU de control**: Cualquier MCSPDU dominio que no sea una MCSPDU de datos.

3.12 **TC inicial**: La primera conexión de transporte de una conexión MCS, utilizada para intercambiar MCSPDU de control y MCSPDU de datos de la prioridad más alta.

3.13 **TC adicional**: Una ulterior conexión de transporte que pertenece a una conexión MCS y se utiliza para intercambiar MCSPDU de datos de una prioridad más baja.

3.14 **subárbol de un proveedor MCS**: En el contexto de un dominio MCS, está constituido por el propio proveedor MCS y sus anexiones MCS, junto con sus proveedores MCS jerárquicamente subordinados y las anexiones MCS de éstos.

3.15 **altura de un proveedor MCS**: En el contexto de un dominio MCS, es un valor igual a la altura máxima de todos los proveedores MCS jerárquicamente subordinados aumentada en una unidad. Un proveedor MCS sin subordinados tiene una altura de uno.

4 Abreviaturas

A los efectos de esta Recomendación se utilizan las siguientes abreviaturas:

MCS	Servicio de comunicación multipunto (*multipoint communication service*)
MCSAP	Punto de acceso al servicio MCS (*MCS service access point*)
MCSPDU	Unidad de datos de protocolo MCS (*MCS protocol data unit*)
TC	Conexión de transporte (*transport connection*)
TS	Servicio de transporte (*transport service*)
TSAP	Punto de acceso al servicio de transporte (*transport service access point*)
TSDU	Unidad de datos del servicio de transporte (*transport service data unit*)

5 Visión de conjunto del protocolo MCS

5.1 Modelo de la capa MCS

Un proveedor MCS comunica con usuarios MCS a través de un MCSAP por medio de las primitivas MCS definidas en la Recomendación UIT-T T.122. Estas primitivas pueden ser la causa o el resultado de intercambio entre proveedores MCS pares que utilizan una conexión MCS, o pueden ser la causa o el resultado de acciones ejecutadas dentro de un solo proveedor MCS. Los intercambios de MCSPDU tienen lugar entre proveedores MCS que atienden el mismo dominio MCS.

Un proveedor MCS puede tener múltiples proveedores pares, a cada uno de los cuales se llega directamente por una conexión MCS o indirectamente a través de un proveedor MCS par. Una conexión MCS se compone de una o más conexiones de transporte, lo que depende del número de prioridades de transferencia de datos implementadas en un dominio MCS. Los intercambios de protocolo se efectúan utilizando los servicios de la capa de transporte a través de un par de TSAP.

Este modelo de la capa MCS se ilustra en la Figura 5-1.

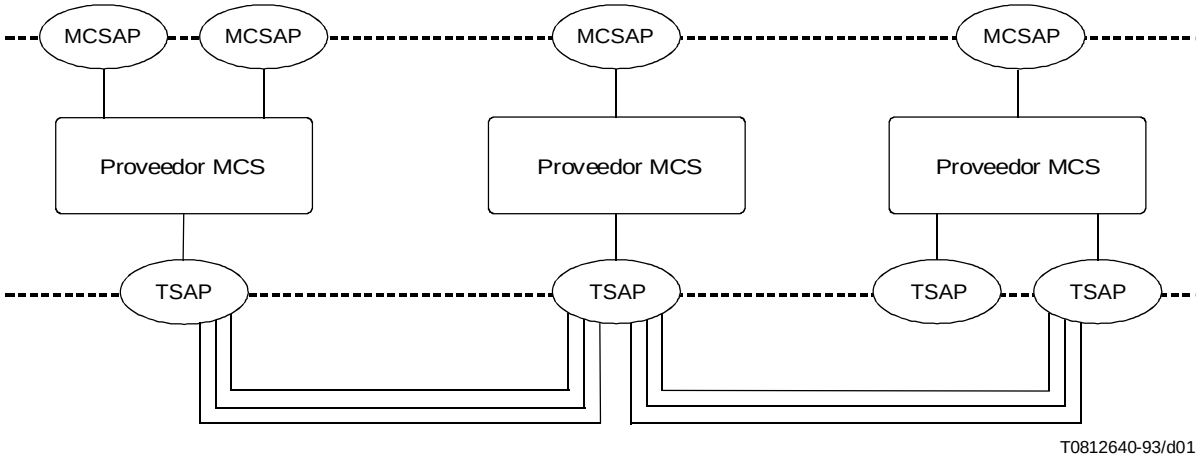


FIGURA 5-1/T.125

Modelo de la capa MCS

5.2 Servicios proporcionados por la capa MCS

El protocolo MCS soporta los servicios definidos en la Recomendación UIT-T T.122. La información se transfiere a y desde un usuario MCS mediante las primitivas indicadas en el Cuadro 5-1.

5.3 Servicios tomados de la capa de transporte

El protocolo MCS hace uso de un subconjunto del servicio de transporte orientado a conexión definido en la Recomendación X.214 del CCITT. Para transferir información a, o desde, un proveedor del servicio de transporte se utilizan las primitivas indicadas en el Cuadro 5-2.

5.4 Funciones de la capa MCS

El Cuadro 5-1 identifica las unidades funcionales de MCS y las MCSPDU asociadas con cada primitiva MCS. Las MCSPDU se definen en la cláusula 7. Las primitivas y las MCSPDU pueden encontrarse en una simple relación de causa a efecto, en un sentido o en el otro. Por ejemplo, petición MCS-ANEXIÓN-USUARIO genera **AUrq**, y **AUcf** genera confirmación MCS-ANEXIÓN-USUARIO. Otros casos pueden ser más complicados. Para la compleción de, por ejemplo, MCS-CONEXIÓN-PROVEEDOR es necesario el intercambio de MCSPDU adicionales como efectos secundarios de la primitiva de cuatro fases. Además, una cualquiera de cinco MCSPDU puede causar una indicación MCS-DESANEXIÓN-USUARIO, y una cualquiera de cuatro puede causar una indicación MCS-EXCLUSIÓN-CANAL.

5.4.1 Gestión de dominio

La capa MCS mantiene la integridad de las conexiones MCS que constituyen un dominio MCS. Una conexión MCS es una conexión orientada, siendo uno de sus extremos el superior jerárquico del otro. Hay un solo proveedor a la cabeza de cada dominio.

Al establecerse una conexión, dos dominios se fusionan en uno. La capa MCS asegura que quede un proveedor superior. De ese modo se resuelve cualquier conflicto de identidad única o propiedad exclusiva que puedan surgir.

Al desconectarse una conexión, un dominio se divide en dos porciones. La porción que contiene el proveedor superior subsiste. La porción inferior desaparece por sí misma.

La capa MCS identifica inequívocamente a los usuarios anexionados a un dominio. Los usuarios pueden tener conocimiento de cada uno de los demás mediante sus interacciones por primitivas MCS. La capa MCS notifica a todos los usuarios de un dominio cuando uno de ellos ha quedado desafectado. La capa MCS recupera los recursos de un usuario desanexionado.

CUADRO 5-1/T.125

Primitivas MCS

Unidad funcional	Primitivas	MCSPDU asociadas
Gestión de dominio	Petición MCS-CONEXIÓN-PROVEEDOR Indicación MCS-CONEXIÓN-PROVEEDOR Respuesta MCS-CONEXIÓN-PROVEEDOR Confirmación MCS-CONEXIÓN-PROVEEDOR (efectos secundarios)	**Conexión-inicial** **Conexión-inicial** **Conexión-respuesta** **Conexión-respuesta** **Conexión-adicional** **Conexión-resultado** **PDin EDrq** **MCrq MCcf PCin** **MTrq MTcf PTin**
	Petición MCS-DESCONEXIÓN-PROVEEDOR Indicación MCS-DESCONEXIÓN-PROVEEDOR	**DPum** **DPum RJum**
	Petición MCS-ANEXIÓN-USUARIO Confirmación MCS-ANEXIÓN-USUARIO	**AUrq** **AUcf**
	Petición MCS-DESANEXIÓN-USUARIO Indicación MCS-DESANEXIÓN-USUARIO	**DUrq** **DUin** **MCcf PCin** **MTcf PTin**
Gestión de canal	Petición MCS-INCORPORACIÓN-CANAL Confirmación MCS-INCORPORACIÓN-CANAL	**CJrq** **CJcf**
	Petición MCS-ABANDONO-CANAL Indicación MCS-ABANDONO-CANAL	**CLrq** **MCcf PCin**
	Petición MCS-FORMACIÓN-CANAL Confirmación MCS-FORMACIÓN-CANAL	**CCrq** **CCcf**
	Petición MCS-DISOLUCIÓN-CANAL Indicación MS-DISOLUCIÓN-CANAL	**CDrq** **MCcf PCin**
	Petición MCS-ADMISIÓN-CANAL Indicación MCS-ADMISIÓN-CANAL	**CArq** **CAin**
	Petición MCS-EXCLUSIÓN-CANAL Indicación MCS-EXCLUSIÓN-CANAL	**CErq** **CEin CDin** **MCcf PCin**
Transferencia de datos	Petición MCS-ENVÍO-DATOS Indicación MCS-ENVÍO-DATOS	**SDrq** **SDin**
	Petición MCS-ENVÍO-DATOS-UNIFORME Indicación MCS-ENVÍO-DATOS-UNIFORME	**USrq** **USin**

CUADRO 5-1/T.125 *(fin)*

Primitivas MCS

Unidad funcional	Primitivas	MCSPDU asociadas
Gestión de testigo	Petición MCS-TOMA-TESTIGO Confirmación MCS-TOMA-TESTIGO	**TGrq** **TGcf**
	Petición MCS-INHIBICIÓN-TESTIGO Confirmación MCS-INHIBICIÓN-TESTIGO	**TIrq** **TIcf**
	Petición MCS-CESIÓN-TESTIGO Indicación MCS-CESIÓN-TESTIGO Respuesta MCS-CESIÓN-TESTIGO Confirmación MCS-CESIÓN-TESTIGO	**TVrq** **TVin** **TVrs** **TVcf**
	Petición MCS-SOLICITUD-TESTIGO Indicación MCS-SOLICITUD-TESTIGO	**TPrq** **TPin**
	Petición MCS-LIBERACIÓN-TESTIGO Confirmación MCS-LIBERACIÓN-TESTIGO	**TRrq** **TRcf**
	Petición MCS-PRUEBA-TESTIGO Confirmación MCS-PRUEBA-TESTIGO	**TTrq** **TTcf**

5.4.2 Gestión de canal

La capa MCS registra qué partes de un dominio MCS contienen uno o más usuarios incorporados a un determinado canal, por lo que puede optimizar la transferencia de datos a destinos que deseen recibirlos.

La capa MCS trata los identificadores de usuario (id de usuario) como si fuesen canales de un solo miembro, a los cuales sólo el usuario designado está autorizado a incorporarse. A petición, puede crear canales privados a los cuales sólo tiene autorizado el acceso los usuarios admitidos, o asignar canales públicos que no tengan incorporados otros usuarios en ese momento.

5.4.3 Transferencia de datos

La capa MCS mantiene un flujo secuenciado de datos a los usuarios que se han incorporado al canal. Un canal se convierte, en efecto, en una lista de distribución para multidifusión, con una gama comprendida entre ningún destino y la totalidad de los destinos en una difusión completa.

Por defecto, la capa MCS encamina datos a cada receptor a través del trayecto más corto de conexiones MCS. Facultativamente, encamina las unidades de datos de servicio MCS especificadas, a través del proveedor MCS superior, con lo que se garantiza su recepción uniforme por todos los receptores, entre los cuales puede estar también incluido el transmisor.

La capa MCS reconoce una o más prioridades de transferencia de datos y les concede un tratamiento preferencial. La segmentación permite trabajar con unidades de datos de servicio MCS de tamaño ilimitado.

La capa MCS regula el flujo global de datos dentro de un dominio. La inaptitud de un receptor para aceptar datos a la velocidad a que son ofrecidos repercute en la aparición de una presión hacia el origen (o retropresión) que causa el bloqueo de los transmisores. Un usuario puede ser desanexionado involuntariamente si deja de mantener una velocidad mínima de recepción.

La capa MCS garantiza la recepción sin error de los datos transmitidos mientras que los usuarios de origen y de destino permanezcan anexionados y el usuario de destino permanezca incorporado al canal. Sin embargo, los datos con prioridad más alta, tienen precedencia, por lo que un exceso de éstos puede demorar indefinidamente la entrega de datos con prioridad más baja.

CUADRO 5-2/T.125

Primitivas del servicio de transporte

Primitivas	Uso	Parámetros	Uso
Petición T-CONEXIÓN Indicación T-CONEXIÓN	X X	Dirección llamada Dirección llamante Opción datos acelerados Calidad de servicio Datos de usuario TS	X X – X –
Respuesta T-CONEXIÓN Confirmación T-CONEXIÓN	X X	Dirección respondedora Opción datos acelerados Calidad de servicio Datos de usuario TS	– – X –
Petición T-DATOS Indicación T-DATOS	X X	Datos de usuario TS	X
Petición T-DATOS-ACELERADOS Indicación T-DATOS-ACELERADOS	– –	Datos de usuario TS	–
Petición T-DESCONEXIÓN	X	Datos de usuario TS	–
Indicación T-DESCONEXIÓN	X	Motivo Datos de usuario TS	– –
X El protocolo MCS presupone que esta prestación está siempre disponible. – El protocolo MCS no utiliza esta prestación.			

5.4.4 Gestión de testigo

La capa MCS implementa operaciones basadas en testigo en el proveedor MCS superior, con lo que se asegura la coherencia y la exclusividad.

5.5 Procesamiento jerárquico

El procesamiento jerárquico en un dominio se ilustra en la Figura 5-2.

Los nodos en la figura representan proveedores MCS y las flechas etiquetadas representan MCSPDU. Este ejemplo se refiere concretamente a un periodo de tiempo después de que el dominio se ha establecido mediante conexiones entre proveedores MCS y el uso de la transferencia de datos está comenzando a expandirse. En el paso 1 el proveedor D pide en nombre de un usuario incorporarse a un canal a través del cual se distribuirán datos, y en el paso 2 la petición es confirmada como exitosa. En el paso 3 un usuario anexionado al proveedor A envía datos y la correspondiente **SDrq** comienza a fluir hacia arriba. Suponiendo que al canal por el que se están enviando los datos sólo se incorporan anexiones en A, C, y D, la MCSPDU se refleja hacia abajo en los pasos 6 y 7 como **SDin**. El proveedor E, al ver que ningún otro subordinado necesita recibir datos, simplemente envía **SDrq** hacia arriba en el paso 4. El proveedor F envía **SDrq** hacia arriba en el paso 5, pero también la refleja hacia abajo en el paso 6, pues sabe que el proveedor C ha expresado su interés en el canal.

Los proveedores MCS no están especialmente conscientes de su nivel (o altura) en la jerarquía, salvo en lo tocante al papel que desempeñan en la imposición de un límite global a la altura del dominio y, en un amplio sentido, al conocimiento de si son el proveedor superior, o no lo son. El proveedor MCS superior no tiene conexión ascendente. Todos los demás tienen exactamente una.

Un proveedor MCS registra información sobre los canales y testigos utilizados en su subárbol de un dominio MCS.

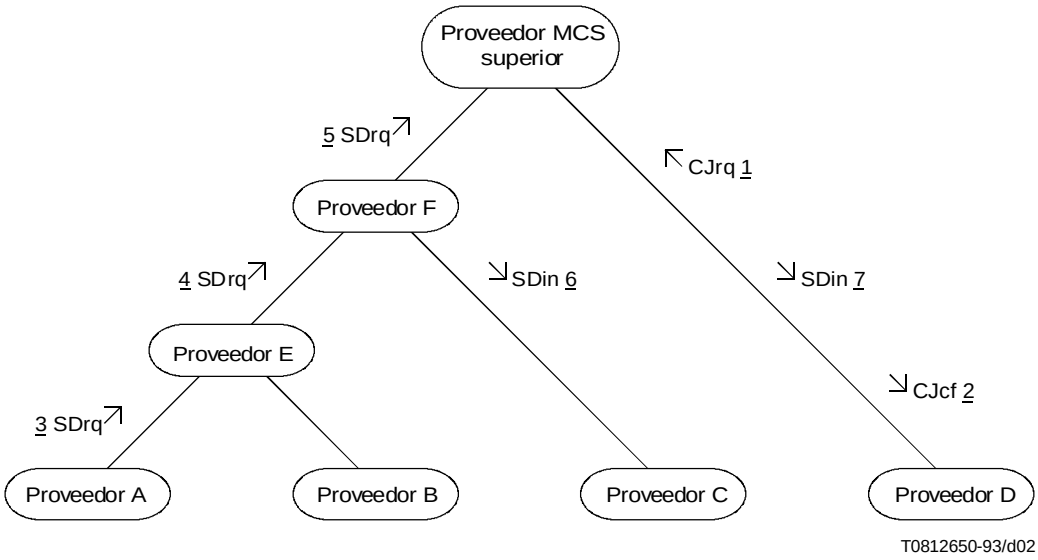


FIGURA 5-1/T.125

Procesamiento jerárquico en un dominio MCS

Un proveedor registra los canales a los que se incorporan usuarios dentro del subárbol y, para cada uno de estos canales, dónde se origina la incorporación, es decir, desde qué anexiones MCS y desde qué conexiones MCS descendentes. Registra los identificadores de usuario que se asignan en el subárbol y dónde se originan. Registra canales privados que tienen un gestor o un usuario admitido en el subárbol y registra los identificadores de los usuarios asociados.

Un proveedor MCS registra los testigos que son tomados o inhibidos por usuarios dentro del subárbol, y registra los identificadores de los usuarios asociados.

Un proveedor MCS examina las peticiones que proceden de su subárbol para verificar que el identificador del usuario iniciador está legítimamente asignado a la anexión o conexión MCS descendente de origen. De esta forma se establecen anillos de protección alrededor del proveedor MCS superior y se limita la cantidad de perturbación que un participante malintencionado puede causar en un dominio que, por lo demás, es cooperativo.

En términos generales, abstracción hecha de varias excepciones, la operación de la capa MCS puede describirse como sigue:

a) Una petición de primitiva MCS invocada en una anexión MCS genera una MCSPDU en el proveedor MCS correspondiente y la despacha en sentido ascendente hacia el proveedor MCS superior. Este proveedor, que tiene la información completa sobre el dominio MCS, actúa sobre la MCSPDU.

b) Se puede generar una MCSPDU de confirmación en el proveedor MCS superior para retornar resultados a la anexión solicitante. Los proveedores MCS que le dan curso actualizan sus registros en función del efecto de la operación sobre el subárbol. Una confirmación se encamina al identificador del usuario iniciador mediante la consulta de registros locales en cada salto sucesivo hacia abajo.

c) En lugar de esto se puede generar una MCSPDU de indicación para informar a otras anexiones sobre la acción ejecutada. Se puede hacer replicaciones de las indicaciones y enviarlas hacia abajo por varias conexiones que conducen a usuarios influidos. Los proveedores MCS pueden también actualizar sus registros, con el efecto de la operación como parte del procesamiento de una indicación.

La descripción precedente es un enunciado general destinado a crear un marco conceptual. Más adelante se dan detalles más completos sobre la información registrada en un proveedor MCS y la manera de procesar MCSPDU específicas.

Entre las excepciones están las siguientes: Algunas peticiones, sobre todo **CJrq** y **CLrq**, pueden dejar de surgir a un nivel a corta distancia por debajo del proveedor superior, y algunas indicaciones, sobre todo **SDin**, a un nivel inferior al del proveedor superior. Una MCSPDU, **TVrs**, pertenece a la categoría de las respuestas. Y algunas MCSPDU pueden ser generadas por un proveedor MCS como una continuación del procesamiento de MCSPDU desiguales como en el caso de una **CLrq** que sigue a una **DUin**.

5.6 Parámetros de dominio

Los proveedores MCS que atienden sólo dominio MCS asignan recursos y ejecutan procedimientos de acuerdo con los siguientes parámetros. Los valores de estos parámetros son idénticos dentro de un dominio.

a) Número máximo de canales MCS que pueden utilizarse simultáneamente. Incluye canales a que se incorporan usuarios, identificadores de usuario que han sido asignados, y canales privados que han sido creados.

b) Número máximo de identificadores de usuario que pueden estar asignados simultáneamente. Este es un sublímite dentro de la restricción del parámetro precedente.

c) Número máximo de los identificadores de testigo que pueden ser tomados o inhibidos simultáneamente.

d) Número de prioridades de transferencia de datos implementadas. Es igual al número de TC en una conexión MCS. Un usuario MCS puede enviar y recibir datos aunque las prioridades estén fuera del límite. Sin embargo, en este caso esas prioridades pueden tratarse como la prioridad más baja que haya sido implementada.

e) Caudal cumplimentado. Aunque el control de flujo global limita la transferencia de datos dentro de un dominio a la velocidad del receptor más lento, no debe permitirse que los receptores trabajen con una velocidad arbitrariamente baja. Si se permitiera esto, un participante en una conferencia podría obstaculizar a todos los demás. Este parámetro obliga a los proveedores MCS a observar una velocidad mínima de recepción en cada anexión MCS y en cada conexión MCS descendente. Los que no cumplen esta obligación corren el riesgo de ser desanexionados involuntariamente o desconectados, respectivamente.

f) Altura máxima. Este parámetro limita la altura de todos los proveedores MCS, en particular la del proveedor superior.

g) Tamaño máximo de las MCSPDU de dominio. El control de flujo global se basa en el almacenamiento de las MCSPDU de dominio (pero no las MCSPDU de conexión) en una memoria tampón dentro de un proveedor MCS. Por razones de sencillez, se supone que las memorias tampón tienen un tamaño fijo. Un proveedor MCS no generará MCSPDU más grandes. Esto limita la cantidad de información que puede estar contenida en una sola MCSPDU de control y sugiere los lugares donde datos de usuario ilimitados pueden segmentarse en varias MCSPDU de datos.

h) Versión del protocolo. Está constituida por uno o dos valores que definen diferentes codificaciones para las MCSPDU de dominio.

NOTA – Una instancia dada de un proveedor MCS puede funcionar con limitaciones de recursos locales que también están parametrizadas. Entre ellas puede estar la cantidad de memoria tampón disponible para el almacenamiento de las MCSPDU que se encuentran en espera de ser transportadas, el número máximo de anexiones MCS, y el número máximo de conexiones MCS a otros proveedores. Estos parámetros se manejan localmente y no se comunican a través de un dominio MCS.

6 Utilización del servicio de transporte

6.1 Modelo del servicio de transporte

Esta descripción se presenta en términos similares a los de las partes pertinentes de la Recomendación X.214 del CCITT, suponiendo que no se utilizan datos acelerados.

El servicio de transporte (TS) ofrece a un usuario TS las siguientes prestaciones:

a) Medios para establecer una TC con otro usuario TS para el intercambio de TSDU. Entre dos mismos usuarios TS puede haber más de una TC.

b) La oportunidad de pedir, negociar, y obtener por acuerdo con el proveedor una cierta calidad de servicio, asociada con cada TC en el momento de su establecimiento y especificada por parámetros que representan características tales como caudal, retardo de tránsito, tasa de error residual, y prioridad.

c) Medio de transferir TSDU en una TC. La transferencia de TSDU, que están constituidas por un número entero de octetos, es transparente, entendiéndose por esto que los confines y el contenido de las TSDU quedan preservados sin modificación por el proveedor TS.

d) Medio por el cual un usuario TS receptor puede controlar la velocidad a la que el usuario TS emisor puede enviar datos.

e) La liberación incondicional, y por consiguiente posiblemente destructiva, de una TC.

La operación de una TC se modela de forma abstracta por un par de colas que enlazan dos TSAP. Hay una cola para cada sentido de flujo de la información.

Este modelo basado en colas se emplea para expresar la prestación de control de flujo. La capacidad de una cola es limitada, pero no tiene necesariamente que ser ni fija ni determinable. Los objetos de conexión, TSDU, y desconexión son introducidos y retirados de una cola como resultado de interacciones en los dos TSAP. La aptitud de un usuario TS para introducir objetos en una cola viene determinada por el comportamiento del usuario TS que retira objetos de esa cola y por el estado de la cola. Los únicos objetos que pueden ser introducidos en una cola por el proveedor TS son los objetos de desconexión. Los objetos se introducen en una cola bajo el control del proveedor TS. Los objetos normalmente se retiran de la cola bajo el control del usuario TS receptor. Normalmente, los objetos se extraen en el mismo orden en que se introdujeron. La única excepción a la extracción normal es que un objeto puede ser suprimido por el proveedor TS única y exclusivamente si el siguiente es un objeto de desconexión.

Hay que proporcionar localmente un mecanismo de identificación de punto extremo de TC si el usuario TS y el proveedor TS necesitan distinguir entre varias TC en un TSAP. En tal caso, todas las primitivas tienen que utilizar este mecanismo de identificación para identificar la TC a que se aplican. Esta identificación implícita no se muestra como un parámetro de las primitivas TS y no debe confundirse con los parámetros de dirección de T-CONEXIÓN.

6.2 Utilización de múltiples conexiones

Una conexión MCS está constituida por una o más TC entre los dos mismos proveedores MCS que forman un par. La primera TC establecida se llama TC inicial; las establecidas ulteriormente se llaman TC adicionales. Todas las TC pertenecientes a una conexión MCS las establece el mismo proveedor MCS, como reacción a una petición MCS-CONEXIÓN-PROVEEDOR. Esta petición contiene parámetros de dirección, que son las direcciones TSAP llamante y llamada. Estas direcciones se utilizan sin modificación alguna en las peticiones T-CONEXIÓN resultantes.

El número de TC por conexión MCS es uniforme en todo un dominio MCS. Este parámetro de dominio es igual al número de niveles de prioridad de transferencia de datos implementados. Se requieren TC separadas porque cada una de ellas es un vehículo para el control de flujo. Los bloqueos en los datos de prioridades más bajas no deben causar retropresiones en perjuicio de datos de más alta prioridad. Para una implementación completa, los datos de prioridades más bajas y los de prioridades más altas deben ser vehiculados por TC diferentes.

La calidad de servicio solicitada para una TC puede variar según la prioridad de datos para la cual se estableció. Los objetivos de calidad de servicio no tienen que ser uniformes en la totalidad de un dominio MCS.

Entre los aspectos de la calidad de servicio que ofrecen interés están el caudal y el retardo de tránsito máximos y medios. Es posible que los datos de alta prioridad requieran un bajo retardo de tránsito para obtener una respuesta en tiempo real, pero que no requieran un caudal elevado. En cambio, es posible que los datos de baja prioridad requieran un caudal elevado, pero que las transferencias masivas no requieran un bajo retardo de tránsito.

La prioridad de la TC es otro aspecto de la calidad de servicio, aunque no se ajuste precisamente al concepto de la prioridad de transferencia de datos MCS. Dicha prioridad especifica el orden relativo en que, en caso necesario, las TC deben ver su calidad de servicio degradada. Una alta prioridad de la TC puede solicitarse junto con otras características, como un bajo retardo de tránsito, para asegurar que los datos prioritarios MCS reciban el tratamiento preferencial que merecen.

Las MCSPDU de conexión sólo aparecen como la primera TSDU transportada en uno u otro sentido de transmisión de una TC. **conexión-inicial** y **conexión-respuesta** atraviesan la TC inicial de una conexión MCS. **conexión-adicional** y **conexión-resultado** atraviesan TC adicionales, si existen.

Un proveedor MCS llamante, al emitir peticiones T-CONEXIÓN, controla mediante sus propias acciones qué TC forman parte de la misma conexión MCS y qué prioridades de transferencia de datos representan.

Un proveedor MCS llamado, al recibir indicaciones T-CONEXIÓN, tiene por lo general que aceptar una TC y leer su primera TSDU para saber lo que significa. **conexión-inicial** identifica una TC entrante como el comienzo de una nueva conexión MCS. Una **conexión-adicional** identifica una TC entrante como parte de una conexión MCS en curso.

Conexión-adicional contiene un valor asignado por el proveedor MCS llamado y transportado al proveedor MCS llamante en una **conexión-respuesta** a través de la TC inicial que designa la TC adicional como perteneciente a la misma conexión MCS. **conexión-adicional** da también explícitamente la prioridad de datos que la TC representa.

Las MCSPDU de conexión se intercambian inmediatamente después del establecimiento de la TC. Una conexión MCS, una vez confirmada, pasa a formar parte de un dominio MCS jerárquico. Después de esto, la conexión MCS transporta MCSPDU de dominio.

Con excepción de las MCSPDU de datos, las MCSPDU de dominio atraviesan la TC inicial de una conexión MCS. Las MCSPDU de datos atraviesan la TC que corresponde a su prioridad de datos. Si la prioridad especificada sobrepasa el número aplicado en un dominio MCS, una MCSPDU de datos atraviesa la TC de la más baja prioridad que ha sido aplicada.

Si sólo se ha aplicado una prioridad en un dominio MCS, cada una de sus conexiones MCS está constituida por una sola TC, no se utiliza **conexión-adicional** ni **conexión-resultado**, y todas las MCSPDU se transmiten en secuencia entre los proveedores.

6.3 Liberación de la conexión de transporte

Un proveedor TS da una mayor fiabilidad a las conexiones de extremo a extremo ejecutando protocolos en grado suficiente para compensar cualquier punto débil en servicios de red subyacentes. Un proveedor MCS no duplica esta funcionalidad. No intenta una ulterior recuperación automática en el caso de un fallo de transporte.

Los errores que no permiten una recuperación (errores «irrecuperables») se anuncian mediante una indicación T-DESCONEXIÓN. Si se desconecta cualquiera de las TC pertenecientes a un MCS, las otras se desconectan también inmediatamente. A menos que esto haya sido solicitado por el usuario, se genera una indicación MCS-DESCONEXIÓN-PROVEEDOR, y se da como motivo iniciada por el proveedor.

Por otra parte, una petición MCS-DESCONEXIÓN-PROVEEDOR debe aparecer como una indicación en el otro lado dándose como motivo solicitada por el usuario. Pese a las seguridades que se dan en la Recomendación X.214, la clase más simple de protocolo de transporte no permite pasar datos de usuario en T-DESCONEXIÓN. En consecuencia, el código de motivo de la desconexión se transfiere en una MCSPDU explícita. Al recibir esta MCSPDU, un proveedor MCS queda obligado a desconectar la conexión MCS que la transportó.

7 Estructura de las MCSPDU

La estructura de las MCSPDU se especifica mediante la notación ASN.1 de la Recomendación X.208 del CCITT. La utilización y el significado de estas MCSPDU se describen más adelante en las cláusulas 9 y 10.

```
MCS-PROTOCOL DEFINITIONS ::=

BEGIN

-- Part 1: Fundamental MCS types

ChannelId ::= INTEGER (0..65535)                -- range is 16 bits

StaticChannelId ::= ChannelId (1..1000)         -- those known permanently

DynamicChannelId ::= ChannelId (1001..65535)    -- those created and deleted

UserId ::= DynamicChannelId                     -- created by Attach-User
                                                -- deleted by Detach-User

PrivateChannelId ::= DynamicChannelId           -- created by Channel-Convene
                                                -- deleted by Channel-Disband

AssignedChannelId ::= DynamicChannelId          -- created by Channel-Join zero
                                                -- deleted by last Channel-Leave

TokenId ::= INTEGER (1..65535)                  -- all are known permanently

TokenStatus ::= ENUMERATED
{
                        notInUse        (0),
                        selfGrabbed     (1),
                        otherGrabbed    (2),
                        selfInhibited   (3),
                        otherInhibited  (4),
                        selfRecipient   (5),
                        selfGiving      (6),
                        otherGiving     (7)

}

DataPriority ::= ENUMERATED
{
                        top             (0),
                        high            (1),
                        medium          (2),
                        low             (3)

}

Segmentation ::= BIT STRING
{
                        begin           (0),
                        end             (1)

} (SIZE (2))
```

```
DomainParameters ::= SEQUENCE
{
        maxChannelIds           INTEGER (0..MAX),
                                -- a limit on channel ids in use,
                                -- static + user id + private + assigned
        maxUserIds              INTEGER (0..MAX),
                                -- a sublimit on user id channels alone
        maxTokenIds             INTEGER (0..MAX),
                                -- a limit on token ids in use
                                -- grabbed + inhibited + giving + ungivable + given
        numPriorities           INTEGER (0..MAX),
                                -- the number of TCs in an MCS connection
        minThroughput           INTEGER (0..MAX),
                                -- the enforced number of octets per second
        maxHeight               INTEGER (0..MAX),
                                -- a limit on the height of a provider
        maxMCSPDUsize           INTEGER (0..MAX),
                                -- an octet limit on domain MCSPDUs
        protocolVersion         INTEGER (0..MAX)
}

-- Part 2: Connect provider

Connect-Initial ::= [APPLICATION 101] IMPLICIT SEQUENCE
{
        callingDomainSelector   OCTET STRING,
        calledDomainSelector    OCTET STRING,
        upwardFlag              BOOLEAN,
                                -- TRUE if called provider is higher
        targetParameters        DomainParameters,
        minimumParameters       DomainParameters,
        maximumParameters       DomainParameters,
        userData                OCTET STRING
}

Connect-Response ::= [APPLICATION 102] IMPLICIT SEQUENCE
{
        result                  Result,
        calledConnectId         INTEGER (0..MAX),
                                -- assigned by the called provider
                                -- to identify additional TCs of
                                -- the same MCS connection
        domainParameters        DomainParameters,
        userData                OCTET STRING
}

Connect-Additional ::= [APPLICATION 103] IMPLICIT SEQUENCE
{
        calledConnectId         INTEGER (0..MAX),
        dataPriority            DataPriority
}

Connect-Result ::= [APPLICATION 104] IMPLICIT SEQUENCE
{
        result                  Result
}

-- Part 3: Merge domain

PDin ::= [APPLICATION 0] IMPLICIT SEQUENCE -- plumb domain indication
{
        heightLimit             INTEGER (0..MAX)
                                -- a restriction on the MCSPDU receiver
}
```

```
EDrq  ::=  [APPLICATION 1] IMPLICIT SEQUENCE  --  erect domain request
{
        subHeight                   INTEGER (0..MAX),
                                    --  height in domain of the MCSPDU transmitter
        subInterval                 INTEGER (0..MAX)
                                    --  its throughput enforcement interval in milliseconds
}

ChannelAttributes  ::=  CHOICE
{
        static                      [0]   IMPLICIT SEQUENCE
        {
            channelId                     StaticChannelId
                                          --  joined is implicitly TRUE
        },
        userId                      [1]   IMPLICIT SEQUENCE
        {
            joined                        BOOLEAN,
            userId                        UserId
                                          --  TRUE if user is joined to its user id
        },
        private                     [2]   IMPLICIT SEQUENCE
        {
            joined                        BOOLEAN,
                                          --  TRUE if channel id is joined below
            channelId                     PrivateChannelId,
            manager                       UserId,
            admitted                      SET OF UserId
                                          --  may span multiple MCrq
        },
        assigned                    [3]   IMPLICIT SEQUENCE
        {
            channelId                     AssignedChannelId
                                          --  joined is implicitly TRUE
        }
}

MCrq  ::=  [APPLICATION 2] IMPLICIT SEQUENCE  --  merge channels request
{
        mergeChannels               SET OF ChannelAttributes,
        purgeChannelIds             SET OF ChannelId
}

MCcf  ::=  [APPLICATION 3] IMPLICIT SEQUENCE  --  merge channels confirm
{
        mergeChannels               SET OF ChannelAttributes,
        purgeChannelIds             SET OF ChannelId
}

PCin  ::=  [APPLICATION 4] IMPLICIT SEQUENCE  --  purge channels indication
{
        detachUserIds               SET OF UserId,
                                    --  purge user id channels
        purgeChannelIds             SET OF ChannelId
                                    --  purge other channels
}
```

```
TokenAttributes ::= CHOICE
{
        grabbed                 [0]   IMPLICIT SEQUENCE
        {
            tokenId                       TokenId,
            grabber                       UserId
        },
        inhibited               [1]   IMPLICIT SEQUENCE
        {
            tokenId                       TokenId,
            inhibitors                    SET OF UserId
                                          -- may span multiple MTrq
        },
        giving                  [2]   IMPLICIT SEQUENCE
        {
            tokenId                       TokenId,
            grabber                       UserId,
            recipient                     UserId
        },
        ungivable               [3]   IMPLICIT SEQUENCE
        {
            tokenId                       TokenId,
            grabber                       UserId
                                          -- recipient has since detached
        },
        given                   [4]   IMPLICIT SEQUENCE
        {
            tokenId                       TokenId,
            recipient                     UserId
                                          -- grabber released or detached
        }
}

MTrq ::= [APPLICATION 5] IMPLICIT SEQUENCE -- merge tokens request
{
        mergeTokens             SET OF TokenAttributes,
        purgeTokenIds           SET OF TokenId
}

MTcf ::= [APPLICATION 6] IMPLICIT SEQUENCE -- merge tokens indication
{
        mergeTokens             SET OF TokenAttributes,
        purgeTokenIds           SET OF TokenId
}

PTin ::= [APPLICATION 7] IMPLICIT SEQUENCE -- purge tokens indication
{
        purgeTokenIds           SET OF TokenId
}

-- Part 4: Disconnect provider

DPum ::= [APPLICATION 8] IMPLICIT SEQUENCE -- disconnect provider ultimatum
{
        reason                  Reason
}

RJum ::= [APPLICATION 9] IMPLICIT SEQUENCE -- reject MCSPDU ultimatum
{
        diagnostic              Diagnostic,
        initialOctets           OCTET STRING
}

-- Part 5: Attach/Detach user

AUrq ::= [APPLICATION 10] IMPLICIT SEQUENCE -- attach user request
{
}
```

```
AUcf  ::=  [APPLICATION 11] IMPLICIT SEQUENCE  --  attach user confirm
{
        result                  Result,
        initiator               UserId OPTIONAL
}

DUrq  ::=  [APPLICATION 12] IMPLICIT SEQUENCE  --  detach user request
{
        reason                  Reason,
        userIds                 SET OF UserId
}

DUin  ::=  [APPLICATION 13] IMPLICIT SEQUENCE  --  detach user indication
{
        reason                  Reason,
        userIds                 SET OF UserId
}

-- Part 6: Channel management

CJrq  ::=  [APPLICATION 14] IMPLICIT SEQUENCE  --  channel join request
{
        initiator               UserId,
        channelId               ChannelId
                                --  may be zero
}

CJcf  ::=  [APPLICATION 15] IMPLICIT SEQUENCE  --  channel join confirm
{
        result                  Result,
        initiator               UserId,
        requested               ChannelId,
                                --  may be zero
        channelId               ChannelId OPTIONAL
}

CLrq  ::=  [APPLICATION 16] IMPLICIT SEQUENCE  --  channel leave request
{
        channelIds              SET OF ChannelId
}

CCrq  ::=  [APPLICATION 17] IMPLICIT SEQUENCE  --  channel convene request
{
        initiator               UserId
}

CCcf  ::=  [APPLICATION 18] IMPLICIT SEQUENCE  --  channel convene confirm
{
        result                  Result,
        initiator               UserId,
        channelId               PrivateChannelId OPTIONAL
}

CDrq  ::=  [APPLICATION 19] IMPLICIT SEQUENCE  --  channel disband request
{
        initiator               UserId,
        channelId               PrivateChannelId
}

CDin  ::=  [APPLICATION 20] IMPLICIT SEQUENCE  --  channel disband indication
{
        channelId               PrivateChannelId
}

CArq  ::=  [APPLICATION 21] IMPLICIT SEQUENCE  --  channel admit request
{
        initiator               UserId,
        channelId               PrivateChannelId,
        userIds                 SET OF UserId
}
```

```
CAin  ::=  [APPLICATION 22] IMPLICIT SEQUENCE  --  channel admit indication
{
        initiator                       UserId,
        channelId                       PrivateChannelId,
        userIds                         SET OF UserId
}

CErq  ::=  [APPLICATION 23] IMPLICIT SEQUENCE  --  channel expel request
{
        initiator                       UserId,
        channelId                       PrivateChannelId,
        userIds                         SET OF UserId
}

CEin  ::=  [APPLICATION 24] IMPLICIT SEQUENCE  --  channel expel indication
{
        channelId                       PrivateChannelId,
        userIds                         SET OF UserId
}

-- Part 7: Data transfer

SDrq  ::=  [APPLICATION 25] IMPLICIT SEQUENCE  --  send data request
{
        initiator                       UserId,
        channelId                       ChannelId,
        dataPriority                    DataPriority,
        segmentation                    Segmentation,
        userData                        OCTET STRING
}

SDin  ::=  [APPLICATION 26] IMPLICIT SEQUENCE  --  send data indication
{
        initiator                       UserId,
        channelId                       ChannelId,
        dataPriority                    DataPriority,
        segmentation                    Segmentation,
        userData                        OCTET STRING
}

USrq  ::=  [APPLICATION 27] IMPLICIT SEQUENCE  --  uniform send data request
{
        initiator                       UserId,
        channelId                       ChannelId,
        dataPriority                    DataPriority,
        segmentation                    Segmentation,
        userData                        OCTET STRING
}

USin  ::=  [APPLICATION 28] IMPLICIT SEQUENCE  --  uniform send data indication
{
        initiator                       UserId,
        channelId                       ChannelId,
        dataPriority                    DataPriority,
        segmentation                    Segmentation,
        userData                        OCTET STRING
}

-- Part 8: Token management

TGrq  ::=  [APPLICATION 29] IMPLICIT SEQUENCE  --  token grab request
{
        initiator                       UserId,
        tokenId                         TokenId
}
```

```
TGcf  ::=  [APPLICATION 30] IMPLICIT SEQUENCE  --  token grab confirm
{
        result                  Result,
        initiator               UserId,
        tokenId                 TokenId,
        tokenStatus             TokenStatus
}

TIrq  ::=  [APPLICATION 31] IMPLICIT SEQUENCE  --  token inhibit request
{
        initiator               UserId,
        tokenId                 TokenId
}

TIcf  ::=  [APPLICATION 32] IMPLICIT SEQUENCE  --  token inhibit confirm
{
        result                  Result,
        initiator               UserId,
        tokenId                 TokenId,
        tokenStatus             TokenStatus
}

TVrq  ::=  [APPLICATION 33] IMPLICIT SEQUENCE  --  token give request
{
        initiator               UserId,
        tokenId                 TokenId,
        recipient               UserId
}

TVin  ::=  [APPLICATION 34] IMPLICIT SEQUENCE  --  token give indication
{
        initiator               UserId,
        tokenId                 TokenId,
        recipient               UserId
}

TVrs  ::=  [APPLICATION 35] IMPLICIT SEQUENCE  --  token give response
{
        result                  Result,
        recipient               UserId,
        tokenId                 TokenId
}

TVcf  ::=  [APPLICATION 36] IMPLICIT SEQUENCE  --  token give confirm
{
        result                  Result,
        initiator               UserId,
        tokenId                 TokenId,
        tokenStatus             TokenStatus
}

TPrq  ::=  [APPLICATION 37] IMPLICIT SEQUENCE  --  token please request
{
        initiator               UserId,
        tokenId                 TokenId
}

TPin  ::=  [APPLICATION 38] IMPLICIT SEQUENCE  --  token please indication
{
        initiator               UserId,
        tokenId                 TokenId
}

TRrq  ::=  [APPLICATION 39] IMPLICIT SEQUENCE  --  token release request
{
        initiator               UserId,
        tokenId                 TokenId
}
```

```
TRcf  ::=  [APPLICATION 40] IMPLICIT SEQUENCE  --  token release confirm
{
        result                  Result,
        initiator               UserId,
        tokenId                 TokenId,
        tokenStatus             TokenStatus
}

TTrq  ::=  [APPLICATION 41] IMPLICIT SEQUENCE  --  token test request
{
        initiator               UserId,
        tokenId                 TokenId
}

TTcf  ::=  [APPLICATION 42] IMPLICIT SEQUENCE  --  token test confirm
{
        initiator               UserId,
        tokenId                 TokenId,
        tokenStatus             TokenStatus
}

-- Part 9: Status codes

Reason  ::=  ENUMERATED         --  in DPum, DUrq, DUin
{
                rn-domain-disconnected          (0),
                rn-provider-initiated           (1),
                rn-token-purged                 (2),
                rn-user-requested               (3),
                rn-channel-purged               (4)
}

Result  ::=  ENUMERATED         --  in Connect, response, confirm
{
                rt-successful                   (0),
                rt-domain-merging               (1),
                rt-domain-not-hierarchical      (2),
                rt-no-such-channel              (3),
                rt-no-such-domain               (4),
                rt-no-such-user                 (5),
                rt-not-admitted                 (6),
                rt-other-user-id                (7),
                rt-parameters-unacceptable      (8),
                rt-token-not-available          (9),
                rt-token-not-possessed          (10),
                rt-too-many-channels            (11),
                rt-too-many-tokens              (12),
                rt-too-many-users               (13),
                rt-unspecified-failure          (14),
                rt-user-rejected                (15)
}

Diagnostic  ::=  ENUMERATED     --  in RJum
{
                dc-inconsistent-merge           (0),
                dc-forbidden-PDU-downward       (1),
                dc-forbidden-PDU-upward         (2),
                dc-invalid-BER-encoding         (3),
                dc-invalid-PER-encoding         (4),
                dc-misrouted-user               (5),
                dc-unrequested-confirm          (6),
                dc-wrong-transport-priority     (7),
                dc-channel-id-conflict          (8),
                dc-token-id-conflict            (9),
```

```
                dc-not-user-id-channel      (10),
                dc-too-many-channels        (11),
                dc-too-many-tokens          (12),
                dc-too-many-users           (13)
}

-- Part 10: MCSPDU repertoire

ConnectMCSPDU ::= CHOICE
{
        connect-initial             Connect-Initial,
        connect-response            Connect-Response,
        connect-additional          Connect-Additional,
        connect-result              Connect-Result
}

DomainMCSPDU ::= CHOICE
{
        pdin            PDin,
        edrq            EDrq,
        mcrq            MCrq,
        mccf            MCcf,
        pcin            PCin,
        mtrq            MTrq,
        mtcf            MTcf,
        ptin            PTin,
        dpum            DPum,
        rjum            RJum,
        aurq            AUrq,
        aucf            AUcf,
        durq            DUrq,
        duin            DUin,
        cjrq            CJrq,
        cjcf            CJcf,
        clrq            CLrq,
        ccrq            CCrq,
        cccf            CCcf,
        cdrq            CDrq,
        cdin            CDin,
        carq            CArq,
        cain            CAin,
        cerq            CErq,
        cein            CEin,
        sdrq            SDrq,
        sdin            SDin,
        usrq            USrq,
        usin            USin,
        tgrq            TGrq,
        tgcf            TGcf,
        tirq            TIrq,
        ticf            TIcf,
        tvrq            TVrq,
        tvin            TVin,
        tvrs            TVrs,
        tvcf            TVcf,
        tprq            TPrq,
        tpin            TPin,
        trrq            TRrq,
        trcf            TRcf,
        ttrq            TTrq,
        ttcf            TTcf
}

END
```

8 Codificación de las MCSPDU

Cada MCSPDU se transporta como una TSDU a través de una TC perteneciente a una conexión MCS. Las MCSPDU de conexión tienen un tamaño ilimitado. Las MCSPDU de dominio tienen su tamaño limitado por un parámetro del dominio MCS.

Para transferir MCSPDU entre proveedores del mismo rango se utiliza una codificación normalizada de valor de datos ASN.1. Se definen dos versiones de este protocolo, las cuales sólo se diferencian en la especificación de las reglas de codificación:

– *Versión 1* – Utiliza las reglas de codificación básica de la Recomendación X.209 del CCITT para todas las MCSPDU.

– *Versión 2* – Utiliza las reglas de codificación básica para las MCSPDU de conexión y las reglas de codificación empaquetada de ISO/CEI 8825-2 para todas las MCSPDU de dominio subsiguientes. Específicamente, la variante ALIGNED de BASIC-PER se aplicará al tipo ASN.1 **DomainMCSPDU**. La cadena de bits producida se transportará como un número entero de octetos. El bit inicial de esta cadena de bits coincidirá con el bit más significativo del primer octeto.

En el Apéndice I se presenta un ejemplo de una MCSPDU de transferencia de datos con las codificaciones alternativas de las versiones 1 y 2.

La negociación de la versión de protocolo comprende el intercambio de una MCSPDU de **conexión-inicial** y de una MCSPDU de **conexión-respuesta** a través de la TC inicial. Estas dos MCSPDU siempre se codifican según las reglas de codificación básica. De la misma forma se codifica cualquier MCSPDU de **conexión-adicional** o de **conexión-resultado** que sigan, antes de que empiecen las MCSPDU de dominio. Las MCSPDU de dominio empiezan con la segunda TSDU transmitida por una TC.

La versión 2 de este protocolo no se utilizará hasta que las reglas de codificación empaquetada hayan sido adoptadas como parte de una Recomendación del UIT-T o de una Norma Internacional ISO/CEI.

NOTAS

1 Las reglas de codificación empaquetada permiten obtener encabezamientos de MCSPDU más compactos.

2 Tanto las reglas de codificación básica (BER) como las reglas de codificación empaquetada (PER) son autolimitativas, en el sentido de que contienen información suficiente para localizar el final de cada MCSPDU codificada. Pudiera alegarse que esto hace innecesario el uso de TSDU y que este protocolo podría implementarse a través de servicios de transporte no normalizados que transporten trenes de octetos sin preservar los confines de las TSDU. No obstante, ese método es más vulnerable a errores de implementación. Si en cualquier circunstancia se perdieran las demarcaciones entre las MCSPDU, una recuperación sería difícil.

9 Encaminamiento de las MCSPDU

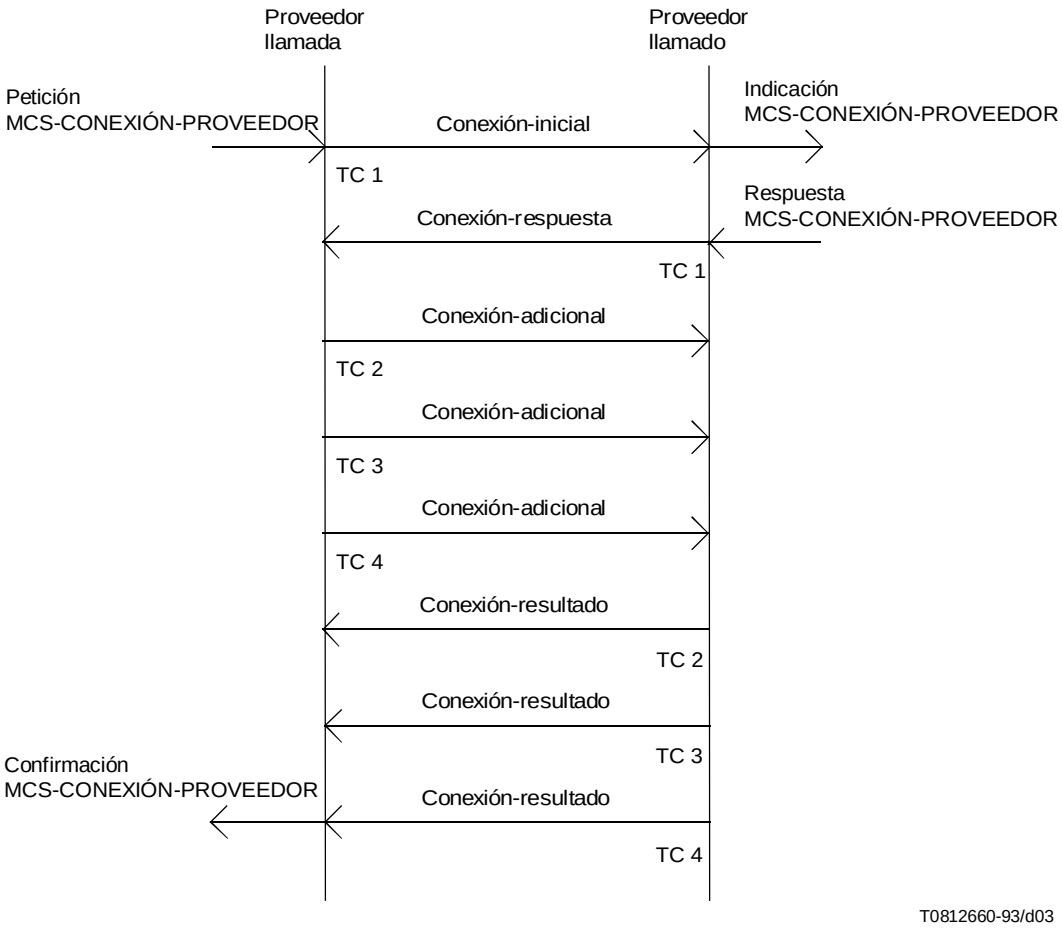
9.1 MCSPDU de conexión

La Figura 9-1 especifica el intercambio de MCSPDU de conexión.

Al recibir una **conexión-respuesta**, el proveedor MCS llamante se entera del valor negociado para el número de prioridades de transferencia de datos implementadas en el dominio. Con fines de ilustración, las MCSPDU de conexión en TC adicionales se han representado siguiendo la secuencia precisa de 2, 3, 4 en el proveedor MCS llamado. En realidad, las conexiones de transporte puede que no se establezcan en el mismo orden en que fueron pedidas. En efecto, una **conexión-adicional** puede llegar en una posición diferente de aquélla en que fue enviada, lo que hace que se retorne una **conexión-resultado** también fuera de secuencia. Puede suceder asimismo que esta última, aunque haya sido enviada en secuencia, aparezca en un orden diferente en tránsito. Un proveedor MCS llamante no tiene que esperar a que se establezcan todas las TC adicionales para enviar la primera **conexión-adicional**. El proveedor MCS llamado no tiene que esperar a que llegue un conjunto completo de MCSPDU de **conexión-adicional** para retornar la primera **conexión-resultado**. La llegada de un conjunto completo de resultados de éxito al proveedor MCS llamante, cualquiera que sea el orden en que se reciban, provoca la generación de una confirmación MCS-CONEXIÓN-PROVEEDOR de éxito.

Una **conexión-respuesta** o **conexión-resultado** de fracaso o una indicación T-DESCONEXIÓN en algún punto intermedio traen consigo la desconexión de todas las TC que hasta ese momento pertenezcan a la conexión MCS y la generación de una confirmación MCS-CONEXIÓN-PROVEEDOR de fracaso.

Una petición MCS-CONEXIÓN-PROVEEDOR especifica cuál de dos proveedores MCS es el más alto. Esta relación jerárquica determina el ulterior encaminamiento de las MCSPDU de dominio, después de lo cual la distinción entre proveedor MCS llamante y llamado es intrascendente. Por ejemplo, como paso siguiente la capa MCS deberá fusionar los recursos de dos dominios que antes eran independientes. El proveedor MCS más bajo genera **MCrq** y **MTrq**, las que se transmiten al proveedor MCS más alto a través de la nueva conexión MCS. Estas MCSPDU pueden transmitirse en el sentido llamante a llamado o en el opuesto, lo que depende de cómo se haya fijado la bandera ascendente.



NOTA – El número y el orden relativo de las TC adicionales puede variar.

FIGURA 9-1/T.125

Flujo de mensajes de las MCSPDU de conexión

9.2 MCSPDU de dominio

El Cuadro 9-1 especifica el encaminamiento de las MCSPDU de dominio.

Si un proveedor MCS genera o reenvía una MCSPDU de categoría *petición*, dicha unidad se transmite en sentido ascendente a través de la conexión MCS única. **EDrq**, **CJrq**, y **CLrq** pueden ser consumidas en algún proveedor MCS intermedio. Otras peticiones suben para ser tratadas por el proveedor MCS superior, a menos que se determine que el contenido de una petición no es válido, en cuyo caso puede ignorarse su MCSPDU, sin confirmación.

CUADRO 9-1/T.125

Encaminamiento de las MCSPDU de dominio

Categoría	MCSPDU			TC	Sentido
Petición	**EDrq**	**MCrq**	**MTrq**	I	Hacia arriba
	AUrq	**DUrq**			
	CJrq	**CLrq**	**CCrq**		
	CDrq	**CArq**	**CErq**		
	TGrq	**TIrq**	**TVrq**		
	TPrq	**TRrq**	**TTrq**		
	SDrq	**USrq**		A	
Indicación	**PDin**	**PCin**	**PTin**	I	Hacia abajo
		DUin			
	CDin	**CAin**	**CEin**		
			TVin		
	TPin				
	SDin	**USin**		A	
Respuesta			**TVrs**	I	Hacia arriba
Confirmación		**MCcf**	**MTcf**	I	Hacia abajo
	AUcf				
	CJcf		**CCcf**		
	TGcf	**TIcf**	**TVcf**		
		TRcf	**TTCF**		
Ultimátum	**DPum**	**RJum**		I	Hacia arriba o abajo

I	La MCSPDU atraviesa la TC inicial.
A	La MCSPDU puede atravesar una TC adicional.
Hacia arriba	La MCSPDU se desplaza hacia el proveedor MCS superior.
Hacia abajo	La MCSPDU se desplaza alejándose del proveedor MCS superior.

Si un proveedor MCS genera o reenvía una MCSPDU de categoría *indicación*, copias de la misma, posiblemente con un contenido modificado, se transmiten en sentido descendente a través de cero o más conexiones MCS, de acuerdo con las siguientes reglas:

a) **PDin** se envía hacia abajo en todas las conexiones MCS. El límite de altura que ella contiene se decrementa en una unidad. Un proveedor MCS que reciba esta MCSPDU con un límite de altura de cero deberá desconectar.

b) **PCin** se envía hacia abajo en todas las conexiones MCS. El conjunto de identificadores de usuario reenviados no se modifica, de modo que todos los usuarios desanexionados serán anunciados a los que permanecen. El conjunto de otros identificadores de canal reenviados puede ser restringido a los utilizados en un subárbol. En un antiguo proveedor superior que está todavía fusionándose con un dominio más alto, ambos identificadores de usuario y otros identificadores de canal están restringidos a aquellos cuya aceptación en el dominio superior ha sido confirmada.

c) **PTin** se envía hacia abajo en todas las conexiones MCS. El conjunto de identificadores de testigo reenviados puede restringirse a los utilizados en un subárbol. En un proveedor superior anterior que está todavía fusionando con un dominio superior, los identificadores de testigo están restringidos a aquellos cuya aceptación en el dominio superior ha sido confirmada.

d) **DUin** se envía hacia abajo en todas las conexiones MCS. El conjunto de identificadores de usuario reenviados se mantiene sin modificación, de modo que todos los usuarios desanexionados serán anunciados a los usuarios que quedan. En un proveedor superior anterior que está aún fusionando con un dominio superior, los identificadores de usuario están restringidos a aquellos cuya aceptación en el dominio superior ha sido confirmada.

e) **CDin** se envía hacia abajo en todas las conexiones que contienen en su subárbol el gestor del canal privado o cualquier usuario admitido.

f) **CAin** y **CEin** se envían hacia abajo en todas las conexiones MCS que contienen uno o más de los usuarios influidos en su subárbol. El conjunto de los identificadores de usuario puede limitarse a los residentes en un subárbol.

g) **TVin** se envía hacia abajo en una sola conexión MCS que contiene el receptor designado en su subárbol.

h) **TPin** se envía hacia abajo en todas las conexiones MCS que contienen en su subárbol un usuario que ha tomado o inhibido el testigo, o al que se le está cediendo el testigo.

i) **SDin** y **USin** se envían hacia abajo en todas las conexiones MCS por las cuales se efectúa una incorporación al canal especificado, excepto que, cuando se genera **SDin**, dicha unidad no se devuelve en la conexión por la que llegó **SDrq**.

No es necesario reenviar las indicaciones **PCin**, **PTin**, **DUin**, **CAin**, y **CEin** si los conjuntos de identificadores que contienen están vacíos.

Si un proveedor MCS genera o reenvía una MCSPDU de categoría *respuesta*, dicha unidad se desplaza hacia arriba a través de la conexión MCS única. La unidad sube para ser tratada por el proveedor MCS superior, a menos que se determine que su contenido es inválido.

Si un proveedor MCS genera o reenvía una MCSPDU de categoría *confirmación*, dicha unidad desciende por una sola conexión MCS, de acuerdo con las siguientes reglas:

a) **MCcf** recorre, en sentido opuesto, el mismo camino de la **MCrq** más antigua que no haya sido contestada por una confirmación. Para esto se requiere que cada proveedor MCS mantenga una cola de tipo primero en entrar primero en salir para las peticiones pendientes.

b) **MTcf** recorre, en sentido opuesto, el mismo camino de la **MTrq** más antigua que no haya sido contestada por una confirmación. Para esto se requiere que cada proveedor MCS mantenga una cola de tipo primero en entrar primero en salir para las peticiones pendientes.

c) **AUcf** recorre, en sentido opuesto, el mismo camino de la **AUrq** más antigua que no haya sido contestada por una confirmación. Si hay pendientes más de una **AUrq**, no es crítico el que sea una u otra; no obstante, por razones de equidad, cada proveedor debe mantener una cola de tipo primero en entrar primero en salir. Después de enviar **AUcf**, un proveedor MCS registrará a qué subárbol se asigna de esa forma el identificador de usuario contenido en ella.

d) Otras MCSPDU de categoría confirmación contienen un identificador de usuario iniciador que fue anteriormente asignado por la acción de **AUcf** como se ha explicado antes. Estas MCSPDU se envían hacia abajo por la conexión MCS que conduce al subárbol donde fue asignado el identificador de usuario. Estas unidades continúan de esta manera y llegan finalmente al proveedor que atiende la afectación MCS solicitante.

Se generan confirmaciones en el curso del procesamiento de peticiones iguales. Todas las confirmaciones con excepción de **CJcf** las genera el proveedor MCS superior.

Si un proveedor MCS genera una MCSPDU de categoría *ultimátum*, ésta se transmite por una conexión MCS única en sentido ascendente o descendente. **DPum** ordena al proveedor MCS receptor que desconecte la conexión que la transporta. **RJum** rechaza una MCSPDU errónea con un código de diagnóstico e invita al proveedor MCS que la transmitió a desconectar. Los ultimátum no se reenvían.

10 Significado de las MCSPDU

En los Cuadros 10-1 a 10-47 se reitera el contenido de las MCSPDU definidas en la cláusula 8.

10.1 Conexión-inicial

Una **conexión-inicial** es generada por una petición MCS-CONEXIÓN-PROVEEDOR. Se envía como la primera TSDU a través de la TC inicial de una nueva conexión MCS. En el receptor, genera una indicación MCS-CONEXIÓN-PROVEEDOR.

CUADRO 10-1/T.125

MCSPDU de conexión inicial

Contenido	Fuente	Sumidero
Selector del dominio llamante	Petición	Indicación
Selector del dominio llamado	Petición	Indicación
Bandera ascendente	Petición	Indicación
Parámetros de dominio deseados	Petición	Indicación
Parámetros de dominio mínimos	Petición	Indicación
Parámetros de dominio máximos	Petición	Indicación
Datos de usuario	Petición	Indicación

La dirección de transporte llamante y la dirección de transporte llamada son parámetros adicionales de la petición y de la indicación MCS-CONEXIÓN-PROVEEDOR. Estas direcciones se convierten en parámetros de T-CONEXIÓN y no se pasan explícitamente en ninguna MCSPDU. El mismo par de direcciones de transporte se utilizará para pedir todas las TC pertenecientes a la misma conexión MCS.

La calidad de servicio de transporte es un parámetro adicional de la petición MCS-CONEXIÓN-PROVEEDOR pero no de la indicación. La calidad de servicio puede variar de una TC a otra, y la calidad disponible sólo viene a conocerse en el curso del establecimiento de una TC dada. Como el número de TC adicionales que se necesitan no se conoce hasta que MCS-CONEXIÓN-PROVEEDOR haya negociado el parámetro de dominio para el número de prioridades de datos implementadas, esta primitiva no puede negociar completamente, al mismo tiempo, su calidad de servicio de transporte. Un proveedor MCS llamado deberá por tanto aceptar automáticamente las TC entrantes con la calidad de servicio ofrecida mientras ésta satisfaga cualquier valor mínimo especificado facultativamente por el proveedor MCS llamante e indicado a través de cada una de las T-CONEXIÓN.

La interpretación de valores de selector de dominio es un asunto local y la determina cada proveedor MCS. Dichos valores están constituidos por cadenas de octetos que tienen las características de una dirección. Se puede determinar valores aceptables mediante el proceso de configuración de un proveedor MCS. Más de un valor pueden seleccionar el mismo dominio. Un selector de dominio no especificado es una cadena de octetos de longitud cero. Por convenio local, puede representarse por algún valor explícito.

La bandera ascendente especifica el sentido de transmisión de una nueva conexión MCS; se pone a verdadero si el proveedor llamado ha de considerarse más alto que el proveedor llamante y se pone a falso en los demás casos. Un proveedor MCS desempeña un cometido en la jerarquía de un dominio basado en el sentido de las conexiones MCS en las que participa. Ningún proveedor permitirá dos conexiones a proveedores más altos. Un proveedor sin una conexión a un proveedor más alto actuará como el proveedor MCS superior.

Los parámetros de dominio deseados de **conexión-inicial** deberán, cada uno de ellos, estar comprendidos entre los valores mínimo y máximo especificados. Un proveedor MCS modificará los parámetros de dominio solicitados para reflejar límites de su implementación o para imponer valores ya convenidos entre los miembros existentes de un dominio. Puede aumentar los mínimos y reducir los máximos. Deberá cambiar los valores deseados con el solo objeto de mantenerlos dentro del intervalo. Un proveedor MCS tiene que estar preparado para satisfacer cualquier respuesta comprendida en la gama de valores que propone.

Los datos de usuario están constituidos por una cadena de octetos de longitud arbitraria. La longitud puede ser cero.

Un proveedor MCS acepta automáticamente cada TC entrante hasta el límite de su capacidad. Los datos de un usuario en T-CONEXIÓN no se utilizan. La primera TSDU recibida en una transferencia de datos, sea una MCSPDU de **conexión-inicial**, o de **conexión-adicional**, determina la naturaleza de la TC. Si el contenido es inaceptable, un proveedor MCS llamado puede desconectar la TC inmediatamente. La reacción preferida es retornar una **conexión-respuesta** o una **conexión-resultado**, según el caso, con la explicación de la causa por la cual fracasó la conexión MCS. Después de esto, el proveedor MCS llamante desconectará.

10.2 Conexión-respuesta

Una **conexión-respuesta** se genera por una respuesta MCS-CONEXIÓN-PROVEEDOR. Es la primera TSDU enviada en sentido de retorno a través de la TC inicial de una nueva conexión MCS. Transporta la aceptación de una conexión MCS al proveedor MCS llamante, quien procede entonces a establecer las TC adicionales que se necesiten.

CUADRO 10-2/T.125

MCSPDU de conexión-respuesta

Contenido	Fuente	Sumidero
Resultado	Respuesta	Confirmación
Parámetros de dominio	Respuesta	Confirmación
Id de conexión llamada	Proveedor llamado	Proveedor llamante
Datos de usuario	Respuesta	Confirmación

Si el resultado es de éxito, esta MCSPDU fija los parámetros de dominio en vigor. Entre éstos se encuentra el número de prioridades de transferencia de datos implementadas, que es igual al número de TC en una conexión MCS. Si este número es mayor que uno, hay que crear TC adicionales y vincularlas a la conexión MCS mediante el intercambio de MCSPDU de **conexión-adicional** y de **conexión-resultado**.

El identificador de conexión llamada sirve de medio para asociar TC entrantes adicionales en el proveedor MCS llamado con esta TC inicial. Su valor se elige exclusivamente para esta finalidad. Deberá identificar unívocamente una conexión MCS en curso en el proveedor llamado. Este identificador no tiene un significado duradero tras la compleción de MCS-CONEXIÓN-PROVEEDOR.

La mayor parte de los parámetros de la confirmación MCS-CONEXIÓN-PROVEEDOR se transportan en **conexión-respuesta**. Si el resultado es de fracaso o no se necesitan TC adicionales, la confirmación se genera inmediatamente. En otro caso, se difiere hasta que se conozcan los resultados para vincular TC adicionales a la conexión MCS.

10.3 Conexión-adicional

Se genera **conexión-adicional** tras la recepción de **conexión-respuesta.** Se envía como la primera TSDU a través de una TC adicional o de una nueva conexión MCS.

La prioridad de datos toma los valores *alta*, *media*, y *baja*, en secuencia, hasta el número de TC adicionales requeridas.

CUADRO 10-3/T.125

MCSPDU de conexión-adicional

Contenido	Fuente	Sumidero
Id de conexión llamada	Proveedor llamante	Proveedor llamado
Prioridad de datos	Proveedor llamante	Proveedor llamado

10.4 Conexión-resultado

Se genera **conexión-resultado** tras la recepción de **conexión-adicional**. Es la primera TSDU enviada en sentido de retorno por una TC adicional o una nueva conexión MCS.

CUADRO 10-4/T.125

MCSPDU de conexión-resultado

Contenido	Fuente	Sumidero
Resultado	Proveedor llamado	Confirmación

Si cualquier resultado es de fracaso, se generará una confirmación MCS-CONEXIÓN-PROVEEDOR inmediatamente. Todas las TC asociadas con la conexión MCS serán desconectadas y se ignorarán las MCSPDU que ellas transporten.

En otro caso, se esperarán resultados de éxito para cada TC adicional. Estos resultados podrán retornar fuera de secuencia. Una vez que se hayan obtenido todos, se generará una confirmación MCS-CONEXIÓN-PROVEEDOR de éxito.

Tras una confirmación MCS-CONEXIÓN-PROVEEDOR, podrán pasar MCSPDU de dominio a través de una conexión MCS. Cada conexión MCS pertenece a un solo dominio. En configuraciones donde un proveedor MCS atiende más de un dominio, la conexión MCS que conduce MCSPDU determina el dominio a que se aplican. Las descripciones de MCSPDU de dominio que se tratan en el resto de esta cláusula se sitúan en el contexto de un dominio único.

10.5 PDin

Se genera **PDin** tras la compleción exitosa de MCS-CONEXIÓN-PROVEEDOR. Esta unidad «hermetiza» la jerarquía de proveedores MCS por debajo de una nueva conexión MCS para asegurar que no se haya creado ningún ciclo. **PDin** la genera también el proveedor MCS superior para imponer la máxima altura de un dominio.

CUADRO 10-5/T.125

MCSPDU PDin

Contenido	Fuente	Sumidero
Límite de altura	Superior antiguo o actual	Subordinados

PDin la genera en el extremo inferior de una nueva conexión MCS el proveedor que ha dejado de ser el superior de un dominio. Su contenido se inicializa al parámetro de dominio para la altura máxima del dominio. **PDin** se transmite en sentido descendente a través de todas las conexiones MCS.

Según se necesite, **PDin** se genera de la misma manera en el proveedor MCS superior.

Dondequiera que se reciba **PDin**, se examina el límite de altura en ella contenido. Si el límite es mayor que cero se decrementa en una unidad, y se reenvía **PDin** en sentido descendente por todas las conexiones MCS. Por otra parte, un valor cero significa que el receptor está demasiado lejos del proveedor superior. Deberá reaccionar desconectando la conexión MCS ascendente. Con esto se suprime un subárbol en su totalidad y se ayuda a reparar la altura del dominio.

En presencia de un ciclo, el límite de altura tiene que disminuir hasta que llegue a cero. El proveedor que detecta esto rompe el ciclo, para lo cual deberá suprimir todos los proveedores que intervienen en el ciclo, y sus subordinados, desde el dominio.

NOTA – Un proveedor MCS que sólo cuente con el conocimiento local de las conexiones MCS no puede impedir la creación de ciclos. Puede asegurar que haya en todo momento, cuando más, una conexión ascendente, pero no puede asegurar que la conexión ascendente no forme bucle con algún proveedor inferior. Cuando se crea un ciclo, la causa inmediata de ello es una conexión ascendente defectuosa desde el proveedor MCS superior. Las aplicaciones de controlador, que especifican las conexiones MCS que habrán de crearse, deberán tratar de evitar esos errores.

10.6 EDrq

Se genera **EDrq** tras la compleción exitosa de MCS-CONEXIÓN-PROVEEDOR. Esta unidad se transmite en sentido ascendente y comunica cambios en la altura de proveedores y sus intervalos de cumplimentación del caudal. Un proveedor MCS genera **EDrq** cada vez que su altura o intervalo cambian.

La altura de un proveedor MCS puede cambiar cuando se añade o se retira una conexión MCS, o cuando un proveedor subordinado comunica un cambio mediante una **EDrq**. Su intervalo de supervisión para cumplimentar el caudal mínimo especificado como un parámetro de dominio puede cambiar al adaptarse a los intervalos comunicados por subordinados, o por otras razones. Si cualquiera de estos valores cambia, un proveedor MCS transmitirá **EDrq** a su superior inmediato.

CUADRO 10-6/T.125

MCSPDU EDrq

Contenido	Fuente	Sumidero
Altura en el dominio	Subordinado	Proveedor más alto
Intervalo de cumplimentación del caudal	Subordinado	Proveedor más alto

10.7 MCrq

Se genera **MCrq** tras la compleción exitosa de MCS-CONEXIÓN-PROVEEDOR. Esta unidad comunica en sentido ascendente los atributos de canales que tenían un antiguo proveedor superior, para poder incorporarlos a un dominio fusionado.

CUADRO 10-7/T.125

MCSPDU MCrq

Contenido	Fuente	Sumidero
Fusión de canales	Antiguo superior	Proveedor superior
Purga de identificadores de canal	Intermedios	Proveedor superior

La **MCrq** puede ser rellenada con los atributos de múltiples canales, hasta el límite impuesto por el dominio al tamaño de las MCSPDU. Como se indica detalladamente en las definiciones ASN.1 de la cláusula 7, cada uno de los cuatro tipos de canales que se están utilizando (estático, identificador de usuario, privado, asignado) tiene su correspondiente conjunto de atributos. Estos atributos están contenidos en la base de información del proveedor MCS superior y están parcialmente replicados en los subárboles donde se utiliza un canal. Cuando se fusionan dos dominios, mediante MCS-CONEXIÓN-PROVEEDOR, los canales que se están utilizando en el dominio más bajo tienen que ser incorporados a la base de información del proveedor más alto, o eliminados (purgados) del proveedor más bajo. Esta decisión incumbe al proveedor superior del dominio fusionado.

Cada canal deberá considerarse individualmente. Si los límites impuestos por el dominio a los canales que se están utilizando lo permiten y no se está dando ya al identificador de canal un uso conflictivo, el dominio más alto deberá expandirse hasta incluirlo. La utilización de un identificador de canal estático nunca da lugar a conflicto. La utilización de un identificador de canal privado en el dominio más bajo no da lugar a conflicto si se utiliza también como identificador de canal privado en el dominio más alto y tiene el mismo identificador de usuario como gestor. Todas las demás combinaciones de uso simultáneo están desautorizadas, y el identificador de canal deberá ser purgado del dominio más bajo.

Si un canal privado tiene un extenso conjunto de usuarios admitidos, sus atributos pudieran no caber en una sola **MCrq** y deberán enviarse hacia arriba en múltiples MCSPDU. Sin embargo, la segunda petición de fusionar el mismo canal privado, y las siguientes, deberán aplazarse hasta que se haya recibido una **MCcf** en respuesta a la primera. Sólo entonces se sabrá si los límites impuestos por el dominio han permitido que se ponga en uso el canal en el dominio más alto. Si la primera petición fracasa, no se repetirá con un subconjunto restante de usuarios admitidos.

Cada **MCrq** provoca una contestación del proveedor MCS superior mediante **MCcf**, en la misma secuencia. Una **MCcf** no contiene nada que identifica explícitamente la **MCrq** precedente. Las contestaciones se encaminarán basándose exclusivamente en el orden en que se recibieron estas MCSPDU. Los proveedores MCS por encima del proveedor superior tomarán nota de cada **MCrq** no contestada, y de su procedencia, de manera que se pueda retornar la correspondiente **MCcf** a través de la misma conexión MCS.

Los proveedores MCS intermedios validarán los identificadores de usuario enunciados en atributos de canales privados, para asegurar que se asignen legítimamente al subárbol en que origina la **MCrq**. Se suprimirán los identificadores de usuario inválidos. Si se suprime el gestor de un canal privado, todos los atributos de canal deberán suprimirse en la petición de fusión y solamente el identificador de canal se incluirá en el conjunto que habrá de purgarse. Salvo esta validación de identificadores de usuario, los proveedores MCS intermedios no modificarán el contenido de una **MCrq**.

Un antiguo proveedor superior esperará la confirmación individual de que todos los identificadores de usuario y de todos los identificadores de testigo hayan sido incorporados al dominio fusionado, o purgados, antes de empezar a someter atributos de canal estático, asignado o privado, para la fusión.

10.8 MCcf

Una **MCcf** contesta a una **MCrq** precedente. Refleja el mismo conjunto de identificadores de canal y un subconjunto de los atributos. Los atributos de canal no incorporados en el dominio fusionado se comunican como identificadores de canal para ser purgados.

Los identificadores de canal aceptados se reflejan con los atributos que se introdujeron en la base de información del proveedor MCS superior. Los proveedores intermedios actualizarán su base de información para hacerla conforme.

CUADRO 10-8/T.125

MCSPDU MCcf

Contenido	Fuente	Sumidero
Fusión de canales	Proveedor superior	Intermedios
Purga de identificadores de canal	Proveedor superior	Antiguo proveedor superior

Los canales que van a ser purgados del dominio más bajo se listan por los identificadores solamente. Si los mismos identificadores de canal se utilizaron en el dominio más alto, se mantienen sin modificación. Los proveedores intermedios reenviarán los identificadores de canal purgados sin tratarlos.

Los proveedores MCS encaminarán **MCcf** a la fuente de la **MCrq** precedente basándose en el conocimiento de que hay una contestación de uno a uno. **MCcf** retorna al antiguo proveedor superior que generó **MCrq**. Allí los canales fusionados pueden ser ignorados, ya que han permanecido en la base de información mientras está pendiente una contestación. Los identificadores de canal purgados se suprimirán, ya que estaban destinados a **PCin**.

Los proveedores MCS intermedios confirmarán que los identificadores de usuario anunciados en atributos de canal privado se asignan al subárbol al cual se encamina **MCcf**. Si un gestor de canal privado ha sido desanexionado y reasignado en algún otro lugar en el tiempo transcurrido desde que la **MCrq** precedente fue validada, un proveedor intermedio generará una **CDrq** haciendo del canal privado una víctima de la fusión de dominio y trasladará el identificador de canal al conjunto purgado. Si cualesquiera usuarios admitidos han sido reasignados en algún otro lugar, los excluirá del canal.

10.9 PCin

PCin se genera en un antiguo proveedor superior tras la recepción de **MCcf**. Se difunde en sentido descendente y purga el uso de identificadores de canal especificados eliminándolos de los proveedores subordinados.

CUADRO 10-9/T.125

MCSPDU PCin

Contenido	Fuente	Sumidero
Identificadores de canal para afectación	Antiguo superior	Subordinados
Identificadores de canal para abandono	Antiguo superior	Subordinados
Identificadores de canal para disolución	Antiguo superior	Subordinados

De acuerdo con el uso que se esté haciendo de un identificador de canal, el efecto de su purga es el siguiente: indicación MCS-DESANEXIÓN-USUARIO a todos los usuarios de un canal de identificador de usuario; indicación MCS-ABANDONO-CANAL a los usuarios incorporados de un identificador de canal estático o asignado; indicación MCS-DISOLUCIÓN-CANAL al gestor, e indicación MCS-EXCLUSIÓN-CANAL a los usuarios admitidos de un identificador de canal privado.

Un proveedor superior anterior que conoce la utilización de todos los canales en su dominio más bajo, puede generar las indicaciones apropiadas a partir solamente de sus identificadores de canal. Sin embargo, sus subordinados sólo pueden tener un conocimiento parcial. Hay que informarles qué identificadores de canal representan a usuarios desanexionados, para los cuales se genera siempre una indicación, y cuáles representan a otras clases de canales, para los cuales se genera solamente una indicación si el canal está en uso en el proveedor subordinado. Por tanto, **PCin** divide los identificadores de canal que han de purgarse en estas dos categorías.

La purga de un identificador de usuario mediante **PCin** tendrá las mismas consecuencias que su supresión mediante **DUin**, excepto que, como el receptor de **PCin** ya no es el proveedor superior, no tiene necesidad de generar **CDin** o **TVcf** como efecto secundario.

NOTA – Un proveedor puede recibir en **PCin** identificadores de canal estático y de canal asignado a los que no se ha incorporado o identificadores de canales privados para los cuales sus afectaciones no son ni gestores ni usuarios admitidos. Registros de utilización mantenidos en la base de información permiten a un proveedor suprimir indicaciones de primitivas para esos identificadores de canal.

10.10 MTrq

Se genera una **MTrq** tras la compleción exitosa de MCS-CONEXIÓN-PROVEEDOR. Dicha unidad comunica en sentido ascendente los atributos de testigos poseídos por un antiguo poseedor superior de modo que puedan ser incorporados al dominio fusionado.

CUADRO 10-10/T.125

MCSPDU MTrq

Contenido	Fuente	Sumidero
Fusión de testigos	Antiguo superior	Proveedor superior
Purga de identificadores de testigo	Intermedios	Proveedor superior

La **MTrq** puede ser rellenada con los atributos de múltiples testigos, hasta el límite impuesto por el dominio al tamaño de las MCSPDU. Como se indica detalladamente en las definiciones ASN.1 de la cláusula 7, cada uno de los tipos de testigos que se están utilizando (tomado, inhibido, en cesión, no cedible, cedido) tiene su correspondiente conjunto de atributos. Estos atributos están contenidos en la base de información del proveedor MCS superior y están parcialmente replicados en los subárboles donde se utiliza un testigo. Cuando se fusionan dos dominios, mediante MCS-CONEXIÓN-PROVEEDOR, los testigos que se están utilizando en el dominio más bajo tienen que ser incorporados a la base de información del dominio más alto, o purgados del dominio más bajo. Esta decisión incumbe al proveedor superior del dominio fusionado.

Cada testigo deberá considerarse individualmente. Si los límites impuestos por el dominio a los testigos que se están utilizando lo permiten y no se está dando al identificador de testigo un uso conflictivo, el dominio más alto deberá expandirse hasta incluirlo. La inhibición de un identificador de testigo en el dominio más bajo no da lugar a conflicto si se inhibe también en el dominio más alto. Todas las demás combinaciones de uso simultáneo están desautorizadas, y el identificador de testigo deberá ser purgado del dominio más bajo.

Si un testigo tiene un extenso conjunto de usuarios inhibidores, sus atributos pudieran no caber en una sola **MTrq** y deberán enviarse hacia arriba en múltiples MCSPDU. Sin embargo, la segunda petición para el mismo testigo inhibido, y las siguientes, deberán aplazarse hasta que se haya recibido una **MTcf** como contestación a la primera. Sólo entonces se sabrá si los límites impuestos por el dominio han permitido que se ponga en uso el testigo en el dominio más alto. Si la primera petición fracasa, no se repetirá con un subconjunto restante de inhibidores.

Cada **MTrq** provoca que el proveedor MCS superior conteste con una **MTcf** en la misma secuencia. Una **MTcf** no contiene nada que identifica explícitamente la **MTrq** precedente. Las contestaciones se encaminarán basándose exclusivamente en el orden en que se recibieron estas MCSPDU. Los proveedores MCS por encima del antiguo proveedor superior tomarán nota de cada **MTrq** no contestada, y de su procedencia, de manera que se pueda retornar la correspondiente **MTcf** a través de la misma conexión MCS.

Los proveedores MCS intermedios validarán los identificadores de usuario anunciados en atributos de testigo, para asegurar que se asignan legítimamente al subárbol en que se origina la **MTrq**. Se suprimirán los identificadores de usuario inválidos. Un testigo que está siendo cedido permanecerá tomado si se suprime quien lo tomó, o quien lo recibió, pero no ambos. Si como consecuencia de esta supresión de identificadores de usuario se libera un testigo, todos sus atributos deberán suprimirse de la petición de fusión y sólo el identificador de testigo se incluirá en el conjunto que va a ser purgado. Un testigo inhibido permanecerá inhibido en **MTrq** incluso si se suprimen todos los inhibidores, dejando un conjunto vacío en los atributos, porque pueden sobrevivir inhibidores de otras MCSPDU. Salvo para esta validación de identificadores de usuario, los proveedores MCS no modificarán el contenido de una **MTrq**.

Un antiguo proveedor superior esperará la confirmación individual de que todos los identificadores de usuario hayan sido incorporados al dominio fusionado, o purgados, antes de empezar a someter atributos de testigo para la fusión.

10.11 MTcf

Una **MTcf** contesta a una **MTrq** precedente. Refleja el mismo conjunto de identificadores de testigo y un subconjunto de los atributos. Los atributos de testigo no incorporados en el dominio fusionado se comunican como identificadores de testigo para ser purgados.

CUADRO 10-11/T.125

MCSPDU MTcf

Contenido	Fuente	Sumidero
Fusión de testigos	Proveedor superior	Intermedios
Identificadores de testigo purgados	Proveedor superior	Antiguo superior

Los identificadores de testigo aceptados se reflejan con los atributos que se introdujeron en la base de información del proveedor MCS superior. Los proveedores intermedios actualizarán su base de información para hacerla conforme.

Los testigos que han de ser purgados del dominio más bajo se listan por los identificadores solamente. Si los mismos identificadores de testigo se utilizaron en el dominio más alto, se mantienen sin modificación. Los proveedores intermedios reenviarán los identificadores de testigo purgados sin tratarlos.

Los proveedores MCS encaminarán **MTcf** a la fuente de la **MTrq** precedente basándose en el conocimiento de que hay una contestación de uno a uno. **MTcf** retorna al antiguo proveedor superior que generó **MTrq**. Allí los testigos fusionados pueden ser ignorados, ya que han seguido en la base de información mientras está pendiente una contestación. Los identificadores de testigo purgados se suprimirán, ya que están destinados a **PTin**.

Los proveedores MCS intermedios confirmarán que los identificadores de usuario anunciados en atributos de testigo se asignan al subárbol al cual se encamina **MTcf**. Si algún identificador de usuario ha sido desanexionado y reasignado en algún otro lugar en el tiempo transcurrido desde que se validó la **MTrq** precedente, un proveedor intermedio generará para ellos una **DUrq** con el código de motivo *canal purgado* haciéndolos víctimas de la fusión de dominio. Si un identificador de testigo no inhibido es liberado mediante esta supresión de identificadores de usuario, será trasladado al conjunto purgado.

10.12 PTin

PTin se genera en un antiguo proveedor superior tras la recepción de **MTcf**. Se difunde en sentido descendente y purga el uso de identificadores de testigo especificados eliminándolos de los proveedores subordinados.

CUADRO 10-12/T.125

MCSPDU PTin

Contenido	Fuente	Sumidero
Purga de identificadores de testigo	Antiguo superior	Subordinados

El efecto de purgar un testigo es grave: indicación MCS-DESANEXIÓN-USUARIO a todo usuario que haya tomado, inhibido, o a que se esté cediendo un identificador de testigo. Un proveedor deberá implementar esto generando **DUrq** a nombre de los usuarios influidos con el motivo *testigo purgado*.

NOTA – Se prevé que la Recomendación T.122 se revisará en el futuro para prever una indicación MCS-LIBERACIÓN-TESTIGO en esta situación. Esto permitiría al usuario influido permanecer afectado aunque se le revoque el derecho de utilizar el testigo.

10.13 DPum

DPum se genera por una petición MCS-DESCONEXIÓN-PROVEEDOR. Esta unidad genera a su vez una indicación MCS-DESCONEXIÓN-PROVEEDOR en el otro extremo de una conexión MCS. **DPum** obliga al receptor a desconectar la conexión MCS que la transportó.

DPum la genera también un proveedor MCS cuando detecta una condición de error como la existencia de un ciclo en la jerarquía del dominio. En tales casos el motivo es uno que no sea *solicitado por el usuario*.

CUADRO 10-13/T.125

MCSPDU DPum

Contenido	Fuente	Sumidero
Motivo (o razón)	Proveedor solicitante	Indicación

10.14 RJum

Se genera **RJum** cuando un proveedor MCS recibe una MCSPDU inválida o detecta un error de protocolo MCS. Esta unidad invita al proveedor par en el otro extremo de una conexión MCS a desconectar, ya que una recuperación tras una situación que no debería ocurrir sería incierta.

RJum diagnostica el error y retorna una porción inicial de la TSDU errónea, por lo general tantos octetos como quepan en una MCSPDU de tamaño máximo. El proveedor receptor puede optar entre desconectar y perseverar.

CUADRO 10-14/T.125

MCSPDU RJum

Contenido	Fuente	Sumidero
Diagnóstico	Proveedor rechazante	Proveedor rechazado
Octetos iniciales	Proveedor rechazante	Proveedor rechazado

10.15 AUrq

AUrq se genera por una respuesta MCS-ANEXIÓN-USUARIO. Esta unidad sube al proveedor MCS superior, que contesta con una **AUcf**. Si el límite impuesto por el dominio al número de los identificadores de usuario lo permite, se genera un nuevo identificador de usuario.

CUADRO 10-15/T.125

MCSPDU AUrq

Contenido	Fuente	Sumidero
(Ninguno)	–	–

La **AUrq** sólo contiene una información: el tipo de MCSPDU. El dominio a que se afecta el usuario viene determinado por la conexión MCS que transporta la MCSPDU. La única característica inicial del identificador de usuario generado es su unicidad.

Un proveedor MCS deberá tomar nota de cada **AUrq** no contestada recibida y de la conexión MCS por la que llegó, a fin de poder contestar con una **AUcf** dirigida a la misma fuente. Para distribuir equitativamente las contestaciones, cada proveedor deberá mantener una cola de tipo primero en entrar primero en salir, con este fin.

10.16 AUcf

Se genera **AUcf** en el proveedor MCS superior al recibir **AUrq**. Cuando se transmite en retorno al proveedor solicitante, esta unidad genera una confirmación MCS-ANEXIÓN-USUARIO.

CUADRO 10-16/T.125

MCSPDU AUcf

Contenido	Fuente	Sumidero
Resultado	Proveedor superior	Confirmación
Iniciador (facultativo)	Proveedor superior	Confirmación

AUcf contiene un identificador de usuario únicamente si el resultado es de éxito. Los proveedores que reciban una **AUcf** de éxito introducirán el identificador de usuario en su base de información.

Los proveedores MCS encaminarán **AUcf** a la fuente de una **AUrq** precedente basándose en el conocimiento de que hay una contestación de uno a uno. Un proveedor que transmita **AUcf** tomará nota de la conexión MCS descendente a la que se asignó de esa forma el nuevo identificador de usuario, de modo que pueda validar el identificador de usuario cuando aparezca después en otras peticiones.

10.17 DUrq

DUrq se genera por una petición MCS-DESANEXIÓN-USUARIO. Si es válida, asciende al proveedor MCS superior, el cual suprime al usuario en su base de información y difunde **DUin** para informar a otros proveedores sobre el cambio.

CUADRO 10-17/T.125

MCSPDU DUrq

Contenido	Fuente	Sumidero
Motivo (o razón)	Proveedor solicitante	Proveedor superior
Identificadores de usuario	Proveedor solicitante	Proveedor superior

Una petición MCS-DESANEXIÓN-USUARIO genera una **DUrq** que contiene el motivo *solicitado por usuario* y un solo identificador de usuario.

DUrq la genera también un proveedor MCS cuando se desconecta una conexión MCS descendente. En ese punto, todos los usuarios en el subárbol influido se pierden y se informará que han sido desanexionados dándose como motivo *dominio desconectado*. Si están pendientes asignaciones de identificadores de usuario, una ulterior contestación, sea **MCcf** o **AUcf**, podrá entonces quedar sin ruta de retorno a su fuente en el subárbol desconectado. Un proveedor en tal situación deberá también generar **DUrq** para suprimir los identificadores de usuario no asignables.

Los proveedores que reciben **DUrq** validarán los identificadores de usuario contenidos en dicha unidad para asegurar que son legítimamente asignados al subárbol de origen. Los identificadores de usuario inválidos se suprimirán. Si no quedan identificadores de usuario, la **DUrq** deberá ignorarse.

Los identificadores de usuario contenidos en **DUrq** no se suprimirán en la base de información hasta que el proveedor reciba **DUin**. De esta forma se mantiene la coherencia con el proveedor MCS superior.

NOTA – Si en un dominio MCS se aplica más de una prioridad de datos, **DUin** podría llegar a un proveedor dado antes de datos enviados con anterioridad al mismo usuario, pero con una prioridad más baja. Este protocolo no impide que se entreguen datos a una anexión incluso después de que se haya comunicado mediante una **DUin** que el emisor se había desanexionado.

10.18 DUin

Se genera **DUin** en el proveedor MCS superior tras la recepción de **DUrq**. Dicha unidad se difunde hacia abajo a todos los demás proveedores y genera indicaciones MCS DESANEXIÓN-USUARIO en todas las anexiones.

En una anexión subsistente, **DUin** genera una MCS DESANEXIÓN-USUARIO para cada identificador de usuario que ella contiene. No importa que el usuario notificado haya tenido o no conocimiento previo de la existencia de un usuario desanexionado.

Al recibir una **DUin** que contiene su propio identificador de usuario, una anexión MCS deja de existir. Cualesquiera canales que dejen de estar incorporados como resultado de la desanexión de un usuario, con excepción del propio canal de identificador de usuario, será abandonado mediante **CLrq**.

CUADRO 10-18/T.125

MCSPDU DUin

Contenido	Fuente	Sumidero
Motivo (o razón)	Proveedor superior	Indicación
Identificadores de usuario	Proveedor superior	Indicación

Los proveedores que reciben **DUin** suprimirán los identificadores de usuario especificados en su base de información. Los canales gestionados por un usuario desanesionado se suprimirán si no hay otros usuarios admitidos. Si quedan otros usuarios, la supresión del gestor tendrá por consecuencia que el proveedor superior les envíe una **CDin** en multidifusión. El estado de todo testigo que haya sido tomado o inhibido por un usuario desanexionado, o que se le esté cediendo, deberá ajustarse correspondientemente. La supresión de un receptor previsto de testigo provocará que el proveedor superior envíe una **TVcf** de fracaso al cedente del testigo, a menos que haya liberado el testigo o se haya desanexionado a sí mismo.

10.19 CJrq

CJrq se genera por una petición MCS-INCORPORACIÓN-CANAL. Si es válida, sube hasta llegar a un proveedor MCS con información suficiente para generar una contestación **CJcf**. Este puede ser el proveedor MCS superior.

CUADRO 10-19/T.125

MCSPDU CJrq

Contenido	Fuente	Sumidero
Iniciador	Proveedor solicitante	Proveedor más alto
Identificador de canal	Petición	Proveedor más alto

El identificador de usuario de la anexión MCS iniciadora la suministra el proveedor MCS que recibe la petición primitiva. Los proveedores que reciben **CJrq** ulteriormente validarán el identificador de usuario para asegurar que sea legítimamente asignado al subárbol de origen. Si el identificador de usuario es inválido se ignorará la MCSPDU.

NOTA – Con esto se tiene en cuenta la posibilidad de que **CJrq** compita en una especie de carrera hacia arriba con una purga de identificador de usuario iniciador que estuviera bajando. Un proveedor que recibe **PCin** primero podría recibir poco después una **CJrq** que contiene un identificador de usuario inválido. Este es un suceso normal y no es motivo para rechazar la MCSPDU.

CJrq puede subir hasta un proveedor MCS que tiene en su base de información el identificador de canal solicitado. Cualquier proveedor en tal situación, al actuar conformemente con el proveedor MCS superior, deberá convenir en que la petición debe o no tener éxito. Si la petición debe fracasar, el proveedor generará una **CJcf** de fracaso. Si debe tener éxito y el proveedor ya se ha incorporado al mismo canal, generará una **CJcf** de éxito. En estos dos casos, MCS-INCORPORACIÓN-CANAL se completa sin que sea necesario llegar al proveedor MCS superior. En otro caso, si la petición debe tener éxito pero todavía no se ha efectuado la incorporación al canal, un proveedor deberá reenviar **CJrq** hacia arriba.

Si **CJrq** sube hasta el proveedor MCS superior, el identificador del canal solicitado podría ser cero, valor que no se encuentra en ninguna base de información porque corresponde a un identificador inválido. Si lo permite el límite impuesto por el dominio al número de canales en uso, deberá generarse un nuevo identificador de canal asignado, y retornarse en una **CJcf** de éxito. Si el identificador del canal solicitado está en la gama estática y el límite impuesto por el dominio al número de canales en uso lo permite, el identificador de canal se introducirá en la base de información y se retornará asimismo en una **CJcf** de éxito.

En los demás casos, la petición sólo tendrá éxito si el identificador del canal se encuentra ya en la base de información del proveedor MCS superior. A un identificador de canal de usuario sólo puede incorporarse el mismo usuario. A un identificador de canal privado sólo pueden incorporarse los usuarios que hayan sido previamente admitidos por su gestor. A un identificador de canal asignado puede incorporarse cualquier usuario.

10.20 CJcf

CJcf se genera en un proveedor más alto al recibirse **CJrq**. Cuando se devuelve al proveedor solicitante genera una confirmación MCS-INCORPORACIÓN-CANAL.

CUADRO 10-20/T.125

MCSPDU CJcf

Contenido	Fuente	Sumidero
Resultado	Proveedor más alto	Confirmación
Iniciador	Proveedor más alto	Encaminamiento MCSPDU
Solicitado	Proveedor más alto	Confirmación
Identificador de canal	Proveedor más alto	Confirmación

CJcf contiene un identificador de canal incorporado solamente si el resultado es éxito.

La petición de identificador de canal es igual que en **CJrq**. Esto ayuda a la anexión iniciadora a relacionar la confirmación MCS-INCORPORACIÓN-CANAL con una petición precedente. Como **CJrq** no tiene que subir al proveedor superior, las confirmaciones se pueden producir fuera de orden.

Si el resultado es éxito, **CJcf** incorpora el proveedor MCS receptor al canal especificado. De allí en adelante, los proveedores más altos le encaminarán todo dato que los usuarios envíen por el canal. Un proveedor permanecerá incorporado al canal mientras lo esté cualquiera de sus anexiones o proveedores subordinados. Para abandonar el canal un proveedor deberá generar **CLrq**.

Los proveedores que reciben una **CJcf** de éxito introducirán el identificador de canal en su base de información. Si no está ya allí, se dará al identificador de canal el tipo estático o asignado, de acuerdo con su rango.

CJcf se reenviará en el sentido de transmisión del identificador del usuario iniciador. Si el identificador de usuario no puede obtenerse porque ya no existe una conexión MCS, el proveedor decidirá si hay o no motivo para permanecer incorporado al canal. Si no lo hay, generará **CLrq**.

10.21 CLrq

CLrq la genera un proveedor para retirarse de un conjunto de canales. Esto puede ser motivado por una petición MCS-ABANDONO-CANAL procedente de la última anexión incorporada al canal. **CLrq** continúa subiendo si los proveedores más altos, en consecuencia, dejan también de tener motivo para estar incorporados.

Los proveedores que reciben **CLrq** detendrán los encaminamientos a la conexión MCS que le transportó cualesquiera datos enviados por usuarios a través de los canales especificados. Cuando la última anexión o proveedor subordinado abandona un canal, un proveedor MCS generará una **CLrq** correspondiente.

CUADRO 10-21/T.125

MCSPDU CLrq

Contenido	Fuente	Sumidero
Identificadores de canal	Proveedor solicitante	Proveedor más alto

10.22 CCrq

CCrq se genera por una petición MCS-FORMACIÓN CANAL. Si es válida, sube al proveedor MCS superior, el cual contesta con una **CCcf**. Si el límite impuesto por el dominio al número de identificadores de canal lo permite, se genera un nuevo identificador de canal privado.

CCrq contiene el identificador del usuario iniciador, que se validará como se ha explicado para **CJrq**.

El solicitante deviene gestor del canal privado. Inicialmente nadie está incorporado al canal y su gestor es el único usuario admitido.

CUADRO 10-22/T.125

MCSPDU CCrq

Contenido	Fuente	Sumidero
Iniciador	Proveedor solicitante	Proveedor superior

10.23 CCcf

CCcf se genera en el proveedor MCS superior al recibirse **CCrq**. Cuando se devuelve al proveedor solicitante, genera una confirmación MCS-FORMACIÓN-CANAL.

CUADRO 10-23/T.125

MCSPDU CCcf

Contenido	Fuente	Sumidero
Resultado	Proveedor superior	Confirmación
Iniciador	Proveedor superior	Encaminamiento MCSPDU
Identificador de canal (facultativo)	Proveedor superior	Confirmación

CCcf contiene un identificador de canal privado sólo si el resultado es satisfactorio.

Los proveedores que reciben una **CCcf** de éxito introducirán el identificador de canal en su base de información como un canal privado con el usuario iniciador como su gestor.

CCcf se reenviará en el sentido de ida del identificador del usuario iniciador. Si el identificador de usuario no puede alcanzarse porque ya no existe una conexión MCS, no es necesario ejecutar acciones especiales, ya que, de todas formas, tiene que llegar posteriormente una **DUin** para comunicar que el iniciador se ha desanexionado. Dado que el iniciador es su gestor, queda con esto suprimido el identificador de canal en la base de información.

10.24 CDrq

CDrq se genera por una petición MCS-DISOLUCIÓN-CANAL. Si es válida sube al proveedor MCS superior, el cual suprime el identificador de canal y genera **CDin**.

CUADRO 10-24/T.125

MCSPDU CDrq

Contenido	Fuente	Sumidero
Iniciador	Proveedor solicitante	Proveedor superior
Identificador de canal	Petición	Proveedor superior

CDrq puede ser generada también por un proveedor MCS por iniciativa propia para disolver un canal.

CDrq contiene el identificador del usuario iniciador, que debe ser validado para asegurar que será legítimamente asignado al subárbol de origen. Si el iniciador no coincide con el gestor del canal privado, tal como está registrado en la base de información, se ignorará la MCSPDU.

10.25 CDin

CDin se genera en el proveedor MCS superior al recibirse **CDrq**. Esta unidad se envía hacia abajo, por multidifusión, a los proveedores que contienen el gestor o un usuario admitido en su subárbol. Genera indicaciones MCS-EXCLUSIÓN-CANAL a los usuarios admitidos con el motivo *canal disuelto* y, si es iniciada por el proveedor, genera una indicación MCS-DISOLUCIÓN-CANAL al gestor.

CUADRO 10-25/T.125

MCSPDU CDin

Contenido	Fuente	Sumidero
Identificador de canal	Proveedor superior	Indicación

CDin deberá ser generada también por el proveedor MCS superior cuando se desanexiona el gestor de un canal privado.

Los proveedores que reciben **CDin** suprimirán el canal en su base de información.

10.26 CArq

CArq se genera por una petición MCS-ADMISIÓN-CANAL. Si es válida, sube al proveedor MCS superior, el cual admite en el canal privado los usuarios especificados y envía **CAin** en multidifusión para avisar a los proveedores en cuyo subárbol residen.

CArq contiene el identificador del usuario iniciador, que se validará como se ha explicado para **CDrq**.

Los otros identificadores de usuario de **CDrq**, que representan usuarios que van a admitirse, se validarán en el proveedor MCS superior, que es el único que conoce el conjunto completo de los usuarios. Los que no sean válidos se excluirán de la **CAin** resultante.

CUADRO 10-26/T.125

MCSPDU CArq

Contenido	Fuente	Sumidero
Iniciador	Proveedor solicitante	Proveedor superior
Identificador de canal	Petición	Proveedor superior
Identificadores de usuario	Petición	Proveedor superior

Los identificadores de usuario contenidos en **CArq** no serán admitidos al canal privado hasta que un proveedor reciba **CAin**. De este modo se mantiene la coherencia con el proveedor MCS superior.

10.27 CAin

CAin se genera en el proveedor MCS superior tras la recepción de **CArq**. Esta unidad se envía hacia abajo por multidifusión a los proveedores que contienen en su subárbol un usuario recientemente admitido. Genera indicaciones MCS-ADMISIÓN-CANAL en las anexiones influidas.

Los proveedores que reciben **CAin** actualizarán normalmente el canal en su base de información, admitiendo a los usuarios especificados que residen en su subárbol. Sin embargo, si un proveedor es el superior anterior de un dominio más bajo que se está fusionando como resultado de MCS-CONEXIÓN-PROVEEDOR, puede rechazar la admisión generando **DUrq** para los identificadores de usuario anexionados con el motivo *canal purgado*.

CUADRO 10-27/T.125

MCSPDU CAin

Contenido	Fuente	Sumidero
Iniciador	Proveedor superior	Indicación
Identificador de canal	Proveedor superior	Indicación
Identificadores de usuario	Proveedor superior	Encaminamiento MCSPDU

10.28 CErq

CErq se genera por una petición MCS-EXCLUSIÓN-CANAL. Si es válida sube al proveedor MCS superior, quien excluye del canal los usuarios especificados y multidifunde **CEin** para avisar a los proveedores en cuyo subárbol residen.

CErq contiene el identificador del usuario iniciador, que se validará como se ha explicado para **CDrq**.

Los otros identificadores de usuario de **CErq**, que representan usuarios que habrán de ser excluidos, serán validados en el proveedor MCS superior, que es el único que conoce el conjunto completo de los usuarios admitidos. Los que hayan sido admitidos no serán incluidos en la **CEin** resultante.

Los identificadores de usuario contenidos en **CErq** no serán excluidos del canal privado hasta que un proveedor reciba **CEin**. De esta forma se mantiene la coherencia con el proveedor MCS superior.

CUADRO 10-28/T.125

MCSPDU CErq

Contenido	Fuente	Sumidero
Iniciador	Proveedor solicitante	Proveedor superior
Identificador de canal	Petición	Proveedor superior
Identificadores de usuario	Petición	Proveedor superior

10.29 CEin

CEin se genera en el proveedor MCS superior al recibirse **CErq**. Se envía hacia abajo por multidifusión a los proveedores que contienen en su subárbol un usuario excluido. Genera indicaciones MCS-EXCLUSIÓN-CANAL en las anexiones influidas, con el motivo solicitado *por usuario*.

CUADRO 10-29/T.125

MCSPDU CEin

Contenido	Fuente	Sumidero
Identificador de canal	Proveedor superior	Indicación
Identificadores de usuario	Proveedor superior	Encaminamiento MCSPDU

Los proveedores que reciben **CEin** actualizarán el canal en su base de información, suprimiendo los usuarios especificados en el conjunto de usuarios admitidos en el canal. Si el conjunto de usuarios admitidos en un canal privado se vacía y el gestor no reside en el subárbol, se suprimirá el identificador de canal en la base de información. En otro caso, si como resultado de las exclusiones no queda nadie incorporado al canal, un proveedor generará una **CLrq** correspondiente.

Un proveedor que reenvía **CEin** determinará, para cada subárbol de destino, si, después de ello, contiene cualesquiera anexiones admitidas en el canal privado. Si no hay ninguna, el proveedor deberá llegar a la conclusión de que el proveedor subordinado correspondiente ya no está incorporado al canal privado y actualizará su base de información inmediatamente a este efecto, sin esperar **CLrq**.

10.30 SDrq

SDrq se genera por una petición MCS-ENVÍO-DATOS. Si es válida, sube hasta el proveedor MCS superior. A lo largo del camino, los proveedores pueden generar a partir de ella una **SDin** con idéntico contenido, y multidifundirla hacia abajo.

SDrq contiene el identificador del usuario iniciador, que será validado como se ha explicado para **CJrq**.

Si el identificador de canal figura en la base de información del proveedor MCS superior como un canal privado y el iniciador de **SDrq** no es un usuario admitido, se ignorará la MCSPDU.

La TC inicial o adicional que transporta **SDrq** cotejará su prioridad de datos, teniendo en cuenta el número de prioridades implementadas en el dominio. Las MCSPDU que llegan a través de una conexión MCS por la TC errónea deberán rechazarse.

CUADRO 10-30/T.125

MCSPDU SDrq

Contenido	Fuente	Sumidero
Iniciador	Proveedor solicitante	Proveedor más alto
Identificador de canal	Petición	Proveedor más alto
Prioridad de datos	Petición	Proveedor más alto
Segmentación	Proveedor solicitante	Proveedor más alto
Datos de usuario	Petición	Proveedor más alto

Un proveedor fijará las banderas de segmentación *comienzo* y *fin* para mostrar la relación de los datos de usuario en la **SDrq** con los confines de una unidad de datos de servicio MCS. Los proveedores están en libertad de fragmentar y reensamblar las MCSPDU que forman parte de la misma unidad de datos de servicio MCS, mientras esto no perturbe la integridad de datos de usuario. Sin embargo, será poco el beneficio que se obtenga de tal manipulación, pues el tamaño máximo de una MCSPDU es constante en la totalidad de un dominio.

Un proveedor generará, a partir de **SDrq**, una **SDin** con el mismo contenido y la transmitirá a todos los proveedores que se hayan incorporado al canal especificado, excepto al proveedor subordinado que transmitió **SDrq** hacia arriba. A menos que el canal esté listado en la base de información del proveedor como un identificador de usuario residente en su subárbol, reenviará también **SDrq** hacia arriba.

10.31 SDin

Se genera **SDin** en un proveedor MCS más alto al recibirse **SDrq**. Esta unidad se multidifunde hacia abajo y genera indicaciones MCS-ENVÍO-DATOS en todas las anexiones que están incorporadas al canal.

CUADRO 10-31/T.125

MCSPDU SDin

Contenido	Fuente	Sumidero
Iniciador	Proveedor más alto	Indicación
Identificador de canal	Proveedor más alto	Indicación
Prioridad de datos	Proveedor más alto	Indicación
Segmentación	Proveedor más alto	Proveedor indicante
Datos de usuario	Proveedor más alto	Indicación

La TC inicial o adicional que transporta **SDin** cotejará su prioridad de datos, teniendo en cuenta el número de prioridades implementadas en el dominio. Las MCSPDU que llegan a través de una conexión MCS por la TC errónea deberán rechazarse.

Las banderas de segmentación *comienzo* y *fin* permiten reensamblar datos de usuario para formar una unidad de datos de servicio MCS completa. Estas banderas se interpretarán en el contexto de las MCSPDU **SDin** que llegan del mismo usuario a través del mismo canal y con la misma prioridad. Un flujo de fragmentos a reensamblar pueden estar entrelazados con otras MCSPDU y datos de otros usuarios transmitidos por otros canales y con otras prioridades.

La manera de indicar las unidades de datos de servicio a los usuarios MCS anexionados es una cuestión de implementación local. Una posible solución consiste en entregar cada MCSPDU como una unidad de datos de interfaz separada, con banderas de segmentación incluidas. En otras soluciones, se puede tratar de reensamblar dentro del proveedor receptor, deben prever el caso de las unidades de datos de servicio de gran tamaño y dar a conocer al usuario el orden relativo en que comienzan a llegar las unidades de datos de servicio.

Los proveedores que reciben **SDin** la reenviarán a todos los subordinados que están incorporados al canal.

10.32 USrq

USrq se genera por una petición MCS-ENVÍO-DATOS-UNIFORME. Si es válida, sube al proveedor MCS superior, el cual, basándose en ella, genera una **USin** de contenido idéntico y la multidifunde hacia abajo.

CUADRO 10-32/T.125

MCSPDU USrq

Contenido	Fuente	Sumidero
Iniciador	Proveedor solicitante	Proveedor superior
Identificador de canal	Petición	Proveedor superior
Prioridad de datos	Petición	Proveedor superior
Segmentación	Proveedor solicitante	Proveedor superior
Datos de usuario	Petición	Proveedor superior

USrq contiene el identificador del usuario iniciador, que se validará como se ha explicado para **CJrq**.

Si el identificador de canal figura en la base de información del proveedor MCS receptor como un canal privado y el iniciador de **USrq** no es un usuario admitido, se ignorará la MCSPDU.

La TC inicial o adicional que transporta **USrq** cotejará su prioridad de datos, teniendo en cuenta el número de prioridades implementadas en el dominio. Las MCSPDU que llegan a través de una conexión MCS por la TC errónea deberán rechazarse.

Un proveedor fijará las banderas de segmentación *comienzo* y *fin* para mostrar la relación de datos de usuario en la **USrq** con los confines de una unidad de datos de servicio MCS. Los proveedores están en libertad de segmentar y reensamblar las MCSPDU que forman parte de la misma unidad de datos de servicio MCS mientras esto no perturbe la integridad de datos de usuario. Sin embargo, será poco el beneficio que se obtenga de tal manipulación, pues el tamaño máximo de una MCSPDU es constante en la totalidad de un dominio.

Basándose en **USrq**, el proveedor MCS superior generará una **USin** con el mismo contenido.

10.33 USin

USin se genera en el proveedor MCS superior al recibirse **USrq**. Esta unidad se multidifunde hacia abajo y genera indicaciones MCS-ENVÍO-DATOS-UNIFORME en todas las anexiones que están incorporadas al canal.

CUADRO 10-33/T.125

MCSPDU USin

Contenido	Fuente	Sumidero
Iniciador	Proveedor superior	Indicación
Identificador de canal	Proveedor superior	Indicación
Prioridad de datos	Proveedor superior	Indicación
Segmentación	Proveedor superior	Proveedor indicante
Datos de usuario	Proveedor superior	Indicación

La TC inicial o adicional que transporta **USin** cotejará su prioridad de datos, teniendo en cuenta el número de prioridades implementadas en el dominio. Las MCSPDU que llegan a través de una conexión MCS por la TC errónea deberán rechazarse.

Las banderas de segmentación *comienzo* y *fin* permiten reensamblar datos de usuario para formar una unidad de datos de servicio MCS completa. Estas banderas se interpretarán en el contexto de las MCSPDU **USin** que llegan del mismo usuario a través del mismo canal y con la misma prioridad. Un flujo de fragmentos a reensamblar pueden entrelazarse con otras MCSPDU y datos de otros usuarios transmitidos por otros canales y con otras prioridades.

La manera de indicar las unidades de datos a los usuarios MCS anexionados es una cuestión de implementación local.

Los proveedores que reciben **USin** la reenviarán a todos los subordinados que estén incorporados al canal.

10.34 TGrq

TGrq se genera por una petición MCS-TOMA-TESTIGO. Si es válida, sube al proveedor MCS, el que contesta con una **TGcf**.

CUADRO 10-34/T.125

MCSPDU TGrq

Contenido	Fuente	Sumidero
Iniciador	Proveedor solicitante	Proveedor superior
Identificador de testigo	Petición	Proveedor superior

TGrq contiene el identificador del usuario iniciador, que se validará como se ha explicado para **CJrq**.

Si el testigo está libre y el límite impuesto por el dominio al número de testigos lo autoriza, será tomado. Si el testigo está inhibido por el usuario solicitante solamente, será tomado. En otro caso, el estado del testigo no cambiará.

10.35 TGcf

TGcf se genera en el proveedor MCS superior al recibirse **TGrq**. Cuando ha sido devuelta al proveedor MCS solicitante genera una confirmación MCS-TOMA-TESTIGO.

CUADRO 10-35/T.125

MCSPDU TGcf

Contenido	Fuente	Sumidero
Resultado	Proveedor superior	Confirmación
Iniciador	Proveedor superior	Encaminamiento MCSPDU
Identificador de testigo	Proveedor superior	Confirmación
Estado de testigo	Proveedor superior	Confirmación

El resultado será *éxito* si el testigo ya estaba libre o si el mismo usuario cambió su estado de inhibido a tomado. Otros resultados son *demasiados testigos* y *testigo no disponible*. Este último se aplica a un testigo ya tomado por el solicitante; esto se determina examinando el estado del testigo.

Los proveedores que reciben **TGcf** actualizarán el estado del testigo en la base información para que concuerde con el estado retornado.

Se reenviará **TGcf** en el mismo sentido que el identificador del usuario iniciador. Si el identificador del usuario no puede obtenerse porque ya no existe una conexión MCS, no es necesario ejecutar acciones especiales, ya que tiene que llegar posteriormente una **DUin** para informar que el iniciador se ha desanexionado. Este liberará su control sobre el identificador de testigo en su base de información.

10.36 TIrq

TIrq se genera por una petición MCS-INHIBICIÓN-TESTIGO. Si es válida, sube al proveedor MCS superior, el cual contesta con una **TIcf**.

CUADRO 10-36/T.125

MCSPDU TIrq

Contenido	Fuente	Sumidero
Iniciador	Proveedor solicitante	Proveedor superior
Identificador de testigo	Petición	Proveedor superior

TIrq contiene el identificador del usuario iniciador, que se validará como se ha explicado para **CJrq**.

Si el testigo está libre y el límite impuesto por el dominio al número de testigos lo autoriza, será inhibido. Si el testigo está tomado por el usuario solicitante, será inhibido. Si el testigo está ya inhibido, se añadirá el solicitante al conjunto de inhibidores. En otro caso, el estado del testigo no cambiará.

10.37 TIcf

TIcf se genera en el proveedor MCS superior al recibirse **TIrq**. Cuando ha sido devuelta al proveedor solicitante, genera una confirmación MCS-INHIBICIÓN-TESTIGO.

El resultado será *éxito* si el testigo ya estaba libre o inhibido, o si el mismo usuario cambió su estado de tomado a inhibido. Otros resultados son *demasiados testigos* y *testigo no disponible*.

CUADRO 10-37/T.125

MCSPDU TIcf

Contenido	Fuente	Sumidero
Resultado	Proveedor superior	Confirmación
Iniciador	Proveedor superior	Encaminamiento MCSPDU
Identificador de testigo	Proveedor superior	Confirmación
Estado del testigo	Proveedor superior	Confirmación

Los proveedores que reciben **TIcf** actualizarán el estado del testigo en la base información para que concuerde con el estado retornado.

Esta MCSPDU se encamina de la misma forma que **TGcf**.

10.38 TVrq

TVrq se genera por una petición MCS-CESIÓN-TESTIGO. Si es válida sube al proveedor MCS superior, el cual genera **TVin** o una **TVcf** de fracaso.

CUADRO 10-38/T.125

MCSPDU TVrq

Contenido	Fuente	Sumidero
Iniciador	Proveedor solicitante	Proveedor superior
Identificador de testigo	Petición	Proveedor superior
Receptor	Petición	Proveedor superior

TVrq contiene el identificador del usuario iniciador, que se validará como se ha explicado para **CJrq**.

Si el solicitante ha tomado el testigo y el receptor deseado existe, se transmite **TVin** hacia el receptor. De no ser así, la petición fracasará, el estado del testigo se mantendrá sin cambiar, y se transmitirá **TVcf** hacia el solicitante con el resultado *testigo no poseído* o *no hay tal usuario*.

10.39 TVin

Se genera **TVin** en el proveedor MCS superior al recibirse **TVrq**. Cuando se ha encaminado al receptor deseado, genera una indicación MCS-CESIÓN-TESTIGO.

Los proveedores que reciben **TVin** por lo general actualizan el identificador del testigo en su base de información como en curso de cesión del iniciador al receptor. Si embargo, si un proveedor es el antiguo superior de un dominio más bajo que se está fusionando como resultado de MCS-CONEXIÓN-PROVEEDOR, puede rechazar el testigo ofrecido generando **TVrs** con el motivo *fusión de dominio*.

CUADRO 10-39/T.125

MCSPDU TVin

Contenido	Fuente	Sumidero
Iniciador	Proveedor superior	Indicación
Identificador de testigo	Proveedor superior	Indicación
Receptor	Proveedor superior	Encaminamiento MCSPDU

Se reenviará **TVin** en el sentido del identificador del usuario receptor. Si el identificador de usuario no puede obtenerse porque ya no existe una conexión MCS, no es necesario ejecutar acciones especiales, ya que ulteriormente tiene que llegar una **DUin** para informar que el receptor se ha desanexionado. Este liberará su control sobre el identificador de testigo en la base de información.

10.40 TVrs

Se genera **TVrs** por una respuesta MCS-CESIÓN-TESTIGO. Si es válida sube al proveedor MCS superior, el cual genera una **TVcf** para comunicar el resultado al cedente del testigo.

CUADRO 10-40/T.125

MCSPDU TVrs

Contenido	Fuente	Sumidero
Resultado	Respuesta	Proveedor superior
Receptor	Proveedor respondedor	Proveedor superior
Identificador de testigo	Proveedor respondedor	Proveedor superior

Un resultado de *éxito* significa que el receptor ha aceptado el testigo ofrecido.

El identificador de usuario de la anexión MCS respondedora lo suministra el proveedor MCS que recibe la respuesta de primitiva. Los proveedores que reciben **TVrs** subsiguientemente validarán el identificador de usuario para asegurar que se asigna legítimamente al subárbol de origen. Si el identificador de usuario no es válido se ignorará la MCSPDU.

Si el identificador de testigo no figura en la base de información del proveedor como en curso de cesión al receptor, se ignorará la MCSPDU. Si el identificador de testigo está todavía tomado por el cedente, su estado se actualizará como tomado por el receptor si el resultado es de éxito; en otro caso, el estado volverá a ser el de tomado por el cedente, o se suprimirá en la base de información, lo que dependerá de que el cedente resida o no en el subárbol del proveedor. Si, después de eso, el identificador de testigo ha sido liberado por el cedente y el resultado no es de éxito, se suprimirá el testigo en la base de información del proveedor.

Si la MCSPDU es válida y no ha sido ignorada, se reenviará hacia arriba. El proveedor MCS superior tratará la **TVrs** como se ha especificado antes. Además, si el cedente no ha liberado ya el testigo, el proveedor superior generará una **TVcf** que contendrá el mismo resultado que **TVrs**.

10.41 TVcf

Se genera **TVcf** en el proveedor MCS superior al recibirse **TVrs**. Cuando se devuelve al proveedor solicitante, genera una confirmación MCS-CESIÓN-TESTIGO.

CUADRO 10-41/T.125

MCSPDU TVcf

Contenido	Fuente	Sumidero
Resultado	Proveedor superior	Confirmación
Iniciador	Proveedor superior	Encaminamiento MCSPDU
Identificador de testigo	Proveedor superior	Confirmación
Estado del testigo	Proveedor superior	Confirmación

TVcf la genera también el proveedor MCS superior al recibir **TVrq** si no se puede ofrecer un testigo al receptor deseado. Esto se hace en lugar de generar **TVin**. **TVcf** se generará asimismo con el resultado *no hay tal usuario* si se desanexiona el receptor antes de recibirse **TVrs**.

Los proveedores que reciben **TVcf** actualizarán el estado del testigo en su base de información para que concuerde con el estado retornado.

Esta MCSPDU se encamina de la misma manera que **TGcf**.

10.42 TPrq

TPrq se genera por una petición MCS-SOLICITUD-TESTIGO. Si es válida, sube al proveedor MCS superior, el cual multidifunde **TPin** para avisar a los usuarios actuales con relación al testigo.

TPrq contiene el identificador del usuario iniciador, que se validará como se ha explicado para **CJrq**.

CUADRO 10-42/T.125

MCSPDU TPrq

Contenido	Fuente	Sumidero
Iniciador	Proveedor solicitante	Proveedor superior
Identificador de testigo	Petición	Proveedor superior

10.43 TPin

TPin se genera en el proveedor MCS superior al recibir **TPrq**. Esta unidad se multidifunde hacia abajo y genera indicaciones MCS-SOLICITUD-TESTIGO.

Los proveedores que reciben **TPin** la reenviarán a todos los subordinados que contienen en su subárbol un usuario que ha tomado o inhibido el testigo especificado, o al que se le está cediendo éste.

CUADRO 10-43/T.125

MCSPDU TPin

Contenido	Fuente	Sumidero
Iniciador	Proveedor superior	Indicación
Identificador de testigo	Proveedor superior	Indicación

10.44 TRrq

TRrq se genera por una petición MCS-LIBERACIÓN-TESTIGO. Si es válida, sube al proveedor MCS superior, que la contesta retornando **TRcf.**

CUADRO 10-44/T.125

MCSPDU TRrq

Contenido	Fuente	Sumidero
Iniciador	Proveedor solicitante	Proveedor superior
Identificador de testigo	Petición	Proveedor superior

TRrq contiene el identificador del usuario iniciador, que se validará como se ha explicado para **CJrq**.

Si el testigo es tomado por el solicitante, deberá quedar libre. Si es inhibido, el solicitante deberá suprimirse en el conjunto de inhibidores; si este conjunto se vacía, el testigo deberá quedar libre. Si el testigo está en curso de dejar de ser poseído por el solicitante, pasará al estado de cedido al receptor deseado, que es un estado intermedio distinto, mientras está pendiente la recepción de **TVrs**. En otro caso, el estado del testigo no cambia.

10.45 TRcf

Se genera **TRcf** en el proveedor MCS superior al recibirse **TRrq**. Cuando se devuelve al proveedor solicitante, genera una confirmación MCS-LIBERACIÓN-TESTIGO.

CUADRO 10-45/T.125

MCSPDU TRcf

Contenido	Fuente	Sumidero
Resultado	Proveedor superior	Confirmación
Iniciador	Proveedor superior	Encaminamiento MCSPDU
Identificador de testigo	Proveedor superior	Confirmación
Estado del testigo	Proveedor superior	Confirmación

El resultado será de *éxito* si el solicitante tomó o inhibió el testigo, o estaba en trámite de cederlo. El otro resultado posible es *testigo no poseído*.

Los proveedores que reciben **TRcf** actualizarán el testigo en su base de información para que concuerde con el estado retornado.

Esta MCSPDU se encamina de la misma forma que **TGcf**.

10.46 TTrq

TTrq se genera por una petición MCS-PRUEBA-TESTIGO. Si es válida, sube al proveedor MCS superior, que la contesta retornando **TTcf**.

TTrq contiene el identificador del usuario iniciador, que se validará como se ha explicado para **CJrq**.

CUADRO 10-46/T.125

MCSPDU TTrq

Contenido	Fuente	Sumidero
Iniciador	Proveedor solicitante	Proveedor superior
Identificador de testigo	Petición	Proveedor superior

10.47 TTcf

Se genera **TTcf** en el proveedor MCS superior al recibirse **TTrq**. Cuando se devuelve al proveedor solicitante, genera una confirmación MCS-PRUEBA-TESTIGO.

Los proveedores que reciben **TTcf** actualizarán el testigo en su base de información para que concuerde con el estado retornado.

Esta MCSPDU se encamina de la misma forma que **TGcf**.

CUADRO 10-47/T.125

MCSPDU TTcf

Contenido	Fuente	Sumidero
Iniciador	Proveedor superior	Encaminamiento MCSPDU
Identificador de testigo	Proveedor superior	Confirmación
Estado del testigo	Proveedor superior	Confirmación

11 Base de información de proveedor MCS

11.1 Replicación de la jerarquía

Un proveedor MCS, aunque puede atender múltiples dominios, sirve a cada uno independientemente. Para cada uno de ellos mantiene bases de información con lógicas separadas en las que registra el estado de los recursos (canales y testigos) que se están utilizando. La descripción que sigue presupone el contexto de un solo dominio.

Los recursos MCS que es necesario manejar en un dominio son los identificadores de canal y los identificadores de testigo. Los identificadores de usuario son un subconjunto de los identificadores de canal. Parámetros del dominio limitan el número de identificadores de cada categoría que pueden usarse simultáneamente. Esto permite a un proveedor calcular la cantidad de memoria necesaria para la base de información en el caso más desfavorable de un dominio utilizado en su totalidad.

En la jerarquía de un dominio, los identificadores en uso en un proveedor MCS cualquiera dado se estabilizan tendiendo a formar un subconjunto de los que se están utilizando en su superior inmediato. La información sobre un identificador se registra en un lugar donde puede utilizarse para soportar servicios MCS en que interviene ese identificador. Un registro más extenso de la información entrañaría costos adicionales en el tráfico de las MCSPDU, para mantener actualizada la información. Puesto que la información registrada en un proveedor está de acuerdo con la registrada en proveedores más altos, dentro del límite del tiempo de propagación de las MCSPDU, puede decirse que la base de información del proveedor está parcialmente replicada a través de la jerarquía del dominio.

Al establecerse la primera conexión MCS de un dominio, los parámetros del dominio quedan fijados y no podrán modificarse. Un proveedor que no tiene la capacidad necesaria para trabajar con el número máximo de identificadores especificados en cada categoría puede negociar para incorporarse a un dominio sobre una base falsa. Puede especular con el hecho de que, dada su baja posición en una jerarquía, no se le exigirá retener más que una fracción de la base de información. Tal proveedor podría no soportar anexiones y subordinados con la gama completa de servicios MCS que éstos esperan. No obstante, mientras su capacidad no haya sido efectivamente rebasada, dicho proveedor podrá actuar como un miembro del dominio igual a los demás. Esta estrategia pudiera convenir nodos terminales con pretensiones limitadas.

Los identificadores se ponen en uso por primera vez en el proveedor MCS superior. En los proveedores subordinados se ponen en uso mediante un flujo descendente selectivo de MCSPDU. La mayor parte de ellos se retiran del uso de la misma manera de arriba a abajo. Existen necesariamente intervalos en los que un proveedor subordinado registra como en uso un identificador que no está registrado como tal en sus superiores, porque la MCSPDU que suprime el identificador está todavía en tránsito. Esta situación, sin embargo, no perdura. Las MCSPDU de control se reciben y procesan en el mismo orden en que se transmiten. Las consecuencias del procesamiento de una MCSPDU, incluida la creación o supresión de identificadores de canal y de testigo, comienzan a surtir efecto antes de que la atención se concentre en el siguiente suceso de entrada.

Una excepción a lo expuesto en el párrafo anterior es la supresión de identificadores de canales estáticos y asignados. Estos identificadores de canal, aunque hayan sido puestos en uso por un flujo descendente de **CJcf**, se suprimen en orden inverso, de abajo a arriba. Específicamente, se suprimen cuando una acumulación de peticiones MCS-ABANDONO-CANAL procedentes de anexiones y MCSPDU **CLrq** procedentes de proveedores subordinados, se combinan para abandonar un canal al que no ha habido incorporación. Esas transiciones dan motivo a la transmisión de **CLrq** más hacia arriba. Así, en estos dos casos, los identificadores de canal registrados como en uso son un subconjunto estricto de los que se encuentran en uso en el proveedor más alto. Este es un corolario accidental de las optimizaciones que tienen por finalidad acelerar la gestión de canal como paso previo a la transferencia de datos.

Los identificadores de canal se ponen en uso mediante **MCcf**, **AUcf**, **CJcf**, **CCcf**, y **CAin**; se suprimen mediante **MCcf**, **PCin**, **DUin**, **CLrq**, **CDin** y **CEin**. Los identificadores de testigo se ponen en uso mediante **MTcf**, **TGcf**, **TIcf**, y **TVin**; se suprimen mediante **MTcf**, **PTin**, **TRcf**, **TVrs**, y **TVcf**. Cuando se pone en uso un identificador en un proveedor dado, la MCSPDU que la causó se puede reenviar a cero, uno, varios, o todos los proveedores subordinados. La utilización de un identificador puede aumentar o reducirse gradualmente cuando, por ejemplo, usuarios individuales se admiten a un canal privado, o se excluyen de éste. Cuando se suprime un identificador en un proveedor dado, la MCSPDU que efectúa esto se reenvía a todos los subordinados que pueden estar todavía registrando el identificador como en uso.

El uso de un identificador está vinculado, en último término, a acciones que sobre un canal o testigo ejecuta un usuario anexionado al dominio (aunque puede haber algún retardo, como se ha explicado, al comunicar cambios mediante la transmisión de MCSPDU). Los identificadores registrados establemente como en uso en un proveedor MCS dado son los que algún usuario emplea activamente en el subárbol del proveedor. Por eso constituyen un subconjunto de los registrados establemente en cualquier proveedor más alto.

La supresión de un identificador de usuario tiene como consecuencia natural la supresión de los identificadores de canal y de testigo de los cuales es el único usuario en un subárbol.

En las subcláusulas que siguen se especifican los criterios seguidos para considerar en uso los identificadores de canal y los identificadores de testigo.

11.2 Información de canal

Cada una de las cuatro clases de canal tiene sus propios criterios para determinar si ha de considerarse que una anexión dada está usando un identificador de canal y, en consecuencia, si ha de estar representado en la base de información del proveedor:

a) Un identificador de canal estático (gama 1..1000) está en uso si el usuario se ha incorporado al canal con una confirmación MCS-INCORPORACIÓN-CANAL de éxito y no lo ha abandonado por una petición o indicación MCS-ABANDONO-CANAL.

b) Un canal de identificador de usuario está en uso si se ha asignado al usuario por una confirmación MCS-ANEXIÓN-USUARIO de éxito y el usuario no se ha desanexionado por una petición o indicación MCS-DESANEXIÓN-USUARIO.

c) Un identificador de canal privado está en uso si el usuario ha creado el canal con una confirmación MCS-FORMACIÓN-CANAL de éxito o ha sido admitido al mismo con una indicación MCS-ADMISIÓN-CANAL y no ha sido expulsado por una indicación MCS-EXCLUSIÓN-CANAL y el canal no ha sido disuelto por una petición o indicación MCS-DISOLUCIÓN-CANAL.

d) Un identificador de canal asignado está en uso si el usuario se ha incorporado al canal con una confirmación MCS-INCORPORACIÓN-CANAL de éxito y no lo ha abandonado por una petición o indicación MCS-ABANDONO-CANAL.

Para un identificador de canal en uso se registrará la siguiente información:

a) La clase de canal que representa (estático, de identificador de usuario, privado, o asignado).

b) Por cuáles anexiones MCS y por cuáles conexiones MCS a proveedores subordinados se ha efectuado la incorporación al canal.

c) Si se trata de un canal de identificador de usuario, el sentido de transmisión correspondiente al mismo, es decir, o bien la anexión MCS local a que se ha asignado el identificador de usuario, o la conexión MCS descendente a un proveedor subordinado en cuyo subárbol reside el usuario.

d) Si se trata de un identificador de canal privado, el identificador de usuario del gestor que lo convocó (esté o no el propio gestor en el subárbol del proveedor) y el conjunto de todos los identificadores de usuario en el subárbol del proveedor que ha sido admitido en el canal.

La información registrada sobre los identificadores de canal se emplea como se ha explicado en la cláusula 10 para validar las MCSPDU de petición y para encaminar las MCSPDU de indicación y de confirmación.

11.3 Información de testigo

Las transiciones de estado de un identificador de testigo se muestran en la Figura 11-1.

Un identificador de un testigo puede estar *tomado* por un solo usuario o *inhibido* por uno o más usuarios. La acción de **TVin** convierte el estado a *cediéndose* a lo largo de la rama de una jerarquía de dominio que va del proveedor MCS superior al receptor deseado. Este estado se degrada a *incedible* si el receptor se desanexiona antes de que el proveedor responda con **TVrs**. En cambio, pasa a *cedido* si el cedente libera explícitamente el testigo o se desanexiona. Durante la cesión de un testigo, la rama de una jerarquía de dominio que parte del cedente intersecta la rama que conduce al receptor al menos en el proveedor MCS superior. El estado del testigo cambia de *tomado* a *cediéndose*, y posiblemente más adelante a *incedible* o *cedido*, solamente a lo largo de esta intersección.

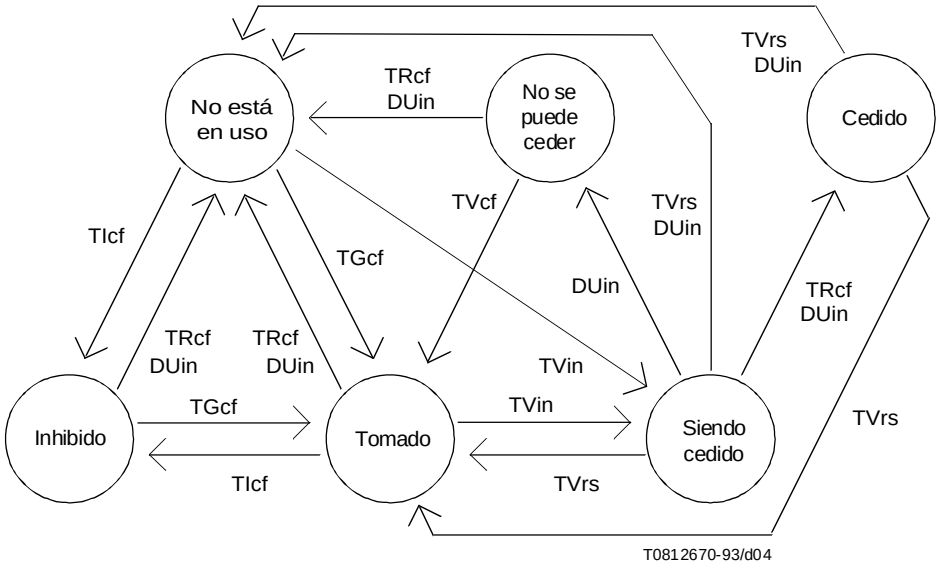


FIGURA 11-1/T.125

Transiciones de estado de un identificador de testigo

El usuario de un testigo y el testigo están en una relación de tomador, inhibidor, receptor, o en la de tomador y receptor (cuando el usuario se cede el testigo a sí mismo):

a) El usuario es un tomador si ha obtenido un testigo con una confirmación MCS-TOMA-TESTIGO de éxito y no lo ha liberado mediante una petición MCS-LIBERACIÓN-TESTIGO o una confirmación MCS-CESIÓN-TESTIGO de éxito, ni lo ha convertido con una confirmación MCS-INHIBICIÓN-TESTIGO de éxito o si ha aceptado un testigo ofrecido con una respuesta MCS-CESIÓN-TESTIGO de éxito.

b) El usuario es un inhibidor si ha obtenido un testigo con una confirmación MCS-INHIBICIÓN-TESTIGO de éxito y no lo ha liberado con una petición MCS-LIBERACIÓN-TESTIGO ni lo ha convertido con una confirmación MCS TOMA TESTIGO de éxito.

c) El usuario es un receptor si se le ha ofrecido un testigo por una indicación MCS-CESIÓN-TESTIGO y no lo ha liberado con una respuesta MCS-CESIÓN-TESTIGO de fracaso.

La siguiente información se registrará para un identificador de testigo en uso:

a) El estado del identificador de testigo en el proveedor MCS (no necesariamente idéntico a su estado en el proveedor superior).

b) Si el estado es tomado o incedible, el identificador de usuario del tomador en el subárbol del proveedor.

c) Si el estado es cediéndose, el identificador de usuario del receptor en el subárbol del proveedor, encuéntrese o no el tomador en el subárbol del proveedor.

d) Si el estado es cediéndose o cedido, el identificador de usuario del receptor en el subárbol del proveedor.

e) Si el estado es inhibido, el conjunto de todos los identificadores de usuario en el subárbol del proveedor que ha inhibido el testigo.

La información registrada para identificadores de testigo en uso se emplea como se ha explicado en la cláusula 10 para validar MCSPDU de respuesta y para encaminar MCSPDU de indicación.

El estado de un identificador de testigo en un proveedor subordinado no tiene que ser idéntico a su estado en el proveedor MCS superior. Esto se debe al hecho de que, por lo general, un cedente de testigo no procesa **TVin** ni **TVrs**, y de que, por lo general, un receptor no procesa la **TRcf** de un cedente. La Figura 11-2 muestra los estados que pueden presentarse en una interacción compleja de testigos.

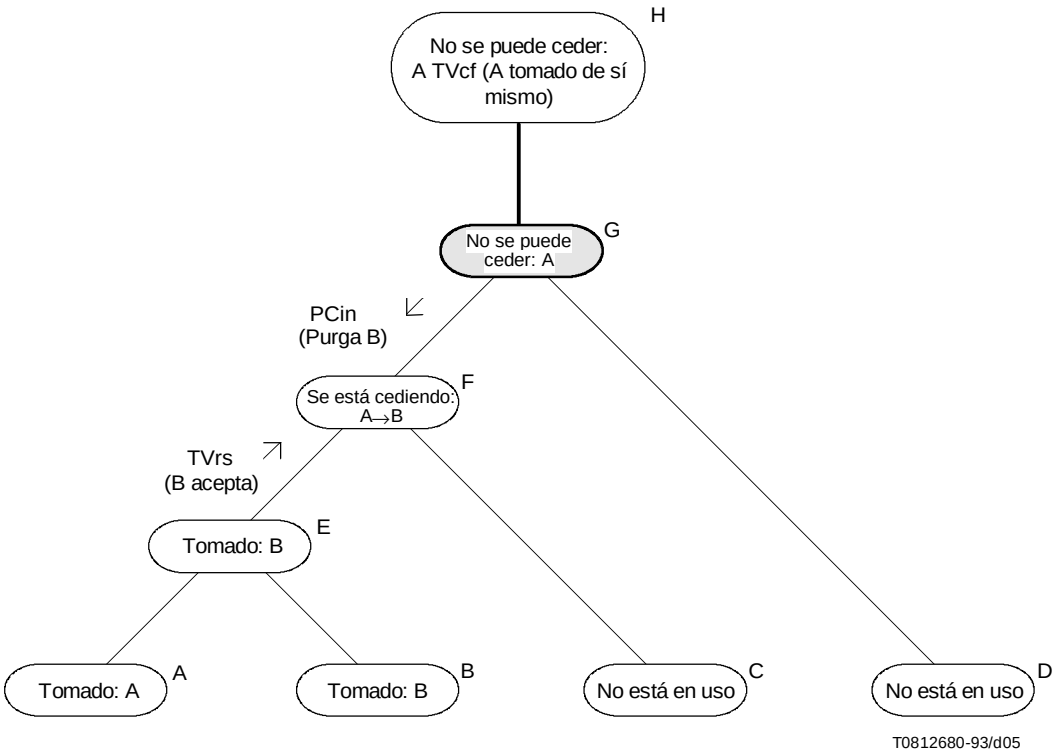


FIGURA 11-2/T.125

Estados de testigo que pueden presentarse en una interacción compleja

La figura destaca esencialmente el caso de un identificador de testigo en la base de información de los proveedores de A a H. Un antecedente plausible es que el usuario A, afectado al proveedor A, cedió el testigo al usuario B, afectado al proveedor B. Sin embargo, antes de que B pudiera responder, el proveedor G conectó el dominio a un nuevo proveedor superior H y comenzó una fusión. El nuevo dominio, al no poder aceptar al usuario B por existir un conflicto en los identificadores de canal, inició su purga mediante **PCin**. Esta MCSPDU se muestra habiendo recorrido parte de su trayectoria, a través de los proveedores G y D. El proveedor G, en consecuencia, ha ajustado el estado de su testigo pasándolo de *cediéndose* a *incedible*, y ha pasado (por fusión) el testigo, en este estado, al nuevo dominio. El nuevo proveedor superior H, al que se ha presentado un testigo en ese estado, ha introducido en la cola de transmisión una **TVcf** de fracaso, para retornar el testigo al usuario A. En el momento mostrado, el testigo ha sido también finalmente aceptado por el usuario B, y una **TVrs** está subiendo por la rama del receptor en la jerarquía del dominio, cambiando el estado del testigo en cada proveedor de *cediéndose* a *tomado* por B. Convergen ahora en el proveedor F dos MCSPDU, y la primera de ellas que llegue determinará su ulterior estado allí: o bien volverá a ser *tomado* por A, o pasará transitoriamente por el estado *tomado* por B y, después que el usuario B se haya desanexionado, será de nuevo *no en uso*. Tanto en uno como en el otro caso, el estado del testigo se estabilizará en *tomado* por A cuando el nuevo proveedor superior transmita la **TVcf** que tiene pendiente.

12 Elementos de procedimiento

12.1 Secuenciación de las MCSPDU

Las MCSPDU de control se mantienen en secuencia entre cualquier par de proveedores MCS porque se transmiten por una TC única: la TC inicial de una conexión MCS. Los proveedores MCS procesarán las MCSPDU recibidas y transmitirán cualesquiera MCSPDU de salida resultantes en el mismo orden en que las recibieron. Esto se aplica tanto a las MCSPDU que son simplemente reenviadas en sentido ascendente o descendente a través de la jerarquía del dominio,

como a las MCSPDU que son transformadas, por ejemplo, las peticiones y las respuestas se transforman respectivamente en indicaciones y confirmaciones. La secuenciación de las MCSPDU se mantendrá dentro de un proveedor MCS si es necesario para introducir en cola las salidas con miras a su ulterior transmisión, como consecuencia de la retropresión de control de flujo a lo largo de una TC.

Las MCSPDU de diferentes prioridades no tienen que mantenerse en secuencia. Por el contrario, la ventaja de las prioridades relativas sólo se pone de manifiesto cuando los datos de prioridad más alta son adelantados y sobrepasan a los de prioridad más baja. Ello significa que los datos deben transmitirse por TC separadas e introducirse en cola separadamente dentro de cada proveedor MCS. Si el número de prioridades de datos implementadas en un dominio es menor que el máximo, hay menos TC disponibles, pero los proveedores que optan por esto pueden mantener aun así, internamente, colas separadas. Con esto se obtienen algunas de las ventajas de la prioridad relativa, pero no todas.

Un proveedor MCS mantendrá la secuencia de las MCSPDU de control transmitidas con una prioridad dada. Esta es una limitación más estricta que la impuesta por la Recomendación UIT-T T.122, que sólo garantiza la secuenciación de unidades de datos de servicio transmitidas con una prioridad dada a través del mismo canal de destino.

Las MCSPDU de control y las de datos de más alta prioridad se transmiten a través de la misma TC inicial y serán tratadas con la misma atención por un proveedor MCS. Los datos de prioridades más bajas pueden rezagarse. Las indicaciones de control que sobrepasan a otras unidades pueden llegar antes que datos con prioridad más baja transmitidos primero.

12.2 Control de flujo de entrada

Los objetivos de un proveedor MCS a veces son contradictorios, por ejemplo: mantener los datos en rápido movimiento a través de un dominio pese a los bloqueos transitorios de las transmisiones hacia algunos receptores, dar a los transmisores un acceso equitativo a la anchura de banda disponible, y evitar que cualquiera de los participantes se quede demasiado rezagado con respecto a otros proveedores pares que están recibiendo los mismos datos multidifundidos. MCS es un servicio fiable que preserva la integridad de los datos de usuario. Dado que un proveedor tiene una capacidad limitada para almacenar las MCSPDU que no pueden transmitirse inmediatamente, tiene que disponer de un medio defensivo que le permita algunas veces rechazar ulteriores entradas. Si bien los detalles de la interfaz con el servicio de transporte es un asunto de índole local, el efecto, en un orden abstracto, deberá ser que las TSDU entrantes se mantengan en tuberías TC, intactas y en secuencia, para que se reciban posteriormente, cuando se haya dejado sin efecto el control de flujo. Cuando las tuberías TC se llenan, los proveedores MCS distantes pueden verse impedidos, como consecuencia de la retropresión, de transmitir ulteriores MCSPDU y tener que recurrir a un medio defensivo similar.

El control de flujo no se señala explícitamente en el protocolo MCS. Esta es una función de capas inferiores cuya duplicación sería inútil. Por tanto, es difícil detectar, a través de un medio de transmisión constituido por una TC participante, si un proveedor distante está oponiéndose a aceptar más entradas. Sin embargo, con el fin de satisfacer lo mejor posible objetivos contradictorios, se recomienda aplicar la política de control de flujo que se describe a continuación.

Un proveedor MCS puede asignar a cada TC entrante una cuota fija de memorias tampón que podrá llenar con MCSPDU antes de que se aplique retropresión. Cada memoria tampón se procesa como se especifica en este protocolo, después de lo cual se asigna para la salida de cero o más TC salientes. La salida puede producirse inmediatamente o demorarse porque una tubería de transporte esté llena. Mientras una memoria tampón no haya descargado sobre la última TC, se cargará contra la cuota de entrada de la TC por la que llegó dicha entrada. Una vez que haya descargado sobre todas las TC requeridas, será reciclada como un incremento de esa cuota. Cuando se ha agotado una cuota, se detienen las ulteriores entradas a través de la TC correspondiente. Las cuotas de entrada podrán fijarse atendiendo a que una TC forme parte de una conexión MCS ascendente o descendente, y a la prioridad de datos que representa.

Mediante el empleo de memorias tampón se puede mitigar los efectos de las diferencias entre las velocidades de los transmisores y los receptores. La fijación de cuotas a las entradas puede impedir que cualquiera de las TC monopolice los recursos. Puede también determinar hasta qué punto dos receptores de los mismos datos multidifundidos podrán estar fuera de paso. Sin embargo, el esquema recomendado no es lo suficientemente inteligente para prever todos los patrones de utilización y pudiera a veces reducir la velocidad de transferencia de datos a través de un dominio en situaciones en que existan alternativas aceptables. El establecimiento de mejores políticas de control de flujo (aplicables localmente y que no requieran la comunicación adicional de MCSPDU) podría ser un medio de conseguir una diferenciación del producto.

12.3 Cumplimentación del caudal

A diferencia del control de flujo de entrada, el protocolo MCS da cierto apoyo explícito a la cumplimentación del caudal. En primer lugar, la velocidad a que se cumplimenta el caudal es un parámetro de dominio que se negocia mediante MCS-CONEXION-PROVEEDOR. En segundo lugar, el intervalo de tiempo durante el cual se supervisa el caudal antes de ejecutar una acción adversa lo comunica cada proveedor MCS a su superior mediante **EDrq**. Para describir el intervalo de cumplimentación del caudal es necesario que los proveedores ajusten su comportamiento a ciertos principios comunes. De todas formas, la cumplimentación del caudal sigue siendo una técnica heurística con margen para la intervención.

La cumplimentación de una velocidad mínima de entrada en cada receptor es una opción de que disponen las aplicaciones de controlador. Esta opción se selecciona mediante el parámetro de dominio para el caudal cumplimentado, que se expresa en octetos por segundo. Si bien parece evidente que no se debe permitir a un participante funcionar a una velocidad arbitrariamente baja y obstaculizar así la transferencia de datos entre los otros, el tratar de cumplimentar un caudal de una manera demasiado estricta entraña un peligro.

Los complejos patrones de utilización de múltiples transmisores constituyen un problema. Las clases de retropresión contra las que reacciona una política de cumplimentación del caudal pueden no ser el resultado de un solo receptor anómalamente lento. En primer lugar, las MCSPDU transmitidas en sentido descendente deben competir, para ser atendidas, con las transmitidas en sentido ascendente, pero son reflejadas en retorno, sobre todo **SDin**. El flujo descendente experimentado por un proveedor par puede por tanto alcanzar sólo una fracción de la anchura de banda nominal y puede variar dinámicamente con el número de las otras conexiones y anexiones al proveedor par. En segundo lugar, cabe esperar que sólo las MCSPDU de máxima prioridad fluyan continuamente por una conexión MCS. Las de prioridades más bajas pueden, de hecho, quedar bloqueadas por un proveedor par durante largos periodos de tiempo debido a la intensidad del tráfico recibido de otras fuentes con prioridades más altas. Por último, puede haber una gran variación en el caudal instantáneo si éste se mide por el tiempo que tiene que esperar una MCSPDU bloqueada para ser aceptada y pasar a una conexión MCS. Esto puede depender, entre otras cosas, del número de MCSPDU procedentes de otras fuentes que son introducidas en cola en el proveedor par, y en qué orden esto se efectúa.

No obstante, la cumplimentación del caudal es una opción valiosa en situaciones prácticas en las que se sabe que los patrones de transferencia de datos son más uniformes. Debe interpretarse como que requiere la aplicación de una salida mínima de MCSPDU a cada anexión MCS directa y a cada conexión MCS descendente durante un intervalo de tiempo que será especificado por el proveedor MCS cumplimentante. Cada salida de MCSPDU, tanto de control como de datos, deberá contar, a los efectos del caudal, como si fuera del tamaño máximo permitido por los parámetros del dominio. La salida de una MCSPDU aplicada a una anexión MCS deberá entrañar la entrega de la indicación o confirmación de primitiva conexa. La salida aplicada a una conexión MCS descendente entrañará la ausencia de retropresión en la interfaz del servicio de transporte y la aceptación para pasar a la correspondiente tubería TC.

La salida hacia una anexión dada o hacia una conexión descendente se deberá supervisar mientras exista una o más MCSPDU en la cola, independientemente de la prioridad de los datos. Cuando las colas se vacían, termina la supervisión, sin que se ejecuten acciones de cumplimentación del caudal. Al mismo tiempo, no se registrará ningún mérito por buena conducta para compensar futuros rezagos. La supervisión se reanudará tan pronto como la retropresión impida la obtención de cualquier MCSPDU a la salida, y, en lugar de ello, haya que introducirla en una cola. Mientras una o más MCSPDU permanezcan en la cola, el número de unidades efectivamente obtenidas a la salida se contará en un intervalo fijo de tiempo.

Cada proveedor MCS elegirá un intervalo de cumplimentación del caudal. El intervalo será lo suficientemente largo para que por lo menos una MCSPDU de tamaño máximo pueda obtenerse a la salida en el caudal mínimo. Un proveedor MCS informará a su superior sobre el intervalo elegido y sus cambios ulteriores, transmitiendo **EDrq** hacia arriba. Un proveedor actuará contra la anexión MCS o la conexión MCS descendiente infractora una vez terminado cualquier intervalo durante el cual el caudal supervisado no se ajuste a su comportamiento esperado. El proveedor desafectará al usuario o desanexionará la conexión.

Los proveedores más altos pueden fijar su intervalo de cumplimentación del caudal de modo que sea más largo que el de cualquier subordinado, más cierto margen de tiempo de reacción. Esto tiene por finalidad incitar a que las acciones de cumplimentación del caudal se ejecuten en primer lugar en el proveedor más bajo cuya posición le permita detectar el problema. Cuando, después del intervalo de algún proveedor más bajo, se elimina un infractor, los proveedores más altos deben tener tiempo suficiente para detectar el restablecimiento del caudal a los niveles adecuados. Si actúan con demasiada rapidez, pudieran penalizar a un subárbol que fuera mayor de lo necesario, y perjudicar a otros que no sean infractores en el dominio.

Las aplicaciones de controlador que establecen parámetros del dominio deberán ser conservadoras en sus pretensiones, y prever que el caudal pueda ocasionalmente sufrir bajas durante periodos de intensos estímulos de aplicación. Si su

preocupación es simplemente dar protección contra receptores que dejan de aceptar todo, por completo, pueden fijar el caudal mínimo muy bajo. Sabiendo el tamaño máximo de las MCSPDU y el caudal cumplimentado, los controladores pueden calcular el intervalo mínimo que debe transcurrir antes de que se detecte y resuelva un bloqueo.

12.4 Configuración de los dominios

La Recomendación UIT-T T.122 no proporciona ningún mecanismo para configurar el conjunto de los dominios soportados por un proveedor MCS. Deberá considerarse que esto es un asunto local cuya normalización queda en estudio. Este protocolo presupone que un proveedor MCS reconocerá algunos selectores de dominio como válidos y otros como inválidos. Prevé que los selectores de dominio se comuniquen como parte del establecimiento de una conexión MCS.

Un proveedor MCS participa implícitamente en la negociación de parámetros de dominio. Sea el lado llamante o el llamado, restringe la gama de valores de parámetro admitidos de acuerdo con los límites para los que está configurado el dominio. Un proveedor MCS congelará la negociabilidad de los parámetros de dominio cuando cualquier usuario se haya anexionado a un dominio o cuando se haya establecido la primera conexión MCS.

12.5 Fusión de dominios

Se fusionan dominios como consecuencia de MCS-CONEXIÓN-PROVEEDOR. Si es fácil conseguir que uno u otro dominio esté vacío en este punto, la fusión no presentará gran complicación. En el caso más general, sin embargo, debe preverse la actualización de la base de información del proveedor superior restante de modo que contenga la base de información del antiguo proveedor superior, y la solución de cualquier conflicto que surja. En la cláusula 10 se da una información más detallada al respecto.

Para facilitar la comprensión, en el grupo de Figuras 12-1 a 12-4 se presenta un ejemplo de fusión de dominios. En este ejemplo, el proveedor E representa un antiguo proveedor superior que se ha incorporado a un nuevo dominio mediante la conexión MCS (dibujada en trazo más grueso) con un proveedor intermedio F. Es intrascendente que la conexión haya sido iniciada por el proveedor E o el proveedor F.

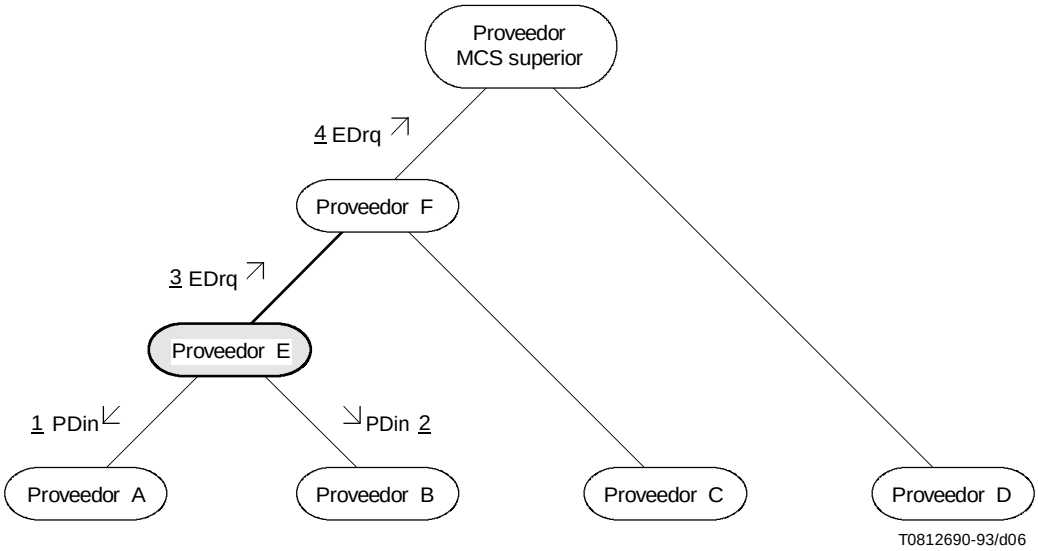


FIGURA 12-1/T.125

Primera etapa de la fusión de dominios – Establecimiento de la jerarquía

El proveedor E, al observar que se encuentra en el extremo inferior de una conexión ascendente, asume la responsabilidad de la ejecución de la fusión. Como no es seguro efectuar una actividad cuando la base de información está activa, el proveedor E detiene la aceptación de entradas de su subárbol. Toda MCSPDU que se encuentre ya en tránsito procedente de los proveedores A y B o las nuevas generadas antes de completarse la fusión serán salvaguardadas y procesadas posteriormente. En cambio, no se impide el flujo descendente de MCSPDU, hayan sido generadas por el proveedor E o reenviadas desde más arriba.

Hasta que la fusión se completa, las únicas MCSPDU de confirmación que el proveedor E recibirá son **MCcf** y **MTcf,** pues no se permiten peticiones de usuario ascendentes. Estas confirmarán o purgarán los identificadores de canal y de testigo ya en uso. Las MCSPDU de indicación **PCin, PTin, DUin** y **CDin** del dominio superior pueden llegar también, suprimiendo identificadores de canal y de testigo del dominio más bajo cuya fusión ha sido confirmada individualmente. Los identificadores no confirmados del dominio más bajo están protegidos contra la supresión porque significan algo diferente con respecto al mismo identificador del dominio superior. **PDin** debe ser obedecida para cumplir el límite de altura del dominio. Las indicaciones restantes no tienen que ser aplicadas por el proveedor E mientras se está efectuando la fusión. En particular, no deben fluir transferencias de datos entre los dominios separados anteriormente hasta que la fusión se completa; no pueden fluir hasta que hayan sido clasificados al menos los canales que las transportan. Puede no ser conveniente aplicar dos indicaciones que pueden poner en uso nuevos identificadores. Para mantener la coherencia de la base de información, si el proveedor E rechaza **CAin** o **TVin**, reaccionará como se especifica en la cláusula 10.

Las primeras acciones del proveedor E son enviar **PDin** hacia abajo, cerciorarse de que la última conexión MCS no haya creado un ciclo que invalidaría el principio de que cada jerarquía de dominio tenga exactamente un proveedor superior, y enviar **EDrq** hacia arriba para comunicar su altura y su intervalo de cumplimentación de canal. El proveedor F, al ascender de ese modo de la altura 2 a la 3, pasa de largo **EDrq** al proveedor superior, que alcanza entonces la altura 4.

En la segunda etapa de la fusión de dominios, el proveedor E envía hacia arriba cuantas instancias de **MCrq** sean necesarias para contener los identificadores de usuario que existen en su base de información. Los identificadores de usuario que no estén en conflicto con el dominio superior se confirman y las restantes se purgan en el mismo número de instancias de **MCcf**. En base a **MCcf**, el proveedor E genera **PCin** para informar sobre las purgas que se hayan efectuado a todo lo largo de su subárbol. Esta etapa termina cuando todos los identificadores de usuario hayan sido explícitamente confirmados o purgados.

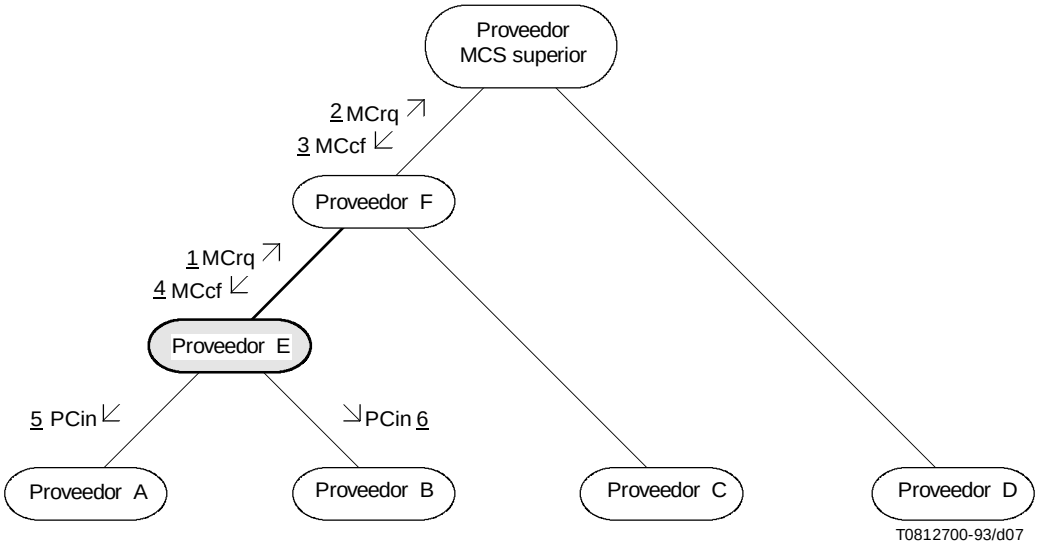


FIGURA 12-2/T.125

Segunda etapa de la fusión de dominios – Fusión de los canales de identificador de usuario

La tercera etapa es similar a la segunda; se diferencian en que ésta se refiere a los identificadores de testigo en lugar de los de canal, y se utiliza un conjunto paralelo de MCSPDU. Si los identificadores de usuario no se hubiesen fusionado previamente, porciones de una ulterior **MTrq** podrían rechazarse como inválidas y los identificadores de testigo influidos serían innecesariamente purgados. En el caso de un testigo inhibido, es posible que el conjunto completo de usuarios inhibidores no quepan en una sola MCSPDU. El proveedor E espera por la confirmación del primer subconjunto que envía hacia arriba antes de volver a enviar el testigo inhibido con los identificadores de usuario restantes. De esta forma se evita que el primer conjunto se rechace porque se esté usando demasiados identificadores de testigo y que, no obstante, se acepten después los restantes, lo que adulteraría la base de información. Esta etapa termina cuando todos los identificadores de testigo han sido explícitamente confirmados o purgados.

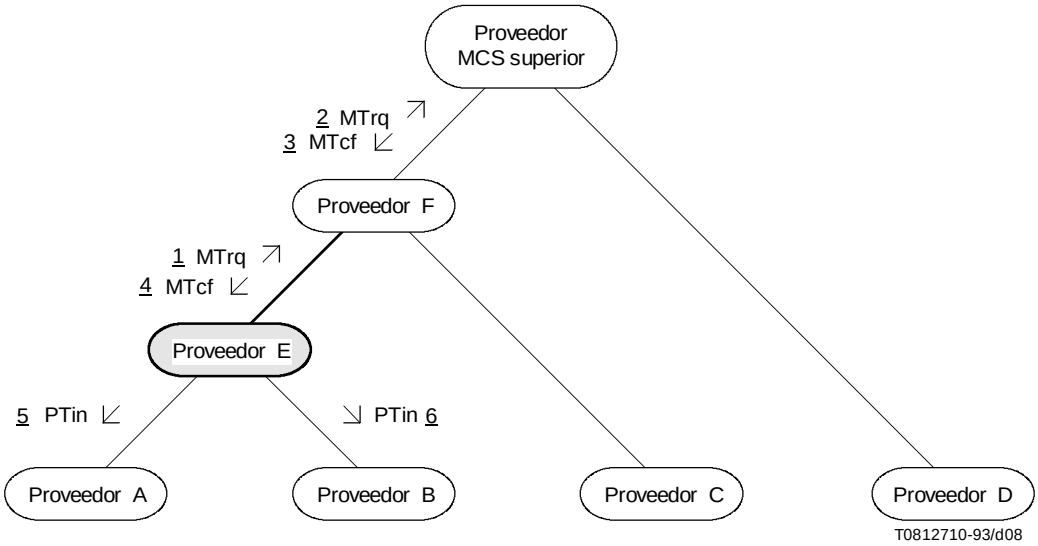


FIGURA 12-3/T.125

Tercera etapa de la fusión de dominios – Fusión de los identificadores de testigo

En la cuarta etapa se utilizan las mismas MCSPDU que en la segunda, pero contienen diferentes identificadores de canal. Una vez tratados los canales de identificador de usuario, los identificadores restantes son los de canales estáticos, privados, y asignados. Si los identificadores de usuario no se hubiesen fusionado previamente, porciones de una ulterior **MCrq** podrían rechazarse como inválidos y los identificadores de canal influidos serían innecesariamente purgados. En el caso de un canal privado, es posible que el conjunto completo de usuarios admitidos por el gestor del canal no quepan en una sola MCSPDU. El proveedor E espera por la confirmación del primer subconjunto que envía hacia arriba antes de volver a enviar el canal privado con los identificadores de usuario restantes. De esta forma se evita que el primer conjunto se rechace porque se esté usando demasiados identificadores de canal y que, no obstante, se acepten después los restantes, lo que adulteraría la base de información. Esta etapa termina cuando todos los identificadores de testigo han sido explícitamente confirmados o purgados.

NOTA – Cuando los identificadores de testigo han sido previamente fusionados, su posesión exclusiva es un medio más eficaz para resolver los conflictos de flujos de datos. En otro caso, mientras se confirman los identificadores de canal, y antes de que el conflicto haya sido revelado por una purga de testigos, pudiera haber fugas de datos entre los dominios.

12.6 Desconexión de dominio

Cuando se desconecta una conexión MCS ascendente, un proveedor MCS eliminará su árbol del dominio desanexionando todas sus anexiones MCS directas y desconectando todas sus otras conexiones MCS. El proveedor influido no puede, en general, establecer un dominio residual en su propio subárbol, porque no dispone de un registro de las MCSPDU de petición que se enviaron hacia arriba y para las cuales nunca recibirá la correspondiente MCSPDU de confirmación.

Cuando se desconecta una conexión MCS descendente, un proveedor generará MCSPDU **DUrq** para todos los usuarios que residen en esa porción de su subárbol, expresando como motivo *dominio desconectado*.

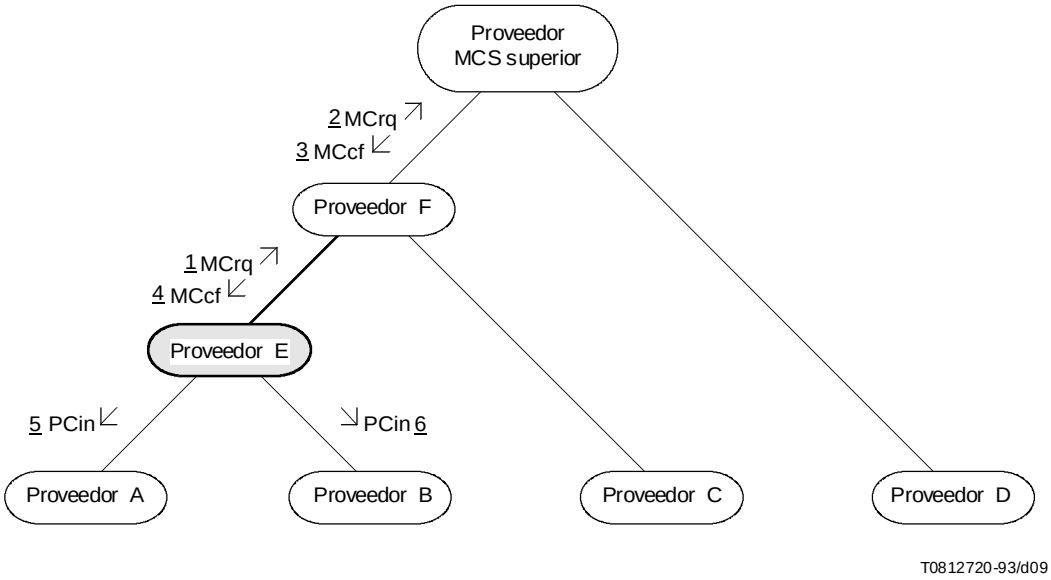


FIGURA 12-4/T.125

Cuarta etapa de la fusión de dominios – Fusión de los restantes identificadores de canal

12.7 Atribución de identificadores de canal

Los identificadores de canal en la gama de 1001 en adelante se atribuyen dinámicamente en el proveedor MCS superior durante el procesamiento de peticiones de primitivas MCS-ANEXIÓN-USUARIO, MCS-INCORPORACIÓN-CANAL cero, y MCS-FORMACIÓN-CANAL. No se exige que los valores atribuidos se ajusten a un determinado patrón. Más bien es conveniente que los valores estén dispersados al azar en toda la gama permitida. Esto hace más probable que dos dominios que operan independientemente durante un periodo considerable de tiempo puedan fusionarse posteriormente sin conflictos en sus respectivas atribuciones de identificadores de canal. Con esto se obtiene también protección contra un reciclado demasiado rápido de los identificadores dentro de un dominio, pues se liberan de un uso y se reatribuyen después para un uso diferente. Las aplicaciones que emplean MCS deben tener tiempo para adaptarse a la desaparición de un identificador de canal o de usuario, antes de que el identificador reaparezca.

En los casos en que se requiere una fusión «inconsutil» de dominios activos, se puede evitar conflictos seleccionando identificadores de canal de una subgama única de cada proveedor. Las subgamas pueden ser creadas dividiendo identificadores de canal atribuidos dinámicamente 1001,..65535 en varias bandas. Además, dentro de una subgama, se pueden asignar secuencialmente identificadores en un sentido, desde arriba o desde abajo. Los proveedores que emplean asignación de subgama obedecerán aún a todos los aspectos de este protocolo, incluido el procedimiento para la fusión de dominios. De este modo, acomodarán proveedores pares que pueden carecer de subgama únicas.

NOTA – Corresponde a la realización local decidir el modo en que un proveedor elige una subgama a partir de la cual efectuar atribuciones. Las conferencias organizadas previamente pueden ser agrupadas por un sistema de gestión de red.

Los identificadores de testigo, a diferencia de los de canal, no son atribuidos, por lo que la línea de demarcación representada por el valor 1001, no tiene ningún significado para ellos. Un identificador de testigo libre con un valor dado está estáticamente disponible para ser tomado o inhibido en cualquier momento, a condición de que se respete el límite impuesto por el dominio al número total de identificadores que se estén utilizando a la vez.

12.8 Estado del testigo

El estado del testigo se define formalmente en la cláusula 7. Se retorna como un componente de las MCSPDU de confirmación de testigo y se utiliza para actualizar el registro para un identificador de testigo en la base de información de los proveedores subordinados. El estado del testigo no tiene que comunicarse directamente mediante una primitiva de confirmación al usuario iniciador, sino que puede comunicarse indirectamente mediante un valor de resultado.

Cuando más de un valor de estado del testigo describen la relación de un usuario dado con un identificador de testigo especificado, el orden de preferencia debe ser el siguiente: *Recibiente de sí mismo* debe comunicarse primero, si cabe, para recordar al usuario que se debe responder a una indicación MCS-CESIÓN-TESTIGO. El siguiente valor preferido es *cediéndose así mismo* para recordar al usuario que se está actuando en una operación no completada, y el siguiente es *tomado por sí mismo* o *inhibido por sí mismo*. Los últimos en el orden de preferencia son los restantes valores de estado, que reflejan el estado actual del testigo atendiendo exclusivamente a su uso por otros participantes.

Un proveedor MCS se basa en la preferencia especificada con respecto a los valores de estado de testigo para actualizar correctamente el estado del testigo en su base de información.

13 Realización de referencia

Los Apéndices II a VII contienen una realización de un proveedor MCS en lenguaje de especificación y descripción (SDL, s*pecification and description language*). Estos apéndices muestran que el protocolo especificado puede realizarse con un esfuerzo razonable. Este ejemplo tiene por objeto ahorrar a los realizadores el trabajo de reinventar una lógica equivalente, y acelerar la introducción de sistemas compatibles.

SDL es una técnica de descripción formal más potente que las tablas de estados convencionales y mejor adaptada a la complejidad de MCS. Admite dos representaciones equivalentes, la textual y la gráfica. Para la realización de referencia se ha utilizado la representación textual, que es más concisa y fácil de trasladar a un lenguaje de programación convencional. Se utilizan profusamente tipos de datos abstractos como el «powerset generator» (generador de superconjuntos). Es posible que los lectores que no estén familiarizados con SDL deseen consultar las siguientes referencias:

- Recomendación Z.100 del CCITT (1988), *Lenguaje de Especificación y Descripción (SDL)*.
- Belina, Hogrefe, y Sarma: *SDL with Applications from Protocol Specification* (Prentice Hall, 1991), ISBN 0-13-785890-6.
- Belina y Hogrefe: *The CCITT Specification and Description Language SDL*, Computer Networks and ISDN Systems, 16 (1988/89), páginas 311-341.

El Apéndice II comienza con una figura que resume las relaciones codificadas en las definiciones de sistema y de bloque que siguen. Los procesos que aparecen en la figura – «control», «domain» (dominio), «endpoint» (punto extremo), y «attachment» (anexión) – se definen individualmente en los Apéndices subsiguientes III a VI. El Apéndice VII explica las hipótesis de que se ha partido en el modelo e ilustra los flujos de señales fundamentales.

Los apéndices han sido examinados por expertos en la materia y estimamos que constituyen una realización correcta del protocolo MCS definido en esta especificación, pero no son normativos. En caso de discrepancia, deberán prevalecer las descripciones contenidas en la parte principal de esta especificación.

El protocolo MCS se puede también realizar mediante otros diseños, que no tienen que basarse necesariamente en la realización de referencia.

La realización de referencia de los Apéndices II a VII está parametrizada de modo que se puede ampliar a un proveedor MCS de carácter general. Otras realizaciones limitadas a un caso especial pueden ser más compactas y eficientes. De particular interés es el caso de un proveedor MCS limitado a una conexión MCS, una condición apropiada para un nodo terminal. Este podría especializarse aún más para simplificar las fusiones mediante una purga voluntaria de los identificadores que se han utilizado en un dominio más bajo. La selección de casos especiales importantes y su realización en SDL es un tema que queda en estudio.

Apéndice I

Codificaciones alternativas de una MCSPDU

(Este apéndice no es parte integrante de la presente Recomendación)

I.1 Petición envío datos

Las definiciones pertinentes, tomadas de la cláusula 7, son:

```
DomainMCSPDU    ::= CHOICE
{
    . . .
    sdrq        SDrq,
    . . .
}

SDrq ::= [APPLICATION 25] IMPLICIT SEQUENCE
{
    initiator               UserId,
    channelId               ChannelId,
    dataPriority            DataPriority,
    segmentation            Segmentation,
    userData                OCTET STRING
}

UserId              ::= DynamicChannelId

DynamicChannelId    ::= ChannelId (1001..65535)

ChannelId           ::= INTEGER (0..65535)

DataPriority        ::= ENUMERATED
{
                        top             (0),
                        high            (1),
                        medium          (2),
                        low             (3)
}

Segmentation        ::= BIT STRING
{
                        begin           (0),
                        end             (1)
} (SIZE (2))
```

Un ejemplo de valor de este tipo es:

```
sdrq
{
    initiator               1701,
    channelId               5,
    dataPriority            high,
    segmentation            {begin},
    userData                '4D4353'H
}
```

I.2 Reglas de codificación básica (BER, *basic encoding rules*)

BER aplica recursivamente un esquema de *identificador*, *longitud*, *contenido* a tipos componentes:

SDrq	Length	Contents		
79	13			
		INTEGER	Length	Contents
		02	02	06 A5
		INTEGER	Length	Contents
		02	01	05

ENUMERATED	Length	Contents
0A	01	01
BIT STRING	**Length**	**Contents**
03	02	06 80
OCTET STRING	**Length**	**Contents**
04	03	4D 43 53

Octetos de encabezamiento excluidos datos de usuario: 18-24, en función del rótulo de la MCSPDU, el identificador de canal, y la longitud de los datos de usuario.

I.3 Reglas de codificación empaquetada (PER, *packed encoding rules*)

PER sólo puede decodificarse si se sabe de antemano qué tipo ASN.1 está contenido en la codificación, pues no se transportan rótulos. Esta es una razón para definir un tipo combinado de MCSPDU de dominio. Los rótulos de aplicación de las MCSPDU de dominio avanzan secuencialmente a partir de cero. Esto es conveniente porque PER codifica valores como distancias con respecto a la base de su gama. Con este convenio, el valor 25 identifica **SDrq** tanto en BER como en PER:

CHOICE	-- *6 bits + pad*
64	
INTEGER (1001..65535)	-- *offset 1001*
02 BC	
INTEGER (0..65535)	-- *offset 0*
00 05	
ENUMERATED + BIT STRING (SIZE (2))	-- *2 bits + 2 bits + pad*
60	
CTET STRING	-- *length + contents*
03 4D 43 53	

Octetos de encabezamiento excluidos datos de usuario: 7-8, en función de la longitud de los datos de usuario.

PER codifica una cadena de octetos, sin restricciones, de una longitud de hasta 127, en un octeto, y de una longitud de hasta 16 384 en dos octetos.

Apéndice II

Descomposición en SDL de un proveedor MCS

(Este apéndice no es parte integrante de la presente Recomendación)

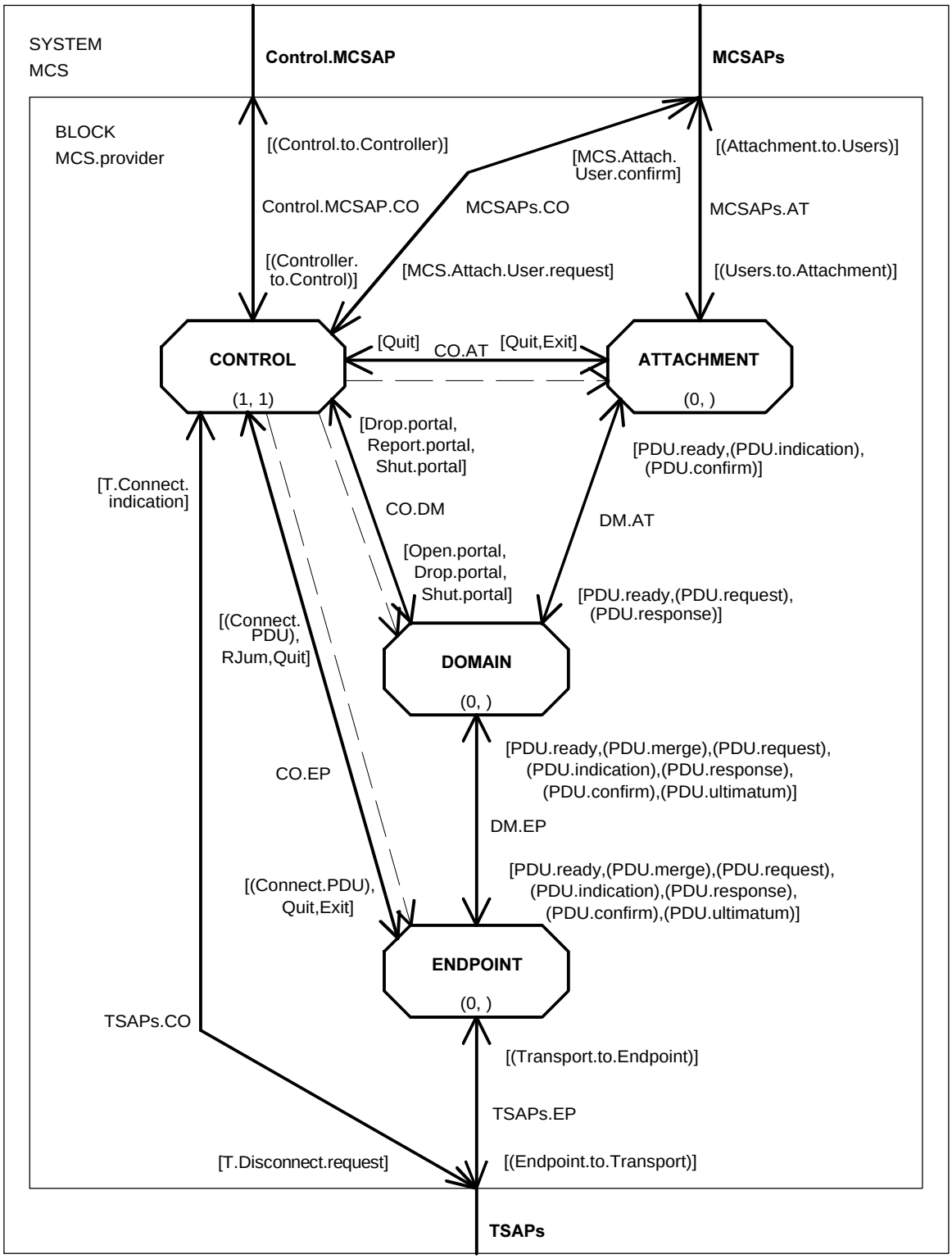


FIGURA II.1/T.125

Descomposición en SDL de un proveedor MCS

```
SYSTEM  MCS;

SYNONYM oneSecond               Duration  =  1000;                    /* time is in milliseconds */

        /* Type definitions */

SYNTYPE     ChannelId                   =  Integer CONSTANTS 0:65535
ENDSYNTYPE;

NEWTYPE     UserId                      INHERITS ChannelId
OPERATORS ALL;
ADDING
OPERATORS
        UserId:     ChannelId       –> UserId;      /* type cast */
        ChannelId:  UserId          –> ChannelId;   /* type cast */

AXIOMS
        UserId(0)  =  0;
        FOR ALL c in ChannelId
        (
                ChannelId(UserId(c))  =  c;
        );
ENDNEWTYPE;

SYNTYPE         TokenId                 =  Integer CONSTANTS 1:65535
ENDSYNTYPE;

NEWTYPE     ChannelIdSet        SetOf(ChannelId);
ENDNEWTYPE;
NEWTYPE     UserIdSet           SetOf(UserId);
ENDNEWTYPE;
NEWTYPE     TokenIdSet          SetOf(TokenId);
ENDNEWTYPE;

NEWTYPE     TokenStatus
LITERALS
        NotInUse,
        SelfGrabbed,
        OtherGrabbed,
        SelfInhibited,
        OtherInhibited,
        SelfRecipient,
        SelfGiving,
        OtherGiving;
ENDNEWTYPE;

SYNTYPE     DataPriority        =  Integer CONSTANTS 0:3
ENDSYNTYPE;

NEWTYPE     Segmentation
STRUCT
        begin                   Boolean;
        end                     Boolean;
ENDNEWTYPE;

NEWTYPE         DomainParameters
STRUCT
        maxChannelIds           Natural;
        maxUserIds              Natural;
        maxTokenIds             Natural;
        numPriorities           Natural;
        minThroughput           Natural;
        maxHeight               Natural;
        maxMCSPDUsize           Natural;
        protocolVersion         Natural;
ENDNEWTYPE;

SYNTYPE         DomainSelector      =  OctetString
ENDSYNTYPE;

SYNTYPE         TSAPAddress         =  OctetString
ENDSYNTYPE;
```

```
NEWTYPE         TransportQOS            /* quality of service */
STRUCT
        throughput      Natural;                /*octets per second */
        transitDelay    Duration;               /* one-way */
        dataPriority    Natural;                /* 0 is highest */
ENDNEWTYPE;

NEWTYPE     TransportQOSByPri       Array(DataPriority, TransportQOS);
ENDNEWTYPE;

SYNTYPE     UserData    =  OctetString
ENDSYNTYPE;

SYNTYPE     TSDU                =  OctetString
ENDSYNTYPE;

SYNTYPE     Octet               =  Integer  CONSTANTS  0:255
ENDSYNTYPE;

NEWTYPE     OctetString         String(Octet, NullString);
ENDNEWTYPE;

GENERATOR SetOf (TYPE ItemType)     /* subsets with choice operator */
/*AS*/
        Powerset(ItemType);
ADDING
OPERATORS
        Pick:   SetOf           --> ItemType;   /* chooses any element */
AXIOMS
        Pick(Empty)  ==  ERROR!;
        FOR ALL s IN SetOf
        (
                s /= Empty ==> Pick(s) in s;
        );
DEFAULT
        Empty;
ENDGENERATOR;

NEWTYPE     Reason
LITERALS
        RN_domain_disconnected,
        RN_provider_initiated,
        RN_token_purged,
        RN_user_requested,
        RN_channel_purged,
        RN_channel_disbanded,                   /* not in MCSPDUs */
        RN_domain_not_hierarchical,         /* not in MCSPDUs */
        RN_parameters_unacceptable,         /* not in MCSPDUs */
        RN_unspecified;                     /* not in MCSPDUs */
ENDNEWTYPE;

NEWTYPE     Result

LITERALS
        RT_successful,
        RT_domain_merging,
        RT_domain_not_hierarchical,
        RT_no_such_channel,
        RT_no_such_domain,
        RT_no_such_user,
        RT_not_admitted,
        RT_other_user_id,
        RT_parameters_unacceptable,
        RT_token_not_available,
        RT_token_not_possessed,
        RT_too_many_channels,
        RT_too_many_tokens,
        RT_too_many_users,
```

```
        RT_unspecified_failure,
        RT_user_rejected,
        RT_congested,                   /* not in MCSPDUs */
        RT_domain_disconnected;         /* not in MCSPDUs */
ENDNEWTYPE;

        /* The next three identifier types distinguish separate instances
        of communication across the interface between an MCS provider and
        its environment.  MCSConnectionId maps to some resource within the
        Control process.  MCSAttachmentId, which equals the process id of
        an Attachment process, could be considered implicit, since it is
        the source or destination address of corresponding signals, but
        the usage is clearer when it is made an explicit signal parameter.
        The same model is assumed for TCEndpointId. */

SYNTYPE         MCSConnectionId  =  Natural
ENDSYNTYPE;

SYNTYPE         MCSAttachmentId  =  PId
ENDSYNTYPE;

SYNTYPE         TCEndpointId    =  PId
ENDSYNTYPE;

        /* Block decomposition */

BLOCK   MCS.provider    REFERENCED;

CHANNEL Control.MCSAP
        FROM ENV TO MCS.provider WITH
                (Controller.to.Control);
        FROM MCS.provider TO ENV WITH
                (Control.to.Controller);

ENDCHANNEL;

SIGNALLIST Controller.to.Control  =
        MCS.Connect.Provider.request,
        MCS.Connect.Provider.response,
        MCS.Disconnect.Provider.request;

SIGNALLIST Control.to.Controller  =
        MCS.Connect.Provider.indication,
        MCS.Connect.Provider.confirm,
        MCS.Disconnect.Provider.indication;

SIGNAL  MCS.Connect.Provider.request
        (
                Natural,                /* requester's label */
                TSAPAddress,            /* calling */
                DomainSelector,
                TSAPAddress,            /* called */
                DomainSelector,
                Boolean,                /* upward */
                DomainParameters        /* target */
                DomainParameters,       /* minimum */
                DomainParameters,       /* maximum */
                TransportQOSByPri,      /* target */
                TransportQOSByPri,      /* minimum */
                UserData

        );
SIGNAL  MCS.Connect.Provider.indication
        (
                MCSConnectionId,        /* provider-assigned */
                TSAPAddress,            /* calling */
                DomainSelector,
                TSAPAddress,            /* called */
                DomainSelector,
```

```
                Boolean,                    /* upward */
                DomainParameters,           /* target */
                DomainParameters,           /* minimum */
                DomainParameters,           /* maximum */
                UserData

        );
SIGNAL  MCS.Connect.Provider.response
        (
                MCSConnectionId,
                Result,
                DomainParameters,
                UserData

        );
SIGNAL  MCS.Connect.Provider.confirm
        (
                Natural,                    /* requester's label */
                Result,
                MCSConnectionId,            /* provider-assigned */
                DomainParameters,
                UserData

        );

SIGNAL  MCS.Disconnect.Provider.request
        (

                MCSConnectionId
        );
SIGNAL  MCS.Disconnect.Provider.indication
        (
                MCSConnectionId,
                Reason

        );

CHANNEL MCSAPs
        FROM ENV TO MCS.provider WITH
                MCS.Attach.User.request,
                (Users.to.Attachment);
        FROM MCS.provider TO ENV WITH
                (Attachment.to.Users);
ENDCHANNEL;

SIGNALLIST Users.to.Attachment  =
        MCS.ready,
        MCS.Detach.User.request,
        MCS.Channel.Join.request,
        MCS.Channel.Leave.request,
        MCS.Channel.Convene.request,
        MCS.Channel.Disband.request,
        MCS.Channel.Admit.request,
        MCS.Channel.Expel.request,
        MCS.Send.Data.request,
        MCS.Uniform.Send.Data.request,
        MCS.Token.Grab.request,
        MCS.Token.Inhibit.request,
        MCS.Token.Give.request,
        MCS.Token.Give.response,
        MCS.Token.Please.request,
        MCS.Token.Release.request,
        MCS.Token.Test.request;

SIGNALLIST Attachment.to.Users =
        MCS.ready,
        MCS.Attach.User.confirm,
        MCS.Detach.User.indication,
        MCS.Channel.Join.confirm,
        MCS.Channel.Leave.indication,
        MCS.Channel.Convene.confirm,
```

```
        MCS.Channel.Disband.indication,
        MCS.Channel.Admit.indication,
        MCS.Channel.Expel.indication,
        MCS.Send.Data.indication,
        MCS.Uniform.Send.Data.indication,
        MCS.Token.Grab.confirm,
        MCS.Token.Inhibit.confirm,
        MCS.Token.Give.indication,
        MCS.Token.Give.confirm,
        MCS.Token.Please.indication,
        MCS.Token.Release.confirm,
        MCS.Token.Test.confirm;

SIGNAL  MCS.ready                       /* allows one MCS.[Uniform.]Send.Data */
        (
                MCSAttachmentId,
                DataPriority

        );

SIGNAL  MCS.Attach.User.request
        (
                Natural,                /* requester's label */
                DomainSelector

        );

SIGNAL  MCS.Attach.User.confirm
        (
                Natural,                /* requester's label */
                Result,
                MCSAttachmentId,        /* provider-assigned */
                UserId

        );

SIGNAL  MCS.Detach.User.request
        (
                MCSAttachmentId
        );
SIGNAL  MCS.Detach.User.indication
        (
                MCSAttachmentId,
                UserId,
                Reason

        );
SIGNAL  MCS.Channel.Join.request
        (
                MCSAttachmentId,
                ChannelId

        );
SIGNAL  MCS.Channel.Join.confirm
        (
                MCSAttachmentId,
                ChannelId,              /* requested */
                Result,
                ChannelId

        );
SIGNAL  MCS.Channel.Leave.request
        (
                MCSAttachmentId,
                ChannelId

        );
```

```
SIGNAL  MCS.Channel.Leave.indication
        (
                MCSAttachmentId,
                ChannelId,
                Reason

        );
SIGNAL  MCS.Channel.Convene.request
        (
                MCSAttachmentId
        );
SIGNAL  MCS.Channel.Convene.confirm
        (
                MCSAttachmentId,
                Result,
                ChannelId

        );
SIGNAL  MCS.Channel.Disband.request
        (
                MCSAttachmentId,
                ChannelId

        );
SIGNAL  MCS.Channel.Disband.indication
        (
                MCSAttachmentId,
                ChannelId,
                Reason

        );
SIGNAL  MCS.Channel.Admit.request
        (
                MCSAttachmentId,
                ChannelId,
                UserIdSet

        );
SIGNAL  MCS.Channel.Admit.indication
        (
                MCSAttachmentId,
                ChannelId,
                UserId

        );
SIGNAL  MCS.Channel.Expel.request
        (
                MCSAttachmentId,
                ChannelId,
                UserIdSet

        );
SIGNAL  MCS.Channel.Expel.indication
        (
                MCSAttachmentId,
                ChannelId,
                Reason

        );
```

```
SIGNAL  MCS.Send.Data.request
        (
                MCSAttachmentId,
                ChannelId,
                DataPriority,
                Segmentation,
                UserData

        );
SIGNAL  MCS.Send.Data.indication
        (
                MCSAttachmentId,
                ChannelId,
                DataPriority,
                UserId,
                Segmentation,
                UserData

        );
SIGNAL  MCS.Uniform.Send.Data.request
        (
                MCSAttachmentId,
                ChannelId,
                DataPriority,
                Segmentation,
                UserData

        );
SIGNAL  MCS.Uniform.Send.Data.indication
        (
                MCSAttachmentId,
                ChannelId,
                DataPriority,
                UserId,
                Segmentation,
                UserData

        );
SIGNAL  MCS.Token.Grab.request
        (
                MCSAttachmentId,
                TokenId

        );
SIGNAL  MCS.Token.Grab.confirm
        (
                MCSAttachmentId,
                TokenId,
                Result

        );
SIGNAL  MCS.Token.Inhibit.request
        (
                MCSAttachmentId,
                TokenId

        );
SIGNAL  MCS.Token.Inhibit.confirm
        (
                MCSAttachmentId,
                TokenId,
                Result

        );
```

```
SIGNAL  MCS.Token.Give.request
        (
                MCSAttachmentId,
                TokenId,
                UserId

        );
SIGNAL  MCS.Token.Give.indication
        (
                MCSAttachmentId,
                TokenId,
                UserId

        );
SIGNAL  MCS.Token.Give.response

        (
                MCSAttachmentId,
                TokenId,
                Result

        );
SIGNAL  MCS.Token.Give.confirm
        (
                MCSAttachmentId,
                TokenId,
                Result

        );
SIGNAL  MCS.Token.Please.request
        (
                MCSAttachmentId,
                TokenId

        );
SIGNAL  MCS.Token.Please.indication
        (
                MCSAttachmentId,
                TokenId,
                UserId

        );
SIGNAL  MCS.Token.Release.request
        (
                MCSAttachmentId,
                TokenId

        );
SIGNAL  MCS.Token.Release.confirm
        (
                MCSAttachmentId,
                TokenId,
                Result

        );
SIGNAL  MCS.Token.Test.request
        (
                MCSAttachmentId,
                TokenId

        );
```

```
SIGNAL  MCS.Token.Test.confirm
        (
                MCSAttachmentId,
                TokenId,
                TokenStatus

        );

CHANNEL TSAPs
        FROM MCS.provider TO ENV WITH
                (Endpoint.to.Transport);
        FROM ENV TO MCS.provider WITH
                T.Connect.indication,
                (Transport.to.Endpoint);
ENDCHANNEL;

SIGNALLIST Endpoint.to.Transport  =
        T.ready,
        T.Connect.request,
        T.Connect.response,
        T.Data.request,
        T.Disconnect.request;

SIGNALLIST Transport.to.Endpoint  =
        T.ready,
        T.Connect.confirm,
        T.Data.indication,
        T.Disconnect.indication;

SIGNAL  T.ready                         /* allows one T.Data */
        (
                TCEndpointId

        );

SIGNAL  T.Connect.request
        (
                Natural,                /* requester's label */
                TSAPAddress,            /* calling */
                TSAPAddress,            /* called */
                TransportQOS,                   /* target */
                TransportQOS            /* minimum */

        );

SIGNAL  T.Connect.indication
        (
                TCEndpointId,           /* provider-assigned */
                TSAPAddress,            /* calling */
                TSAPAddress,            /* called */
                TransportQOS,                   /* offered */
                TransportQOS            /* minimum */

        );

SIGNAL  T.Connect.response
        (
                TCEndpointId,
                TransportQOS            /* selected */

        );

SIGNAL  T.Connect.confirm
        (
                Natural,                /* requester's label */
                TCEndpointId,           /* provider-assigned */
                TransportQOS            /* selected */

        );
```

```
SIGNAL  T.Data.request
        (
                TCEndpointId,
                TSDU

        );

SIGNAL  T.Data.indication
        (
                TCEndpointId,
                TSDU

        );

SIGNAL  T.Disconnect.request
        (
                TCEndpointId

        );

SIGNAL  T.Disconnect.indication
        (
                Natural,                        /* requester's label */
                TCEndpointId                    /* provider-assigned */

        );

ENDSYSTEM;

BLOCK   MCS.provider;

SYNONYM maxPortalIds            Natural  =  EXTERNAL;              /* an implementation limit */

        /* Data type definitions */

NEWTYPE         PDUKind                 /* domain MCSPDUs */
LITERALS
        PDin,           /* plumb domain indication */
        EDrq,           /* erect domain request */
        MCrq,           /* merge channels request */
        MCcf,           /* merge channels confirm */
        PCin,           /* purge channels indication */
        MTrq,           /* merge tokens request */
        MTcf,           /* merge tokens confirm */
        PTin,           /* purge tokens indication */
        DPum,           /* disconnect provider ultimatum */
        RJum,           /* reject MCSPDU ultimatum */
        AUrq,           /* attach user request */
        AUcf,           /* attach user confirm */
        DUrq,           /* detach user request */
        DUin,           /* detach user confirm */
        CJrq,           /* channel join request */
        CJcf,           /* channel join confirm */
        CLrq,           /* channel leave request */
        CCrq,           /* channel convene request */
        CCcf,           /* channel convene confirm */
        CDrq,           /* channel disband request */
        CDin,           /* channel disband confirm */
        CArq,           /* channel admit request */
        CAin,           /* channel admit confirm */
        CErq,           /* channel expel request */
        CEin,           /* channel expel confirm */
        SDrq,           /* send data request */
        SDin,           /* send data indication */
        USrq,           /* uniform send data request */
        USin,           /* uniform send data indication */
        TGrq,           /* token grab request */
        TGcf,           /* token grab confirm */
        TIrq,           /* token inhibit request */
```

```
        Tlcf,           /* token inhibit confirm */
        TVrq,           /* token give request */
        TVin,           /* token give indication */
        TVrs,           /* token give response */
        TVcf,           /* token give confirm */
        TPrq,           /* token please request */
        TPin,           /* token please indication */
        TRrq,           /* token release request */
        TRcf,           /* token release confirm */
        TTrq,           /* token test request */
        TTcf;           /* token test confirm */

ENDNEWTYPE;

NEWTYPE         PDUStruct
STRUCT
        kind                    PDUKind;
        /* fields used depend on kind */
        channelId               ChannelId;
        channelIds              ChannelIdSet;
        dataPriority            DataPriority;
        detachUserIds           UserIdSet;
        diagnostic              Diagnostic;
        heightLimit             Natural;
        initialOctets           OctetString;
        initiator               UserId;
        mergeChannels           ChannelAttributesSet;
        mergeTokens             TokenAttributesSet;
        purgeChannelIds         ChannelIdSet;
        purgeTokenIds           TokenIdSet;
        reason                  Reason;
        recipient               UserId;
        requested               ChannelId;
        result                  Result;
        segmentation            Segmentation;
        subHeight               Natural;
        subInterval             Duration;
        tokenId                 TokenId;
        tokenStatus             TokenStatus;
        userData                UserData;
        userIds                 UserIdSet;
ENDNEWTYPE;

NEWTYPE         ChannelKind
LITERALS
        Static,         /* range 1:1000  =  static: known permanently */
        UserId,         /* dynamic: Attach-User / Detach-User */
        Private,        /* dynamic: Channel-Convene / Channel-Disband */
        Assigned;       /* dynamic: Channel-Join zero / last Channel-Leave */
ENDNEWTYPE;

NEWTYPE         ChannelAttributes
STRUCT
        channelId       ChannelId;      /* the channel with these attributes */
        kind            ChannelKind;    /* (Static,UserId,Private,Assigned) */
        manager         UserId;         /* if (Private): the channel manager */
        admitted        UserIdSet;      /* if (Private): zero or more users */
        joined          Boolean;        /* if (UserId,Private): True if joined */
ENDNEWTYPE;

NEWTYPE         ChannelAttributesSet   SetOf(ChannelAttributes);
ENDNEWTYPE;
```

```
NEWTYPE         TokenKind
LITERALS
        Grabbed,        /* assigned exclusively to one user */
        Inhibited,      /* inhibited by one or more users */
        Giving,         /* reassigning grabbed to a new user */
        Ungivable,      /* the recipient has since detached */
        Given;          /* donor released token or detached */
ENDNEWTYPE;

NEWTYPE         TokenAttributes
STRUCT
        tokenId         TokenId;        /* the token with these attributes */
        kind            TokenKind;      /* (Grabbed,Inhibited,Giving,Ungivable,Given) */
        grabber         UserId;         /* if (Grabbed,Giving,Ungivable): user */
        recipient       UserId;         /* if (Giving,Given): an intended user */
        inhibitors      UserIdSet;      /* if (Inhibited): one or more users */
ENDNEWTYPE;

NEWTYPE         TokenAttributesSet      SetOf(TokenAttributes);
ENDNEWTYPE;

SYNTYPE         PortalId        =  Integer CONSTANTS 0:maxPortalIds
ENDSYNTYPE;

NEWTYPE         PortalIdSet     SetOf(PortalId);
ENDNEWTYPE;

NEWTYPE         PortalKind
LITERALS
        Attached,       /* MCS Attachment through an MCSAP */
        Downlink,       /* MCS Connection to a provider below */
        Uplink;         /* MCS Connection to a provider above */
ENDNEWTYPE;

NEWTYPE         PIdByPri        Array(DataPriority, PId);
ENDNEWTYPE;

NEWTYPE         Diagnostic
LITERALS
        DC_inconsistent_merge,
        DC_forbidden_PDU_downward,
        DC_forbidden_PDU_upward,
        DC_invalid_BER_encoding,
        DC_invalid_PER_encoding,
        DC_misrouted_user,
        DC_unrequested_confirm,
        DC_wrong_transport_priority,
        DC_channel_id_conflict,
        DC_token_id_conflict,
        DC_not_user_id_channel,
        DC_too_many_channels,
        DC_too_many_tokens,
        DC_too_many_users,
        DC_OK,                                  /* not in MCSPDUs */
        DC_ignore,                              /* not in MCSPDUs */
        DC_height_limit_exceeded,               /* not in MCSPDUs */
        DC_throughput_inadequate;               /* not in MCSPDUs */
ENDNEWTYPE;

        /* Process decomposition */
PROCESS  Contro         (1,1)   REFERENCED;  /* CO */
PROCESS  Attachment     (0,)    REFERENCED;  /* AT */
PROCESS  Domain         (0,)    REFERENCED;  /* DM */
PROCESS  Endpoint       (0,)    REFERENCED;   /* EP */
```

```
CONNECT Control.MCSAP AND Control.MCSAP.CO;

SIGNALROUTE Control.MCSAP.CO
        FROM ENV TO Control WITH
            (Controller.to.Control);
        FROM Control TO ENV WITH
            (Control.to.Controller);

CONNECT MCSAPs AND MCSAPs.CO, MCSAPs.AT;

SIGNALROUTE MCSAPs.CO
        FROM ENV TO Control WITH
            MCS.Attach.User.request;
        FROM Control TO ENV WITH
            MCS.Attach.User.confirm;

SIGNALROUTE MCSAPs.AT
        FROM ENV TO Attachment WITH
            (Users.to.Attachment);
        FROM Attachment TO ENV WITH
            (Attachment.to.Users);

CONNECT TSAPs AND TSAPs.CO, TSAPs.EP;

SIGNALROUTE TSAPs.CO
        FROM ENV TO Control WITH
            T.Connect.indication;
        FROM Control TO ENV WITH
            T.Disconnect.request;

SIGNALROUTE TSAPs.EP
        FROM ENV TO Endpoint WITH
            (Transport.to.Endpoint);
        FROM Endpoint TO ENV WITH
            (Endpoint.to.Transport);

SIGNALROUTE CO.AT
        FROM Control TO Attachment WITH
            Quit,
            Exit;
        FROM Attachment TO Control WITH
            Quit;

SIGNALROUTE CO.DM
        FROM Control TO Domain WITH
                Open.portal,
                Drop.portal,
                Shut.portal;
        FROM Domain TO Control WITH
                Drop.portal,
                Report.portal,
                Shut.portal;

SIGNALROUTE CO.EP
        FROM Control TO Endpoint WITH
                (Connect.PDU),
                Quit,
                Exit;
        FROM Endpoint TO Control WITH
                (Connect.PDU),
                RJum,
                Quit;

SIGNALROUTE DM.AT
        FROM Domain TO Attachment WITH
                PDU.ready,
                (PDU.indication),
                (PDU.confirm);
        FROM Attachment TO Domain WITH
                PDU.ready,
                (PDU.request),
                (PDU.response);
```

```
SIGNALROUTE DM.EP
        FROM Domain TO Endpoint WITH
                PDU.ready,
                (PDU.merge),
                (PDU.request),
                (PDU.indication),
                (PDU.response),
                (PDU.confirm),
                (PDU.ultimatum);
        FROM Endpoint TO Domain WITH
                PDU.ready,
                (PDU.merge),
                (PDU.request),
                (PDU.indication),
                (PDU.response),
                (PDU.confirm),
                (PDU.ultimatum);

SIGNAL  Open.portal
        (
                PortalId,
                PortalKind,
                PIdByPri
        );

SIGNAL  Report.portal
        (
                PortalId,
                Diagnostic
        );

SIGNAL  Drop.portal
        (
                PortalId,
                Reason
        );

SIGNAL  Shut.portal
        (
                PortalId
        );

SIGNAL  Quit;

SIGNAL  Exit;

SIGNALLIST Connect.PDU  =
            Connect.Initial,
            Connect.Response,
            Connect.Additional,
            Connect.Result;

SIGNAL  Connect.Initial
        (
                DomainSelector,             /* calling */
                DomainSelector,             /* called */
                Boolean,                     /* upward */
                DomainParameters,            /* target */
                DomainParameters,            /* minimum */
                DomainParameters,            /* maximum */
                UserData
        );

SIGNAL  Connect.Response
        (
                Result,
                Natural,
                DomainParameters,
                UserData
        );
```

```
SIGNAL  Connect.Additional
        (
                Natural,
                DataPriority
        );

SIGNAL  Connect.Result
        (
                Result
        );

SIGNALLIST PDU.merge =
        PDin, EDrq, MCrq, MCcf, MTrq,
        MTcf;

SIGNALLIST PDU.request =
        AUrq, DUrq, CJrq, CLrq, CCrq,
        CDrq, CArq, CErq, SDrq, USrq,
        TGrq, TIrq, TVrq, TPrq, TRrq,
        TTrq;

SIGNALLIST PDU.indication  =
        PCin, PTin, DUin, CDin, CAin,
        CEin, SDin, USin, TVin, TPin;

SIGNALLIST PDU.response =
        TVrs;

SIGNALLIST PDU.confirm  =
        AUcf, CJcf, CCcf, TGcf, TIcf,
        TVcf, TRcf, TTcf;

SIGNALLIST PDU.ultimatum  =
        DPum, RJum;

SIGNAL  PDU.ready                       /* allows one domain MCSPDU */
        (
                DataPriority
        );

SIGNAL  PDin    (PDUStruct);    /* plumb domain indication */
SIGNAL  EDrq    (PDUStruct);    /* erect domain request */
SIGNAL  MCrq    (PDUStruct);    /* merge channels request */
SIGNAL  MCcf    (PDUStruct);    /* merge channels confirm */
SIGNAL  PCin    (PDUStruct);    /* purge channels indication */
SIGNAL  MTrq    (PDUStruct);    /* merge tokens request */
SIGNAL  MTcf    (PDUStruct);    /* merge tokens confirm */
SIGNAL  PTin    (PDUStruct);    /* purge tokens indication */
SIGNAL  DPum    (PDUStruct);    /* disconnect provider ultimatum */
SIGNAL  RJum    (PDUStruct);    /* reject MCSPDU ultimatum */
SIGNAL  AUrq    (PDUStruct);    /* attach user request */
SIGNAL  AUcf    (PDUStruct);    /* attach user confirm */
SIGNAL  DUrq    (PDUStruct);    /* detach user request */
SIGNAL  DUin    (PDUStruct);    /* detach user confirm */
SIGNAL  CJrq    (PDUStruct);    /* channel join request */
SIGNAL  CJcf    (PDUStruct);    /* channel join confirm */
SIGNAL  CLrq    (PDUStruct);    /* channel leave request */
SIGNAL  CCrq    (PDUStruct);    /* channel convene request */
SIGNAL  CCcf    (PDUStruct);    /* channel convene confirm */
SIGNAL  CDrq    (PDUStruct);    /* channel disband request */
```

```
SIGNAL  CDin     (PDUStruct);     /* channel disband confirm */
SIGNAL  CArq     (PDUStruct);     /* channel admit request */
SIGNAL  CAin     (PDUStruct);     /* channel admit confirm */
SIGNAL  CErq     (PDUStruct);     /* channel expel request */
SIGNAL  CEin     (PDUStruct);     /* channel expel confirm */
SIGNAL  SDrq     (PDUStruct);     /* send data request */
SIGNAL  SDin     (PDUStruct);     /* send data indication */
SIGNAL  USrq     (PDUStruct);     /* uniform send data request */
SIGNAL  USin     (PDUStruct);     /* uniform send data indication */
SIGNAL  TGrq     (PDUStruct);     /* token grab request */
SIGNAL  TGcf     (PDUStruct);     /* token grab confirm */
SIGNAL  TIrq     (PDUStruct);     /* token inhibit request */
SIGNAL  TIcf     (PDUStruct);     /* token inhibit confirm */
SIGNAL  TVrq     (PDUStruct);     /* token give request */
SIGNAL  TVin     (PDUStruct);     /* token give indication */
SIGNAL  TVrs     (PDUStruct);     /* token give response */
SIGNAL  TVcf     (PDUStruct);     /* token give confirm */
SIGNAL  TPrq     (PDUStruct);     /* token please request */
SIGNAL  TPin     (PDUStruct);     /* token please indication */
SIGNAL  TRrq     (PDUStruct);     /* token release request */
SIGNAL  TRcf     (PDUStruct);     /* token release confirm */
SIGNAL  TTrq     (PDUStruct);     /* token test request */
SIGNAL  TTcf     (PDUStruct);     /* token test confirm */

ENDBLOCK;
```

Apéndice III

Especificación en SDL del proceso de control

(Este apéndice no es parte integrante de la presente Recomendación)

```
PROCESS     Control;

        /* Type definitions */

NEWTYPE         Proc
LITERALS
        Nil,
        Receiving,                      /*                  Endpoint */
        Responding,                     /*                  Endpoint */
        Engaged,                                /* Attachment Endpoint */
        Quitting,                       /* Attachment Endpoint */
        Quit;                           /* Attachment Endpoint */
ENDNEWTYPE;

NEWTYPE         ProcByPri       Array(DataPriority, Proc);
ENDNEWTYPE;

NEWTYPE         CallSide
LITERALS
        Calling,
        Called;
ENDNEWTYPE;
```

```
NEWTYPE         PortalStruct
STRUCT
        mcId            MCSConnectionId;                    /* equals PortalId index */
        ccId            Natural;                    /* equals PortalId index */
        kind            PortalKind;                 /* (Attached,Downlink,Uplink) */
        pids            PIdByPri;                   /* processes comprising a portal */
        proc            ProcByPri;                  /* state of each created process */
        label           Natural;                    /* requester's label for confirm */
        domain          DomainSelector;             /* domain selected by the portal */
        opened          Boolean;                    /* True if portal has been opened */
        notify          Boolean;                    /* True to notify when portal quit */
        minParms        DomainParameters;           /* lower limit for negotiation */
        maxParms        DomainParameters;           /* upper limit for negotiation */
        parameters      DomainParameters;           /* values negotiated by portal */
        callSide        CallSide;                   /* portal is calling or called */
        localTSAP       TSAPAddress;                /* local address for T.Connect */
        remoteTSAP      TSAPAddress;                /* remote address for T.Connect */
        targetQOSByPri  TransportQOSByPri;          /* desired quality of service */
        minQOSByPri     TransportQOSByPri;                  /* minimum that is acceptable */
        userData        UserData;                   /* of response, pending confirm */
ENDNEWTYPE;
        /* Note:  In a practical implementation, the user data stored from
        Connect.Response, awaiting establishment of additional TCs, need be
        only one transport interface data unit, not a complete TSDU.  Any
        excess can be left in the pipeline of the initial TC to be read out
        when MCS.Connect.Provider.confirm is issued. */

NEWTYPE         Portal          Array(PortalId, PortalStruct);
ENDNEWTYPE;

NEWTYPE         DomainStruct
STRUCT
        pid             PId;                        /* process for the domain or Null */
        portals         Natural;                    /* number of portals open to domain */
        upward          portalId;                   /* unique connection upward or zero */
        minParms        DomainParameters;           /* lower limit of configuration */
        maxParms        DomainParameters;           /* upper limit of configuration */
        parameters      DomainParameters;           /* values established in domain */
ENDNEWTYPE;

NEWTYPE         Domain          Array(DomainSelector, DomainStruct);
ENDNEWTYPE;

NEWTYPE         DomainSelectorSet           SetOf(DomainSelector);
ENDNEWTYPE;

        /* Data declarations */

DCL     domain          Domain,                     /* resource arrays */
        portal          Portal;

        /* Note: The fields of a domain or portal array element
        are undefined if the corresponding index is not in dUsed
        or pUsed respectively. */
DCL     dUsed           DomainSelectorSet,          /* indexes used */
        pUsed           PortalIdSet;

DCL     pFree           PortalIdSet;                /* indexes free */

DCL     nullParms       DomainParameters;           /* initializer */

        /* Procedure decomposition */

        /*              Initialize_resources
                        Identify_sender             (p, dp)
                        Min_parms                   (min, a, b)
```

```
                Max_parms                           (max, a, b)
                Test_parms                          (result, x, min, max)
                MCS_Connect_Provider_request        (...)
                T_Connect_indication                        (...)
                Connect_Initial                     (...)
                MCS_Connect_Provider_response       (...)
                Connect_Response                    (...)
                Connect_Additional                  (...)
                Connect_Result                      (...)
                MCS_Disconnect_Provider_request     (...)
                MCS_Attach_User_request             (...)
                Open_portal                         (p)
                Drop_portal                         (p, reason)
                Report_portal                       (p, diagnostic)
                RJum                                (pdu)
                Quit_portal                         (p, reason)
                Quit
                Shut_portal                         (p)
                Exit_portal                         (p)       */

                                                                        /*--------------------------------*/
PROCEDURE       Initialize_resources;                                   /*  Initialize_resources  */
                                                                        /*--------------------------------*/
        DCL     p               PortalId,
                ds              DomainSelector,
                dSet            DomainSelectorSet;
        START
        COMMENT 'Initialize data structures during process start-up
                before accepting the first input signal.
                Note that each SetOf automatically defaults to Empty.
                Configure one hypothetical domain as an example.
                Some limits are determined by the implementation.
                ';
        TASK    nullParms!maxChannelIds := 0,
                nullParms!maxUserIds := 0,
                nullParms!maxTokenIds := 0,
                nullParms!numPriorities := 0,
                nullParms!minThroughput := 0,
                nullParms!maxHeight := 0,
                nullParms!maxMCSPDUsize := 0,
                nullParms!protocolVersion := 0,
                p := maxPortalIds;
         1b :   /* for p = ?..1 */
        DECISION p > 0;
         (True): TASK   portal(p)!mcId := p,
                        portal(p)!ccId := p,
                        pFree := Incl(p, pFree),
                        p := p - 1;
                JOIN 1b;
        ELSE:ENDDECISION;
        TASK    ds := Mkstring(77) // Mkstring(67) // Mkstring(83),
                dUsed := Incl(ds, dUsed),
                domain(ds)!Pid := Null,
                domain(ds)!portals := 0,
                domain(ds)!upward := 0,
                dSet := dUsed;
         2b :   /* for ds in dSet */
        DECISION dSet = Empty;
         (False): TASK  ds := Pick(dSet),
                        dSet := Del(ds, dSet),
                        domain(ds)!minParms!maxChannelIds := 0,
                        domain(ds)!minParms!maxUserIds := 0,
                        domain(ds)!minParms!maxTokenIds := 0,
                        domain(ds)!minParms!numPriorities := 1,
                        domain(ds)!minParms!minThroughput := 0,
                        domain(ds)!minParms!maxHeight := 1,
                        domain(ds)!minParms!maxMCSPDUsize := 35,
```

```
                        domain(ds)!minParms!protocolVersion := 1,
                        domain(ds)!maxParms!maxChannelIds := 65535,
                        domain(ds)!maxParms!maxUserIds := 65535,
                        domain(ds)!maxParms!maxTokenIds := 65535,
                        domain(ds)!maxParms!numPriorities := 4,
                        domain(ds)!maxParms!minThroughput := 1000000,
                        domain(ds)!maxParms!maxHeight := 1000,
                        domain(ds)!maxParms!maxMCSPDUsize := 32768,
                        domain(ds)!maxParms!protocolVersion := 2;
                JOIN 2b;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                                        /*-------------------------*/
PROCEDURE       Identify_sender;                                        /*  Identify_sender  */
FPAR    IN/OUT  p               PortalId,                               /*-------------------------*/
        IN/OUT  dp              DataPriority;

        DCL     pSet            PortalIdSet;
        START
        COMMENT 'An alternative would be to carry this
                information explicitly in SDL signals.
                ';
        TASK    pSet := pUsed;
         1b :   /* for p in pSet */
        DECISION pSet = Empty;
         (False):       TASK    p := Pick(pSet),
                                pSet := Del(p, pSet),
                                dp := 0;
                         2b :   /* for dp = 0..3 */
                        DECISION dp < 4;
                         (True):    DECISION portal(p)!pids(dp) = SENDER;
                                     (True): RETURN;
                                    ELSE:ENDDECISION;
                                    TASK  dp := dp + 1;
                                    JOIN 2b;
                        ELSE:ENDDECISION;
                        JOIN 1b;
        ELSE:ENDDECISION;
        TASK    p := 0,
                dp := 0;
        RETURN;
        ENDPROCEDURE;

                                                                        /*--------------------*/
PROCEDURE       Min_parms;                                              /*  Min_parms  */
FPAR    IN/OUT  min             DomainParameters,                       /*--------------------*/
                a               DomainParameters,
                b               DomainParameters;

        START
        COMMENT 'Return the minimum of two parameter sets.
                ';
        TASK    min!maxChannelIds := IF a!maxChannelIds < b!maxChannelIds
                                THEN a!maxChannelIds ELSE b!maxChannelIds FI,
                min!maxUserIds := IF a!maxUserIds < b!maxUserIds
                                THEN a!maxUserIds ELSE b!maxUserIds FI,
                min!maxTokenIds := IF a!maxTokenIds < b!maxTokenIds
                                THEN a!maxTokenIds ELSE b!maxTokenIds FI,
                min!numPriorities := IF a!numPriorities < b!numPriorities
                                THEN a!numPriorities ELSE b!numPriorities FI,
                min!minThroughput := IF a!minThroughput < b!minThroughput
                                THEN a!minThroughput ELSE b!minThroughput FI,
                min!maxHeight := IF a!maxHeight < b!maxHeight
                                THEN a!maxHeight ELSE b!maxHeight FI,
```

```
                min!maxMCSPDUsize := IF a!maxMCSPDUsize < b!maxMCSPDUsize
                                THEN a!maxMCSPDUsize ELSE b!maxMCSPDUsize FI,
                min!protocolVersion := IF a!protocolVersion < b!protocolVersion
                                THEN a!protocolVersion ELSE b!protocolVersion FI;
        RETURN;
        ENDPROCEDURE;

                                                                        /*--------------------*/
PROCEDURE       Max_parms;                                              /*  Max_parms  */
FPAR    IN/OUT  max             DomainParameters,                       /*--------------------*/
                a               DomainParameters,
                b               DomainParameters;

        START
        COMMENT 'Return the maximum of two parameter sets.
                ';
        TASK    max!maxChannelIds := IF a!maxChannelIds > b!maxChannelIds
                                THEN a!maxChannelIds ELSE b!maxChannelIds FI,
                max!maxUserIds := IF a!maxUserIds > b!maxUserIds
                                THEN a!maxUserIds ELSE b!maxUserIds FI,
                max!maxTokenIds := IF a!maxTokenIds > b!maxTokenIds
                                THEN a!maxTokenIds ELSE b!maxTokenIds FI,
                max!numPriorities := IF a!numPriorities > b!numPriorities
                                THEN a!numPriorities ELSE b!numPriorities FI,
                max!minThroughput := IF a!minThroughput > b!minThroughput
                                THEN a!minThroughput ELSE b!minThroughput FI,
                max!maxHeight := IF a!maxHeight > b!maxHeight
                                THEN a!maxHeight ELSE b!maxHeight FI,
                max!maxMCSPDUsize := IF a!maxMCSPDUsize > b!maxMCSPDUsize
                                THEN a!maxMCSPDUsize ELSE b!maxMCSPDUsize FI,
                max!protocolVersion := IF a!protocolVersion > b!protocolVersion
                                THEN a!protocolVersion ELSE b!protocolVersion FI;
        RETURN;
        ENDPROCEDURE;

                                                                        /*--------------------*/
PROCEDURE       Test_parms;                                             /*  Test_parms  */
FPAR    IN/OUT  result          Result,                                 /*--------------------*/
                z               DomainParameters,
                min             DomainParameters,
                max             DomainParameters;

        START
        COMMENT 'Check that the parameters lie between min and max.
                ';
        DECISION (z!maxChannelIds >= min!maxChannelIds)
              and (z!maxChannelIds <= max!maxChannelIds)
              and (z!maxUserIds >= min!maxUserIds)
              and (z!maxUserIds <= max!maxUserIds)
              and (z!maxTokenIds >= min!maxTokenIds)
              and (z!maxTokenIds <= max!maxTokenIds)
              and (z!numPriorities >= min!numPriorities)
              and (z!numPriorities <= max!numPriorities)
              and (z!minThroughput >= min!minThroughput)
              and (z!minThroughput <= max!minThroughput)
              and (z!maxHeight >= min!maxHeight)
              and (z!maxHeight <= max!maxHeight)
              and (z!maxMCSPDUsize >= min!maxMCSPDUsize)
              and (z!maxMCSPDUsize <= max!maxMCSPDUsize)
              and (z!protocolVersion >= min!protocolVersion)
              and (z!protocolVersion <= max!protocolVersion);
         (True):  TASK  result := RT_successful;
         (False): TASK  result := RT_parameters_unacceptable;
        ENDDECISION;
        RETURN;
        ENDPROCEDURE;
```

```
                                                        /*-------------------------------------------------------*/
PROCEDURE       MCS_Connect_Provider_request;           /*  MCS_Connect_Provider_request  */
FPAR            label               Natural,            /*-------------------------------------------------------*/
                localTSAP           TSAPAddress,
                localDomain         DomainSelector,
                remoteTSAP          TSAPAddress,
                remoteDomain        DomainSelector,
                upward              Boolean,
                targetParms         DomainParameters,
                minParms            DomainParameters,
                maxParms            DomainParameters,
                targetQOSByPri      TransportQOSByPri,
                minQOSByPri         TransportQOSByPri,
                userData            UserData;


        DCL     result              Result,
                p                   PortalId,
                dp                  DataPriority,
                ds                  DomainSelector;
        START
        COMMENT 'Process an MCS.Connect.Provider.request input signal.
                Begin parameter negotiation and allocate a portal.
                Create an endpoint process for the initial TC and
                transmit Connect.Initial through it.
                ';
        DECISION pFree = Empty;
         (True):        TASK    result := RT_congested;
                        JOIN 1f;
        ELSE:ENDDECISION;
        TASK            ds := localDomain;
        DECISION ds in dUsed;
         (False):       TASK    result := RT_no_such_domain;
                        JOIN 1f;
        ELSE:ENDDECISION;
        DECISION upward and domain(ds)!upward /= 0;
         (True):        TASK    result := RT_domain_not_hierarchical;
                        JOIN 1f;
        ELSE:ENDDECISION;
        CALL            Max_parms(minParms, minParms, domain(ds)!minParms);
        CALL            Min_parms(maxParms, maxParms, domain(ds)!maxParms);
        DECISION domain(ds)!portals > 0;
         (True):        TASK    targetParms := domain(ds)!parameters;
         (False):       CALL    Max_parms(targetParms, targetParms, minParms);
                        CALL    Min_parms(targetParms, targetParms, maxParms);
        ENDDECISION;
        CALL            Test_parms(result, targetParms, minParms, maxParms);
        DECISION result = RT_successful;
         (False): 1f :
                        OUTPUT  MCS.Connect.Provider.confirm
                                (label, result, 0, nullParms, NullString);
                        RETURN;
        ELSE:ENDDECISION;
        DECISION domain(ds)!portals > 0;
         (True):        TASK    minParms := targetParms,
                                maxParms := targetParms;
        ELSE:ENDDECISION;
        CREATE          Endpoint(Null, localTSAP, remoteTSAP,
                        targetQOSByPri(0), minQOSByPri(0),
                        nullParms);
        OUTPUT          Connect.Initial(localDomain, remoteDomain, upward,
                        targetParms, minParms, maxParms, userData)
                        TO OFFSPRING;
        TASK            p := Pick(pFree),
                        pFree := Del(p, pFree),
                        pUsed := Incl(p, pUsed),
                        portal(p)!kind := IF upward THEN Uplink ELSE Downlink FI,
```

```
                portal(p)!label := label,
                portal(p)!domain := ds,
                portal(p)!opened := False,
                portal(p)!notify := True,
                portal(p)!minParms := minParms,
                portal(p)!maxParms := maxParms,
                portal(p)!parameters := nullParms,
                portal(p)!callSide := Calling,
                portal(p)!localTSAP := localTSAP,
                portal(p)!remoteTSAP := remoteTSAP,
                portal(p)!targetQOSByPri := targetQOSByPri,
                portal(p)!minQOSByPri := minQOSByPri,
                portal(p)!pids(0) := OFFSPRING,
                portal(p)!proc(0) := Receiving,
                dp := 1;
 2b :           /* for dp = 1..3 */
DECISION dp < 4;
 (True):        TASK    portal(p)!pids(dp) := Null,
                        portal(p)!proc(dp) := Nil,
                        dp := dp + 1;
                JOIN 2b;
ELSE:ENDDECISION;
RETURN;
ENDPROCEDURE;


                                                        /*-----------------------------------*/
PROCEDURE       T_Connect_indication;                   /*  T_Connect_indication  */
FPAR            tcId            TCEndpointId,           /*-----------------------------------*/
                remoteTSAP      TSAPAddress,
                localTSAP       TSAPAddress,
                offeredQOS      TransportQOS,
                minQOS          TransportQOS;


DCL     p               PortalId,
        dp              DataPriority;
START
COMMENT 'Process a T.Connect.indication input signal.
        Allocate a portal, in case this is an initial TC.
        Create an endpoint process to receive Connect.Initial
        or Connect.Additional.
        ';
DECISION pFree = Empty
        or offeredQOS!throughput < minQOS!throughput
        or offeredQOS!transitDelay > minQOS!transitDelay
        or offeredQOS!dataPriority > minQOS!dataPriority;
 (True):        OUTPUT  T.Disconnect.request(tcId);
                RETURN;
ELSE:ENDDECISION;
CREATE  Endpoint(tcId, localTSAP, remoteTSAP,
        offeredQOS, minQOS,
        nullParms);
TASK    p := Pick(pFree),
        pFree := Del(p, pFree),
        pUsed := Incl(p, pUsed),
        portal(p)!kind := Downlink,
        portal(p)!opened := False,
        portal(p)!notify := False,
        portal(p)!parameters := nullParms,
        portal(p)!callSide := Called,
        portal(p)!localTSAP := localTSAP,
        portal(p)!remoteTSAP := remoteTSAP,
        portal(p)!pids(0) := OFFSPRING,
        portal(p)!proc(0) := Receiving,
        dp := 1;
 1b :   /* for dp = 1..3 */
```

```
        DECISION dp < 4;
         (True):    TASK    portal(p)!pids(dp) := Null,
                            portal(p)!proc(dp) := Nil,
                            dp := dp + 1;
                    JOIN 1b;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;


                                                                    /*-------------------------*/
PROCEDURE           Connect_Initial;                                /*  Connect_Initial  */
FPAR                remoteDomain    DomainSelector,                 /*-------------------------*/
                    localDomain     DomainSelector,
                    upward          Boolean,
                    targetParms     DomainParameters,
                    minParms        DomainParameters,
                    maxParms        DomainParameters,
                    userData        UserData;


        DCL         result          Result,
                    p               PortalId,
                    dp              DataPriority,
                    ds              DomainSelector;
        START
        COMMENT     'Process a Connect.Initial input signal.
                    Retain the portal and begin parameter negotiation.
                    Indicate the connection to the controlling user.
                    ';
        CALL        Identify_sender(p, dp);
        DECISION p in pUsed and portal(p)!callSide = Called
                    and dp = 0 and portal(p)!proc(0) = Receiving;
         (False):   TASK    result := RT_unspecified_failure;
                    JOIN 1f;
        ELSE:ENDDECISION;
        TASK        ds := localDomain;
        DECISION ds in dUsed;
         (False):   TASK    result := RT_no_such_domain;
                    JOIN 1f;
        ELSE:ENDDECISION;
        DECISION upward or domain(ds)!upward = 0;
         (False):   TASK    result := RT_domain_not_hierarchical;
                    JOIN 1f;
        ELSE:ENDDECISION;
        CALL        Max_parms(minParms, minParms, domain(ds)!minParms);
        CALL        Min_parms(maxParms, maxParms, domain(ds)!maxParms);
        DECISION domain(ds)!portals > 0;
         (True):    TASK    targetParms := domain(ds)!parameters;
         (False):   CALL    Max_parms(targetParms, targetParms, minParms);
                    CALL    Min_parms(targetParms, targetParms, maxParms);
        ENDDECISION;
        CALL        Test_parms(result, targetParms, minParms, maxParms);
        DECISION result = RT_successful;
         (False): 1f :
                    OUTPUT  Connect.Response(result, 0, nullParms, NullString)
                            TO SENDER;
                    CALL    Quit_portal(p, RN_unspecified);
                    RETURN;
        ELSE:ENDDECISION;
        DECISION domain(ds)!portals > 0;
         (True):    TASK    minParms := targetParms,
                            maxParms := targetParms;
        ELSE:ENDDECISION;
        TASK        portal(p)!kind := IF upward THEN Downlink ELSE Uplink FI,
                    portal(p)!domain := ds,
                    portal(p)!notify := True,
                    portal(p)!minParms := minParms,
```

```
                portal(p)!maxParms := maxParms,
                portal(p)!proc(0) := Responding;
        OUTPUT  MCS.Connect.Provider.indication
                (portal(p)!mcId, portal(p)!localTSAP, localDomain, portal(p)!remoteTSAP,
                remoteDomain, upward, targetParms, minParms, maxParms, userData);
        RETURN;
        ENDPROCEDURE;


                                                        /*-------------------------------------------------------*/
PROCEDURE       MCS_Connect_Provider_response;          /*  MCS_Connect_Provider_response  */
FPAR            mcId            MCSConnectionId,        /*-------------------------------------------------------*/
                result          Result,
                parameters      DomainParameters,
                userData        UserData;


        DCL     p               PortalId;
        START
        COMMENT 'Process an MCS.Connect.Provider.response input signal.
                Check negotiated parameters and force a preference.
                Transmit Connect.Response.
                If there are no additional TCs, open the portal.
                ';
        TASK    p := mcId;
        DECISION p in pUsed and portal(p)!proc(0) = Responding;
         (False):       RETURN;
        ELSE:ENDDECISION;
        DECISION result = RT_successful;
         (False):       TASK    result := RT_user_rejected,
                                portal(p)!notify := False;
                        JOIN 1f;
        ELSE:ENDDECISION;
        CALL            Test_parms(result, parameters, portal(p)!minParms, portal(p)!maxParms);
        DECISION result = RT_successful;
         (False): 1f :
                        OUTPUT  Connect.Response(result, 0, parameters, userData)
                                TO portal(p)!pids(0);
                        CALL    Quit_portal(p, RN_parameters_unacceptable);
                        RETURN;
        ELSE:ENDDECISION;
        CALL            Max_parms(parameters, parameters, portal(p)!minParms);
        OUTPUT          Connect.Response(result, portal(p)!ccId, parameters, userData)
                        TO portal(p)!pids(0);
        TASK            portal(p)!parameters := parameters,
                        portal(p)!proc(0) := Engaged;
        CALL            Open_portal(p);
        RETURN;
        ENDPROCEDURE;


                                                        /*--------------------------------*/
PROCEDURE       Connect_Response;                       /*  Connect_Response  */
FPAR            result          Result,                 /*--------------------------------*/
                ccId            Natural,
                parameters      DomainParameters,
                userData        UserData;


        DCL     p               PortalId,
                dp              DataPriority;
        START
        COMMENT 'Process a Connect.Response input signal.
                Check the negotiated parameters.
                Create endpoint processes for additional TCs and
                transmit Connect.Additional through them.
                If there are no additional TCs, open the portal.
                ';
        CALL    Identify_sender(p, dp);
```

```
        DECISION p in pUsed and portal(p)!callSide = Calling
                        and dp = 0 and portal(p)!proc(0) = Receiving;
         (False):       CALL    Quit_portal(p, RN_unspecified);
                        RETURN;
        ELSE:ENDDECISION;
        TASK            portal(p)!parameters := parameters,
                        portal(p)!userData := userData;
        DECISION result = RT_successful;
         (False):       JOIN 1f;
        ELSE:ENDDECISION;
        CALL            Test_parms(result, parameters, portal(p)!minParms, portal(p)!maxParms);
        DECISION result = RT_successful;
         (False): 1f :
                        OUTPUT  MCS.Connect.Provider.confirm
                                (portal(p)!label, result, 0, parameters, userData);
                        TASK    portal(p)!notify := False;
                        CALL    Quit_portal(p, RN_unspecified);
                        RETURN;
        ELSE:ENDDECISION;
        TASK            portal(p)!proc(0) := Engaged,
                        dp := 1;
         2b :           /* for dp = 1..? */
        DECISION dp < parameters!numPriorities;
         (True):        CREATE  Endpoint(Null, portal(p)!localTSAP, portal(p)!remoteTSAP,
                                portal(p)!targetQOSByPri(dp), portal(p)!minQOSByPri(dp),
                                parameters);
                        OUTPUT  Connect.Additional(ccId, dp)
                                TO OFFSPRING;
                        TASK    portal(p)!pids(dp) := OFFSPRING,
                                portal(p)!proc(dp) := Receiving,
                                dp := dp + 1;
                        JOIN 2b;
        ELSE:ENDDECISION;
        CALL            Open_portal(p);
        RETURN;
        ENDPROCEDURE;

                                                                        /*--------------------------------*/
PROCEDURE               Connect_Additional;                             /*  Connect_Additional  */
FPAR                    ccId            Natural,                        /*--------------------------------*/
                        dp              DataPriority;

        DCL             p               PortalId,
                        r               PortalId,
                        x               DataPriority;
        START
        COMMENT         'Process a Connect.Additional input signal.
                        Release the allocated portal and piggyback onto
                        the preceding Connect.Initial.
                        Transmit Connect.Result.
                        If all TCs are established, open the portal.
                        ';
        CALL            Identify_sender(r, x);
        DECISION r in pUsed and portal(r)!callSide = Called
                        and x = 0 and portal(r)!proc(0) = Receiving;
         (False):       JOIN 1f;
        ELSE:ENDDECISION;
        TASK            p := ccId;
        DECISION p in pUsed and portal(p)!callSide = Called
                        and dp > 0 and dp < portal(p)!parameters!numPriorities
                        and portal(p)!proc(0) = Engaged and portal(p)!proc(dp) = Nil;
         (False): 1f :
                        OUTPUT  Connect.Result(RT_unspecified_failure)
                                TO SENDER;
                        CALL    Quit_portal(r, RN_unspecified);
                        RETURN;
        ELSE:ENDDECISION;
```

```
        TASK        pUsed := Del(r, pUsed),
                    pFree := Incl(r, pFree);
        OUTPUT      Connect.Result(RT_successful)
                    TO SENDER;
        TASK        portal(p)!pids(dp) := SENDER,
                    portal(p)!proc(dp) := Engaged;
        CALL        Open_portal(p);
        RETURN;
        ENDPROCEDURE;

                                                                    /*-------------------------*/
PROCEDURE           Connect_Result;                                       * Connect_Result */
FPAR                result          Result;                         /*-------------------------*/

        DCL         p               PortalId,
                    dp              DataPriority;
        START
        COMMENT     'Process a Connect.Result input signal.
                    If all TCs are established, open the portal.
                    ';
        CALL        Identify_sender(p, dp);
        DECISION p in pUsed and portal(p)!callSide = Calling
                    and dp > 0 and portal(p)!proc(dp) = Receiving;
         (False):   CALL    Quit_portal(p, RN_unspecified);
                    RETURN;
        ELSE:ENDDECISION;
        DECISION result = RT_successful;
         (False):   OUTPUT  MCS.Connect.Provider.confirm
                            (portal(p)!label, result, 0,
                            portal(p)!parameters, portal(p)!userData);
                    TASK    portal(p)!notify := False;
                    CALL    Quit_portal(p, RN_unspecified);
                    RETURN;
        ELSE:ENDDECISION;
        TASK        portal(p)!proc(dp) := Engaged;
        CALL        Open_portal(p);
        RETURN;
        ENDPROCEDURE;

                                                                    /*-------------------------------------------------------*/
PROCEDURE           MCS_Disconnect_Provider_request;                /*  MCS_Disconnect_Provider_request  */
FPAR                mcId            MCSConnectionId;                /*-------------------------------------------------------*/

        DCL         p               PortalId,
                    ds              DomainSelector;
        START
        COMMENT     'Process an MCS.Disconnect.Provider.request input signal.
                    If the portal is open, this can be done gracefully.
                    ';
        TASK        p := mcId;
        DECISION p in pUsed and portal(p)!notify;
         (True):    TASK    portal(p)!notify := False,
                            ds := portal(p)!domain;
                    DECISION portal(p)!opened and domain(ds)!portals > 0;
                     (True):    OUTPUT Drop.portal(p, RN_user_requested) TO domain(ds)!pid;
                     (False):   CALL  Quit_portal(p, RN_unspecified);
                    ENDDECISION;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                                    /*-----------------------------------------*/
PROCEDURE           MCS_Attach_User_request;                        /*  MCS_Attach_User_request  */
FPAR                label           Natural,                        /*-----------------------------------------*/
                    localDomain     DomainSelector;
```

```
DCL             p               PortalId,
                dp              DataPriority,
                ds              DomainSelector;
START
COMMENT         'Process an MCS.Attach.User.request input signal.
                Allocate a portal, expecting to create an attachment,
                and open the portal.
                ';
DECISION pFree = Empty;
 (True):        TASK    result := RT_congested;
                JOIN 1f;
ELSE:ENDDECISION;
TASK            ds := localDomain;
DECISION ds in dUsed;
 (False):       TASK    result := RT_no_such_domain;
                 1f :
                OUTPUT  MCS.Attach.User.confirm
                        (label, result, Null, 0);
                RETURN;
ELSE:ENDDECISION;
TASK            p := Pick(pFree),
                pFree := Del(p, pFree),
                pUsed := Incl(p, pUsed),
                portal(p)!kind := Attached,
                portal(p)!label := label,
                portal(p)!domain := ds,
                portal(p)!opened := False,
                portal(p)!notify := False,
                portal(p)!pids(0) := Null,
                portal(p)!proc(0) := Engaged,
                dp := 1;
 2b :           /* for dp = 1..3 */
DECISION dp < 4;
 (True):        TASK    portal(p)!pids(dp) := Null,
                        portal(p)!proc(dp) := Nil,
                        dp := dp + 1;
                JOIN 2b;
ELSE:ENDDECISION;
CALL            Open_portal(p);
RETURN;
ENDPROCEDURE;


                                                        /*----------------------*/
PROCEDURE       Open_portal;                            /*  Open_portal  */
FPAR            p               PortalId;               /*----------------------*/


DCL             dp              DataPriority,
                ds              DomainSelector,
                numP            Natural,
                parameters      DomainParameters;
START
COMMENT         'When all TCs have been established, or if this
                is a user attachment, open the portal to a domain.
                If the domain process is stopping, try again later.
                Attachments must wait until domain parameters are set.
                ';
TASK            ds := portal(p)!domain,
                dp := 0;
DECISION portal(p)!kind = Attached;
 (True):        TASK    parameters := domain(ds)!minParms,
                        numP := 1;
 (False):       TASK    parameters := portal(p)!parameters,
                        numP := parameters!numPriorities;
ENDDECISION;
 1b :           /* for dp = 0..? */
```

```
        DECISION dp < numP;
         (True):    DECISION portal(p)!proc(dp) = Engaged;
                     (False):RETURN;
                    ELSE:ENDDECISION;
                    TASK     dp := dp + 1;
                    JOIN 1b;
        ELSE:ENDDECISION;
        DECISION domain(ds)!pid = Null;
         (False):   DECISION domain(ds)!portals = 0;
                     (True):  RETURN;
                    ELSE:ENDDECISION;
         (True):    CREATE  Domain(parameters);
                    TASK     domain(ds)!pid := OFFSPRING,
                             domain(ds)!portals := 0,
                             domain(ds)!upward := 0,
                             domain(ds)!parameters := parameters;
        ENDDECISION;
        DECISION portal(p)!kind = Attached;
         (True):    CREATE  Attachment(portal(p)!label, domain(ds)!parameters);
                    TASK     portal(p)!pids(0) := OFFSPRING;
         (False):   DECISION domain(ds)!parameters = parameters;
                     (False): CALL  Quit_portal(p, RN_parameters_unacceptable);
                              RETURN;
                    ELSE:ENDDECISION;
                    DECISION portal(p)!kind = Uplink;
                     (True):  DECISION domain(ds)!upward = 0;
                               (False):  CALL     Quit_portal(p, RN_domain_not_hierarchical);
                                         RETURN;
                               (True):   TASK     domain(ds)!upward := p;
                              ENDDECISION;
                    ELSE:ENDDECISION;
                    DECISION portal(p)!callSide = Called;
                     (False): OUTPUT    MCS.Connect.Provider.confirm
                                        (portal(p)!label, RT_successful, portal(p)!mcId,
                                        portal(p)!parameters, portal(p)!userData);
                    ELSE:ENDDECISION;
        ENDDECISION;
        OUTPUT      Open.portal(p, portal(p)!kind, portal(p)!pids) TO domain(ds)!pid;
        TASK        domain(ds)!portals := domain(ds)!portals + 1,
                    portal(p)!opened := True;
        RETURN;
        ENDPROCEDURE;

                                                                    /*---------------------*/
PROCEDURE           Drop_portal;                                    /*  Drop_portal  */
FPAR                p           PortalId,                           /*---------------------*/
                    reason      Reason;

        START
        COMMENT'    Process a Drop.portal input signal.
                    ';
        CALL        Quit_portal(p, reason);
        RETURN;
        ENDPROCEDURE;

                                                                    /*-----------------------*/
PROCEDURE           Report_portal;                                  /*  Report_portal  */
FPAR                p           PortalId,                           /*-----------------------*/
                    diagnostic  Diagnostic;

        DCL         reason      Reason;
        START
        COMMENT     'Process a Report.portal input signal.
                    For testing, local diagnostic could be logged.
                    ';
        TASK        reason := RN_unspecified;
```

```
        DECISION diagnostic;
         (DC_throughput_inadequate):
                        TASK    reason := RN_provider_initiated;
         (DC_height_limit_exceeded):
                        TASK    reason := RN_domain_not_hierarchical;
        ELSE:ENDDECISION;
        CALL            Quit_portal(p, reason);
        RETURN;
        ENDPROCEDURE;


                                                                        /*-----------*/
PROCEDURE               RJum;                                           /*  RJum  */
FPAR                    pdu             PDUStruct;                      /*-----------*/


        DCL             p               PortalId,
                        dp              DataPriority;
        START
        COMMENT         'Process an RJum input signal.
                        For testing, remote pdu!diagnostic could be logged.
                        ';
        CALL            Identify_sender(p, dp);
        CALL            Quit_portal(p, RN_unspecified);
        RETURN;
        ENDPROCEDURE;


                                                                        /*--------------------*/
PROCEDURE               Quit_portal;                                    /*  Quit_portal  */
FPAR                    p               PortalId,                       /*--------------------*/
                        reason          Reason;


        DCL             result          Result,
                        dp              DataPriority;
        START
        COMMENT         'If necessary, notify the controlling user.
                        Quiesce the portal processes.
                        ';
        DECISION p in pUsed;
         (False):       RETURN;
        ELSE:ENDDECISION;
        DECISION portal(p)!notify;
         (True):        DECISION portal(p)!callSide = Called or portal(p)!opened;
                         (True):    OUTPUT  MCS.Disconnect.Provider.indication
                                            (portal(p)!mcId, reason);
                         (False):   DECISION reason;
                                     (RN_domain_not_hierarchical):
                                            TASK    result := RT_domain_not_hierarchical;
                                     (RN_parameters_unacceptable):
                                            TASK    result := RT_parameters_unacceptable;
                                    ELSE:
                                            TASK    result := RT_unspecified_failure;
                                    ENDDECISION;
                                    OUTPUT  MCS.Connect.Provider.confirm
                                            (portal(p)!label, result, 0, nullParms, NullString);
                        ENDDECISION;
        ELSE:ENDDECISION;
        TASK            portal(p)!notify := False,
                        dp := 0;
         1b :           /* for dp = 0..3 */
        DECISION dp < 4;
         (True):        DECISION portal(p)!proc(dp);
                        (Receiving, Responding, Engaged):
                                OUTPUT  Quit TO portal(p)!pids(dp);
                                TASK    portal(p)!proc(dp) := Quitting;
                        ELSE:ENDDECISION;
                        TASK    dp := dp + 1;
                        JOIN 1b;
```

```
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                                    /*---------*/
PROCEDURE       Quit;                                               /*  Quit  */
                                                                    /*---------*/
        DCL     p           PortalId,
                pSet        PortalIdSet,
                dp          DataPriority,
                ds          DomainSelector;
        START
        COMMENT 'Process a Quit input signal.
                When all processes are quiesced, it is safe
                to shut this portal on the domain.
                If an upward portal, quiesce all others too.
                ';
        CALL    Identify_sender(p, dp);
        DECISION p in pUsed;
         (False): RETURN;
        ELSE:ENDDECISION;
        TASK    portal(p)!proc(dp) := Quit;
        CALL    Quit_portal(p, RN_unspecified);
        TASK    dp := 0;
         1b :   /* for dp = 0..3 */
        DECISION dp < 4;
         (True): DECISION portal(p)!proc(dp);
                 (Receiving, Responding, Engaged, Quitting):
                         RETURN;
                 ELSE:ENDDECISION;
                 TASK    dp := dp + 1;
                 JOIN 1b;
        ELSE:ENDDECISION;
        DECISION portal(p)!opened;
         (False): CALL   Exit_portal(p);
                 RETURN;
        ELSE:ENDDECISION;
        TASK    ds := portal(p)!domain,
                domain(ds)!portals := domain(ds)!portals - 1;
        OUTPUT  Shut.portal(p) TO domain(ds)!pid;
        DECISION portal(p)!kind = Uplink;
         (True): TASK  domain(ds)!upward := 0,
                       pSet := pUsed;
          2b :         /* for p in pSet */
               DECISION pSet = Empty;
                (False):       TASK   p := Pick(pSet),
                                      pSet := Del(p, pSet);
                               DECISION portal(p)!opened and portal(p)!domain = ds;
                                (True):  CALL   Quit_portal(p, RN_domain_disconnected);
                               ELSE:ENDDECISION;
                               JOIN 2b;
               ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                                    /*--------------------*/
PROCEDURE       Shut_portal;                                        /*  Shut_portal  */
FPAR            p           PortalId;                               /*--------------------*/

        DCL     ds          DomainSelector,
                pSet        PortalIdSet;
        START
        COMMENT 'Process a Shut.portal input signal.
                It is now safe to stop the portal processes.
                If this was the last portal, the domain process stops too
                so that it can be recreated with different parameters.
                ';
```

```
        TASK        ds := portal(p)!domain;
        CALL        Exit_portal(p);
        DECISION domain(ds)!portals = 0;
         (True):    TASK    domain(ds)!pid := Null,
                            pSet := pUsed;
                    1b :    /* for p in pSet */
                    DECISION pSet = Empty;
                     (False): TASK      p := Pick(pSet),
                                        pSet := Del(p, pSet);
                              DECISION portal(p)!domain = ds;
                               (True):  CALL    Open_portal(p);
                              ELSE:ENDDECISION;
                              JOIN 1b;
                    ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                                    /*------------------*/
PROCEDURE           Exit_portal;                                    /*  Exit_portal  */
FPAR                p               PortalId;                       /*------------------*/

        DCL         dp              DataPriority;
        START
        COMMENT     'Release the portal and stop its processes.
                    ';
        TASK        pUsed := Del(p, pUsed),
                    pFree := Incl(p, pFree),
                    dp := 0;
         1b :       /* for dp = 0..3 */
        DECISION dp < 4;
         (True):    DECISION portal(p)!proc(dp);
                     (Quit):
                              OUTPUT    Exit TO portal(p)!pids(dp);
                              TASK      portal(p)!proc(dp) := Nil;
                    ELSE:ENDDECISION;
                    TASK    dp := dp + 1;
                    JOIN 1b;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;

        /* Input transitions */

        DCL         p               PortalId,
                    dp              DataPriority,
                    pdu             PDUStruct,
                    mcId            MCSConnectionId,
                    ccId            Natural,
                    tcId            TCEndpointId,
                    reason          Reason,
                    result          Result,
                    diagnostic      Diagnostic,
                    label           Natural,
                    localTSAP       TSAPAddress,
                    localDomain     DomainSelector,
                    remoteTSAP      TSAPAddress,
                    remoteDomain    DomainSelector,
                    upward          Boolean,
                    targetParms     DomainParameters,
                    minParms        DomainParameters,
                    maxParms        DomainParameters,
                    parameters      DomainParameters,
                    targetQOSByPri  TransportQOSByPri,
                    minQOSByPri     TransportQOSByPri,
                    offeredQOS      TransportQOS,
                    minQOS          TransportQOS,
                    userData        UserData;
```

```
        START
        COMMENT   'The state machine contains a single state.
                  ';
        CALL      Initialize_resources;
        NEXTSTATE ~;

STATE ~;INPUT     MCS.Connect.Provider.request(label, localTSAP, localDomain,
                  remoteTSAP, remoteDomain, upward, targetParms, minParms, maxParms,
                  targetQOSByPri, minQOSByPri, userData);
        CALL      MCS_Connect_Provider_request(label, localTSAP, localDomain,
                  remoteTSAP, remoteDomain, upward, targetParms, minParms, maxParms,
                  targetQOSByPri, minQOSByPri, userData);
        NEXTSTATE -;

STATE ~;INPUT     MCS.Connect.Provider.response(mcId, result, parameters, userData);
        CALL      MCS_Connect_Provider_response(mcId, result, parameters, userData);
        NEXTSTATE -;

STATE ~;INPUT     MCS.Disconnect.Provider.request(mcId);
        CALL      MCS_Disconnect_Provider_request(mcId);
        NEXTSTATE -;

STATE ~;INPUT     MCS.Attach.User.request(label, localDomain);
CALL              MCS_Attach_User_request(label, localDomain);
        NEXTSTATE -;

STATE ~;INPUT     T.Connect.indication(tcId, remoteTSAP, localTSAP, offeredQOS, minQOS);
        CALL      T_Connect_indication(tcId, remoteTSAP, localTSAP, offeredQOS, minQOS);
        NEXTSTATE -;

STATE ~;INPUT     Connect.Initial(remoteDomain, localDomain, upward,
                  targetParms, minParms, maxParms, userData);
        CALL      Connect_Initial(remoteDomain, localDomain, upward,
                  targetParms, minParms, maxParms, userData);
        NEXTSTATE -;

STATE ~;INPUT     Connect.Response(result, ccId, parameters, userData);
        CALL      Connect_Response(result, ccId, parameters, userData);
        NEXTSTATE -;

STATE ~;INPUT     Connect.Additional(ccId, dp);
        CALL      Connect_Additional(ccId, dp);
        NEXTSTATE -;

STATE ~;INPUT     Connect.Result(result);
        CALL      Connect_Result(result);
        NEXTSTATE -;

STATE ~;INPUT     Drop.portal(p, reason);
        CALL      Drop_portal(p, reason);
        NEXTSTATE -;

STATE ~;INPUT     Report.portal(p, diagnostic);
        CALL      Report_portal(p, diagnostic);
        NEXTSTATE -;

STATE ~;INPUT     Shut.portal(p);
        CALL      Shut_portal(p);
        NEXTSTATE -;

STATE ~;INPUT     RJum(pdu);
        CALL      RJum(pdu);
        NEXTSTATE -;
```

```
STATE ~;INPUT       Quit;
        CALL        Quit;
        NEXTSTATE -;

ENDPROCESS;
```

Apéndice IV

Especificación en SDL del proceso de dominio

(Este apéndice no es parte integrante de la presente Recomendación)

```
PROCESS     Domain;
FPAR        parameters      DomainParameters;           /* values established in domain */

SYNONYM     maxBufferIds    Natural = EXTERNAL;             /* an implementation limit */

TIMER       Time.portal(PortalId);                      /* at most one timer per portal */

            /* Type definitions */

NEWTYPE         ChannelStruct
STRUCT
        kind                ChannelKind;        /* (Static,UserId,Private,Assigned) */
        joined              PortalIdSet;            /* directions where channel is joined */
        portal              PortalId;           /* if (UserId): the direction to it */
        manager             UserId;             /* if (Private): channel's manager */
        admitted            UserIdSet;          /* if (Private): zero or more users */
        uMerge              UserIdSet;          /* if (Private): still to be merged */
ENDNEWTYPE;

NEWTYPE         Chan                    Array(ChannelId, ChannelStruct);
ENDNEWTYPE;

NEWTYPE         TokenStruct
STRUCT
        kind                TokenKind;          /* (Grabbed,Inhibited,Giving,Ungivable,Given) */
        grabber             UserId;             /* if (Grabbed,Giving,Ungivable): user */
        recipient           UserId;             /* if (Giving,Given): an intended user */
        inhibitors          UserIdSet;          /* if (Inhibited): one or more users */
        uMerge              UserIdSet;          /* if (Inhibited): still to be merged */
ENDNEWTYPE;

NEWTYPE         Token                   Array(TokenId, TokenStruct);
ENDNEWTYPE;

NEWTYPE         BooleanByPri            Array(DataPriority, Boolean);
ENDNEWTYPE;
NEWTYPE         NaturalByPri            Array(DataPriority, Natural);
ENDNEWTYPE;
NEWTYPE         BufferIdByPri           Array(DataPriority, BufferId);
ENDNEWTYPE;
NEWTYPE         BufferIdQueueByPri      Array(DataPriority, BufferIdQueue);
ENDNEWTYPE;
NEWTYPE         PortalIdSetByPri        Array(DataPriority, PortalIdSet);
ENDNEWTYPE;
```

```
NEWTYPE         NaturalByPriByKind          Array(PortalKind, NaturalByPri);
ENDNEWTYPE;

NEWTYPE         PortalStruct
STRUCT
        kind                PortalKind;             /* (Attached,Downlink,Uplink) */
        pids                PIdByPri;               /* processes constituting a portal */
        inCredit            NaturalByPri;           /* permission to allocate inBuffer */
        inBuffer            BufferIdByPri;          /* PDU input coming from a process */
        outReady            BooleanByPri;           /* True if process allows an output */
        outQueue            BufferIdQueueByPri;             /* PDUs awaiting output to process */
        outCount            Natural;                /* number queued for all priorities */
        outFlow             Natural;                /* number output since timer was set */
        elapsed             Duration;               /* interval set for portal timer */
        interval            Duration;               /* new interval to set for timer */
        subHeight           Natural;                /* subordinate's height or zero */
        subInterval         Duration;               /* subordinate's interval or zero */
ENDNEWTYPE;

NEWTYPE     Portal                          Array(PortalId, PortalStruct);
ENDNEWTYPE;

NEWTYPE         PortalIdQueue               Queue(PortalId);
ENDNEWTYPE;

SYNTYPE         BufferId                    = Integer CONSTANTS 0:maxBufferIds
ENDSYNTYPE;

NEWTYPE         BufferIdSet                 SetOf(BufferId);
ENDNEWTYPE;

NEWTYPE         BufferStruct
STRUCT
        receiver            PortalId;           /* source of inCredit and input PDU */
        dataPriority        DataPriority;       /* index into inBuffer and outQueue */
        portals             Natural;            /* number of outQueues buffer is in */
        pdu                 PDUStruct;          /* the content of one domain MCSPDU */
ENDNEWTYPE;

NEWTYPE         Buffer                      Array(BufferId, BufferStruct);
ENDNEWTYPE;

NEWTYPE         BufferIdQueue               Queue(BufferId);
ENDNEWTYPE;

GENERATOR       Queue       (TYPE ItemType)         /* a first-in first-out queue */
LITERALS
        EmptyQueue;
OPERATORS
        Push:   ItemType,  Queue    ->  Queue;      /* appends next item */
        Next:   Queue               ->  ItemType;   /* reveals first item */
        Pull:   Queue               ->  Queue;      /* deletes first item */
AXIOMS
        Next(EmptyQueue) == ERROR!;
        Pull(EmptyQueue) == ERROR!;
        FOR ALL q IN Queue (
        FOR ALL item IN ItemType
        (
                Next(Push(item,q)) == IF q = EmptyQueue THEN item
                                      ELSE Next(q) FI;
                Pull(Push(item,q)) == IF q = EmptyQueue THEN q
                                      ELSE Push(item,Pull(q)) FI;
        ));
DEFAULT
        EmptyQueue;
```

```
ENDGENERATOR;

        /* Data declarations */

DCL     control             PId;                /* the Control process */

DCL     upward              PortalId;           /* unique MCS Connection upward or */
                                                /* zero if this provider is the top */

DCL     merging             Boolean,            /* True if domain is merging upward */
        pdSend              Boolean,            /* True if PDin to be sent downward */
        edSend              Boolean;            /* True if EDrq to be sent upward */

DCL     uMerge              UserIdSet,          /* users still to be merged */
        uConfirm            UserIdSet,          /* users with merge confirmed */
        cMerge              ChannelIdSet,       /* channels still to be merged */
        cConfirm            ChannelIdSet,       /* channels with merge confirmed */
        tMerge              TokenIdSet,         /* tokens still to be merged */
        tConfirm            TokenIdSet,         /* tokens with merge confirmed */

DCL     mcrqQueue           PortalIdQueue,          /* origin of each pending MCrq */
        mtrqQueue           PortalIdQueue,          /* origin of each pending MTrq */
        aurqQueue           PortalIdQueue;          /* origin of each pending AUrq */

DCL     pDrop               PortalIdSet,            /* dropped portals needing DPum */
        uDetach             UserIdSet,          /* disconnected users needing DUrq */
        uRevoke             UserIdSet,          /* revoked token users needing DUrq */
        cLeave              ChannelIdSet,       /* unjoined channels needing CLrq */
        cDisband            ChannelIdSet,           /* unmanaged channels needing CDin */
        tReject             TokenIdSet;         /* ungivable tokens needing TVcf */

DCL     pBufferWait         PortalIdSetByPri,   /* portals requiring an inBuffer */
        pInputWait          PortalIdSetByPri;   /* inputs suspended during merge */

DCL     chan                Chan,               /* resource arrays */
        token               Token,
        portal              Portal,
        buffer              Buffer;

        /* Note: The fields of a chan, token, portal, or buffer
        array element are undefined if the corresponding index
        is not in cUsed, tUsed, pUsed, or bUsed, respectively. */

DCL     cUsed               ChannelIdSet,       /* indexes used */
        tUsed               TokenIdSet,
        pUsed               PortalIdSet,
        bUsed               BufferIdSet;

DCL     cFree               ChannelIdSet,       /* indexes free */
        tFree               TokenIdSet,
        pFree               PortalIdSet,
        bFree               BufferIdSet;

DCL     uUsed               UserIdSet;          /* subset of cUsed */

DCL     numChannelIds       Natural,            /* number in use */
        numUserIds          Natural,
        numTokenIds         Natural;

DCL     height              Natural,            /* current height of provider */
        interval            Duration,           /* min throughput of one MCSPDU */
        maxInterval         Duration;           /* maximum of portal intervals */

DCL     inCredit            NaturalByPriByKind;     /* initializer */

        /* Procedure decomposition */
```

```
/*      Initialize_resources
        Take_user               (c, diagnostic)
        Take_channel            (c, diagnostic)
        Take_token              (t, diagnostic)
        New_user                (u, result)
        New_channel             (c, result)
        Open_portal             (p, pKind, pids)
        Time_portal             (p)
        Drop_portal             (p, reason)
        Shut_portal             (p)
        Clean_queue             (p, queue)
        Erect_domain
        Identify_sender         (p, dp)
        PDU_ready               (dp)
        Input_PDU               (pdu)
        Allocate_buffer         (b)
        Cast_buffer             (b, p)
        Release_buffer          (b)
        Route_user              (u, p)
        Multicast_buffer        (b, uSet)
        Broadcast_buffer        (b)
        Crank_engine
        Drop_portals
        Merge_users
        Merge_channels
        Merge_tokens
        Detach_users
        Leave_channels
        Disband_channels
        Reject_tokens
        Process_PDU             (r, dp)
        Validate_input          (r, b, diagnostic)
        Top_provider            (r, b)
        Apply_PDU                   (r, b, diagnostic)
        Token_status            (pdu)
        Token_route             (u, x, p)
        Delete_user             (u)
        Delete_channel          (c)
        Delete_token            (t)
        Purge_users             (uSet)
        Purge_channels          (cSet)
        Purge_tokens            (tSet)
        Output_buffer           (b, p)  */


                                                        /*-------------------------------*/
PROCEDURE       Initialize_resources;                   /*  Initialize_resources  */
                                                        /*-------------------------------*/
    DCL     c           ChannelId,
            t           TokenId,
            p           PortalId,
            b           BufferId,
            n           NaturalByPri;
    START
    COMMENT 'Initialize data structures during process start-up
            before accepting the first input signal.
            Note that each SetOf automatically defaults to Empty
            and each Queue to EmptyQueue.
            Fixed buffer credits are an example; other values could
            be computed from maxPortalIds and maxBufferIds.
            ';
    TASK    control := PARENT,
            upward := 0,
            merging := False,
            pdSend := False,
            edSend := False,
            numChannelIds := 0,
```

```
                numUserIds := 0,
                numTokenIds := 0,
                height := 1,
                interval := IF parameters!minThroughput = 0 THEN 0 ELSE oneSecond *
                    (Float(parameters!maxMCSPDUsize) / Float(parameters!minThroughput)) FI,
                maxInterval := 0,
                c := 65535,
                t := 65535,
                p := maxPortalIds,
                b := maxBufferIds;
 1b :           /* for c = ?..1 */
DECISION c > 0;
 (True):        TASK    cFree := Incl(c, cFree),
                        c := c - 1;
                JOIN 1b;
ELSE:ENDDECISION;
 2b :           /* for t = ?..1 */
TASK            tFree := Incl(t, tFree);
DECISION t > 1;
 (True):        TASK    t := t - 1;
                JOIN 2b;
ELSE:ENDDECISION;
 3b :           /* for p = ?..1 */
DECISION p > 0;
 (True):        TASK    pFree := Incl(p, pFree),
                        p := p - 1;
                JOIN 3b;
ELSE:ENDDECISION;
 4b :           /* for b = ?..1 */
DECISION b > 0;
 (True):        TASK    bFree := Incl(b, bFree),
                        b := b - 1;
                JOIN 4b;
ELSE:ENDDECISION;
TASK            n(0) := 2, n(1) := 1, n(2) := 1, n(3) := 1,
                inCredit(Attached) := n,
                n(0) := 3, n(1) := 2, n(2) := 1, n(3) := 1,
                inCredit(Downlink) := n,
                n(0) := 4, n(1) := 3, n(2) := 2, n(3) := 1,
                inCredit(Uplink) := n;
RETURN;
ENDPROCEDURE;


                                                                        /*------------------*/
PROCEDURE       Take_user;                                              /*  Take_user  */
FPAR            c               ChannelId,                              /*------------------*/
        IN/OUT  diagnostic      Diagnostic;


        DCL     u               UserId;
        START
        COMMENT 'Put the free channel id to use as a user id.
                ';
        DECISION numUserIds < parameters!maxUserIds;
         (False):       TASK    diagnostic := DC_too_many_users;
                        RETURN;
        ELSE:ENDDECISION;
        CALL            Take_channel(c, diagnostic);
        DECISION diagnostic = DC_OK;
         (True):        TASK    u := UserId(c),
                                numUserIds := numUserIds + 1,
                                uUsed := Incl(u, uUsed),
                                chan(c)!kind := UserId;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;
```

```
                                                                    /*-----------------------*/
PROCEDURE           Take_channel;                                   /*  Take_channel  */
FPAR                c               ChannelId,                      /*-----------------------*/
        IN/OUT      diagnostic      Diagnostic;

        START
        COMMENT     'Put the free channel id to unspecified use.
                    ';
        DECISION c in cFree;
         (False):   TASK    diagnostic := DC_channel_id_conflict;
                    RETURN;
        ELSE:ENDDECISION;
        DECISION numChannelIds < parameters!maxChannelIds;
         (False):   TASK    diagnostic := DC_too_many_channels;
                    RETURN;
        ELSE:ENDDECISION;
        TASK        diagnostic := DC_OK,
                    numChannelIds := numChannelIds + 1,
                    cFree := Del(c, cFree),
                    cUsed := Incl(c, cUsed),
                    chan(c)!joined := Empty,
                    chan(c)!admitted := Empty;
        RETURN;
        ENDPROCEDURE;

                                                                    /*---------------------*/
PROCEDURE           Take_token;                                     /*  Take_token  */
FPAR                t               TokenId,                        /*---------------------*/
        IN/OUT      diagnostic      Diagnostic;

        START
        COMMENT     'Put the free token id to unspecified use.
                    ';
        DECISION t in tFree;
         (False):   TASK    diagnostic := DC_token_id_conflict;
                    RETURN;
        ELSE:ENDDECISION;
        DECISION numTokenIds < parameters!maxTokenIds;
         (False):   TASK    diagnostic := DC_too_many_tokens;
                    RETURN;
        ELSE:ENDDECISION;
        TASK        diagnostic := DC_OK,
                    numTokenIds := numTokenIds + 1,
                    tFree := Del(t, tFree),
                    tUsed := Incl(t, tUsed),
                    token(t)!inhibitors := Empty;
        RETURN;
        ENDPROCEDURE;

                                                                    /*-------------------*/
PROCEDURE           New_user;                                       /*  New_user  */
FPAR    IN/OUT      u               UserId,                         /*-------------------*/
        IN/OUT      result          Result;

        DCL         c               ChannelId;
        START
        COMMENT     'Allocate a new user id or fail and return 0.
                    ';
        TASK        u := 0;
        DECISION numUserIds < parameters!maxUserIds;
         (False):   TASK    result := RT_too_many_users;
                    RETURN;
        ELSE:ENDDECISION;
        CALL        New_channel(c, result);
        DECISION result = RT_successful;
         (True):    TASK    u := UserId(c),
                            numUserIds := numUserIds + 1,
```

```
                        uUsed := Incl(u, uUsed),
                        chan(c)!kind := UserId;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;


                                                                    /*----------------------*/
PROCEDURE           New_channel;                                    /*  New_channel  */
FPAR    IN/OUT      c           ChannelId,                          /*----------------------*/
        IN/OUT      result      Result;


        DCL         diagnostic  Diagnostic;
        START
        COMMENT     'Allocate a new channel id or fail and return 0.
                    ';
        TASK        c := 0;
        DECISION numChannelIds < parameters!maxChannelIds;
         (False):   TASK    result := RT_too_many_channels;
                    RETURN;
        ELSE:ENDDECISION;
         1b :       /* keep trying */
        TASK        c := ANY(ChannelId);  /* randomize */
        DECISION c >= 1001 and c in cFree;
         (False):   JOIN 1b;
        ELSE:ENDDECISION;
        CALL        Take_channel(c, diagnostic);
        TASK        result := RT_successful;
        RETURN;
        ENDPROCEDURE;


                                                                    /*----------------------*/
PROCEDURE           Open_portal;                                    /*  Open_portal  */
FPAR                p           PortalId,                           /*----------------------*/
                    pKind       PortalKind,
                    pids        PIdByPri;


        DCL         pid         PId,
                    dp          DataPriority,
                    c           ChannelId,
                    cSet        ChannelIdSet,
                    t           TokenId,
                    tSet        TokenIdSet;
        START
        COMMENT     'Process an Open.portal input signal.
                    Accept a new MCS connection or attachment to the domain.
                    If this is an upward connection, prepare for merge.
                    ';
        DECISION pKind = Attached;
         (True):    TASK    pid := pids(0),
                            pids(1) := pid,
                            pids(2) := pid,
                            pids(3) := pid;
        ELSE:ENDDECISION;
        TASK        pFree := Del(p, pFree),
                    pUsed := Incl(p, pUsed),
                    portal(p)!kind := pKind,
                    portal(p)!pids := pids,
                    portal(p)!inCredit := inCredit(pKind),
                    portal(p)!outCount := 0,
                    portal(p)!interval := IF pKind = Uplink THEN 0 ELSE interval FI,
                    portal(p)!subHeight := 0,
                    portal(p)!subInterval := 0,
                    dp := 0;
         1b :       /* for dp = 0..? */
```

```
DECISION dp < parameters!numPriorities;
 (True):        TASK      portal(p)!inBuffer(dp) := 0,
                          portal(p)!outReady(dp) := False,
                          portal(p)!outQueue(dp) := EmptyQueue,
                          pBufferWait(dp) := Incl(p, pBufferWait(dp)),
                          dp := dp + 1;
                JOIN 1b;
ELSE:ENDDECISION;
DECISION portal(p)!kind = Uplink;
 (True):        TASK      upward := p,
                          merging := True,
                          pdSend := True,
                          edSend := True,
                          uMerge := uUsed,
                          uConfirm := Empty,
                          cMerge := cUsed,
                          cConfirm := Empty,
                          tMerge := tUsed,
                          tConfirm := Empty,
                          cLeave := Empty,
                          cDisband := Empty,
                          tReject := Empty,
                          cSet := cMerge;
                 2b :     /* for c in cSet */
                DECISION cSet = Empty;
                 (False): TASK      c := Pick(cSet),
                                    cSet := Del(c, cSet),
                                    chan(c)!uMerge := chan(c)!admitted;
                          JOIN 2b;
                ELSE:ENDDECISION;
                TASK      tSet := tMerge;
                 3b :     /* for t in tSet */
                DECISION tSet = Empty;
                 (False): TASK      t := Pick(tSet),
                                    tSet := Del(t, tSet),
                                    token(t)!uMerge := token(t)!inhibitors;
                          JOIN 3b;
                ELSE:ENDDECISION;
ELSE:ENDDECISION;
CALL            Erect_domain;
CALL            Crank_engine;
RETURN;
ENDPROCEDURE;



                                                                  /*--------------------*/
PROCEDURE       Time_portal;                                      /*  Time_portal  */
FPAR            p               PortalId;                         /*--------------------*/


START
COMMENT         'Process a Time.portal input signal.
                Ensure that minimum throughput is maintained.
                Allow for the fact that outFlow takes integer steps.
                ';
DECISION portal(p)!elapsed > interval * (Float(portal(p)!outFlow) + 0.99);
 (True):        OUTPUT    Report.portal(p, DC_throughput_inadequate) TO control;
 (False):       TASK      portal(p)!outFlow := 0,
                          portal(p)!elapsed := portal(p)!interval;
                SET(NOW + portal(p)!interval, Time.portal(p));
ENDDECISION;
CALL            Crank_engine;
RETURN;
ENDPROCEDURE;
```

```
                                                                    /*--------------------*/
PROCEDURE       Drop_portal;                                        /*  Drop_portal  */
FPAR            p           PortalId,                               /*--------------------*/
                reason      Reason;

        START
        COMMENT 'Process a Drop.portal input signal,
                knowing the only reason is user-requested.
                ';
        DECISION p in pUsed and portal(p)!kind /= Attached;
         (True):    TASK    pDrop := Incl(p, pDrop);
        ELSE:ENDDECISION;
        CALL        Crank_engine;
        RETURN;
        ENDPROCEDURE;


                                                                    /*--------------------*/
PROCEDURE       Shut_portal;                                        /*  Shut_portal  */
FPAR            p           PortalId;                               /*--------------------*/

        DCL         dp          DataPriority,
                    b           BufferId,
                    bSet        BufferIdSet,
                    c           ChannelId,
                    cSet        ChannelIdSet,
                    u           UserId;
        START
        COMMENT 'Process a Shut.portal input signal.
                Remove the corresponding MCS connection or attachment.
                When the last one is gone, stop the domain process.
                ';
        RESET(Time.portal(p));
        TASK        pUsed := Del(p, pUsed),
                    pFree := Incl(p, pFree),
                    pDrop := Del(p, pDrop),
                    dp := 0;
         1b :       /* for dp = 0..? */
        DECISION dp < parameters!numPriorities;
         (True):    2b :    /* for b in outQueue(dp) */
                    DECISION portal(p)!outQueue(dp) = EmptyQueue;
                     (False):   TASK    b := Next(portal(p)!outQueue(dp)),
                                        portal(p)!outQueue(dp) := Pull(portal(p)!outQueue(dp)),
                                        buffer(b)!portals := buffer(b)!portals - 1;
                                CALL    Release_buffer(b);
                                JOIN 2b;
                    ELSE:ENDDECISION;
                    TASK    b := portal(p)!inBuffer(dp),
                            portal(p)!inBuffer(dp) := 0;
                    CALL    Release_buffer(b);
                    TASK    pBufferWait(dp) := Del(p, pBufferWait(dp)),
                            pInputWait(dp) := Del(p, pInputWait(dp)),
                            dp := dp + 1;
                    JOIN 1b;
        ELSE:ENDDECISION;
        TASK        bSet := bUsed;
         3b :       /* for b in bSet */
        DECISION bSet = Empty;
         (False):   TASK    b := Pick(bSet),
                            bSet := Del(b, bSet);
                    DECISION buffer(b)!receiver = p;
                     (True):    TASK    buffer(b)!receiver := 0;
                    ELSE:ENDDECISION;
                    JOIN 3b;
        ELSE:ENDDECISION;
        TASK        cSet := cUsed;
         4b :       /* for c in cSet */
```

```
        DECISION cSet = Empty;
         (False):    TASK     c := Pick(cSet),
                              cSet := Del(c, cSet);
                     DECISION p in chan(c)!joined;
                      (True):  TASK     chan(c)!joined := Del(p, chan(c)!joined);
                               DECISION chan(c)!joined = Empty;
                                (True):  TASK     cLeave := Incl(c, cLeave);
                               ELSE:ENDDECISION;
                     ELSE:ENDDECISION;
                     DECISION chan(c)!kind = UserId and chan(c)!portal = p;
                      (True):  TASK     chan(c)!portal := 0,
                                        u := UserId(c),
                                        uDetach := Incl(u, uDetach),
                                        cLeave := Del(c, cLeave);
                     ELSE:ENDDECISION;
                     JOIN 4b;
        ELSE:ENDDECISION;
        DECISION portal(p)!kind = Uplink;
         (True):     TASK     upward := 0,
                              merging := False,
                              mcrqQueue := EmptyQueue,
                              mtrqQueue := EmptyQueue,
                              aurqQueue := EmptyQueue;
         (False):    CALL     Clean_queue(p, mcrqQueue);
                     CALL     Clean_queue(p, mtrqQueue);
                     CALL     Clean_queue(p, aurqQueue);
        ENDDECISION;
        OUTPUT       Shut.portal(p) TO control;
        CALL         Erect_domain;
        CALL         Crank_engine;
        RETURN;
        ENDPROCEDURE;

                                                                    /*----------------------*/
PROCEDURE            Clean_queue;                                   /*  Clean_queue  */
FPAR                 p              PortalId,                       /*----------------------*/
        IN/OUT       queue          PortalIdQueue;

        DCL          x              PortalId,
                     clean          PortalIdQueue;
        START
        COMMENT      'Remove a shut portal id from a queue.
                     ';
        TASK         clean := EmptyQueue;
         1b :        /* for x in queue */
        DECISION queue = EmptyQueue;
         (False):    TASK     x := Next(queue),
                              queue := Pull(queue),
                              x := IF x = p THEN 0 ELSE x FI,
                              clean := Push(x, clean);
                     JOIN 1b;
        ELSE:ENDDECISION;
        TASK         queue := clean;
        RETURN;
        ENDPROCEDURE;

                                                                    /*----------------------*/
PROCEDURE            Erect_domain;                                  /*  Erect_domain  */
                                                                    /*----------------------*/

        DCL          p              PortalId,
                     pSet           PortalIdSet,
                     h              Natural,
                     hmax           Natural,
                     i              Duration,
                     imax           Duration;
```

```
        START
        COMMENT     'Recalculate the provider height and maxInterval.
                    If either changed, communicate them upward.
                    ';
        TASK        hmax := 1,
                    imax := 0,
                    pSet := pUsed;
         1b :       /* for p in pSet */
        DECISION pSet = Empty;
         (False):   TASK     p := Pick(pSet),
                             pSet := Del(p, pSet),
                             h := portal(p)!subHeight + 1,
                             hmax := IF h > hmax THEN h ELSE hmax FI,
                             i := portal(p)!interval,
                             imax := IF i > imax THEN i ELSE imax FI;
                    JOIN 1b;
        ELSE:ENDDECISION;
        DECISION height = hmax and maxInterval = imax;
         (False):   TASK     height := hmax,
                             maxInterval := imax,
                             edSend := True;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;
```

```
                                                                /*-------------------------*/
PROCEDURE           Identify_sender;                            /*  Identify_sender  */
FPAR    IN/OUT      p               PortalId,                   /*-------------------------*/
        IN/OUT      dp              DataPriority;


        DCL         pSet            PortalIdSet;
        START
        COMMENT     'An alternative would be to carry this
                    information explicitly in SDL signals.
                    ';
        TASK        pSet := pUsed;
         1b :       /* for p in pSet */
        DECISION pSet = Empty;
         (False):   TASK     p := Pick(pSet),
                             pSet := Del(p, pSet),
                             dp := 0;
                    2b :     /* for dp = 0..? */
                    DECISION dp < parameters!numPriorities;
                     (True):  DECISION portal(p)!pids(dp) = SENDER;
                               (True):    RETURN;
                              ELSE:ENDDECISION;
                              TASK       dp := dp + 1;
                              JOIN 2b;
                    ELSE:ENDDECISION;
                    JOIN 1b;
        ELSE:ENDDECISION;
        TASK        p := 0,
                    dp := 0;
        RETURN;
        ENDPROCEDURE;
```

```
                                                                /*--------------------*/
PROCEDURE           PDU_ready;                                  /*  PDU_ready  */
FPAR                dp              DataPriority;               /*--------------------*/
        DCL         p               PortalId,
                    x               DataPriority,
                    b               BufferId;
```

```
        START
        COMMENT   'Process a PDU.ready input signal.
                  If a buffer is waiting, it can be output.
                  ';
        CALL      Identify_sender(p, x);
        DECISION p in pUsed;
         (False):     RETURN;
        ELSE:ENDDECISION;
        DECISION portal(p)!outQueue(dp) = EmptyQueue;
         (True):      TASK     portal(p)!outReady(dp) := True;
         (False):     TASK     b := Next(portal(p)!outQueue(dp)),
                               portal(p)!outQueue(dp) := Pull(portal(p)!outQueue(dp)),
                               buffer(b)!portals := buffer(b)!portals - 1;
                      CALL     Output_buffer(b, p);
                      TASK     portal(p)!outReady(dp) := False;
                      CALL     Release_buffer(b);
                      TASK     portal(p)!outFlow := portal(p)!outFlow + 1,
                               portal(p)!outCount := portal(p)!outCount - 1;
                      DECISION portal(p)!outCount = 0;
                       (True):  RESET(Time.portal(p));
                      ELSE:ENDDECISION;
        ENDDECISION;
        CALL      Crank_engine;
        RETURN;
        ENDPROCEDURE;


                                                                    /*------------------*/
PROCEDURE         Input_PDU;                                        /*  Input_PDU  */
FPAR              pdu          PDUStruct;                           /*------------------*/

        DCL       p            PortalId,
                  dp           DataPriority,
                  b            BufferId;
        START
        COMMENT   'Process an MCSPDU input signal.
                  If merging, requests and responses must wait.
                  ';
        CALL      Identify_sender(p, dp);
        DECISION p in pUsed;
         (False):     RETURN;
        ELSE:ENDDECISION;
        DECISION portal(p)!kind = Attached;
         (True):      DECISION pdu!kind;
                       (SDrq, SDin, USrq, USin):
                              TASK     dp := pdu!dataPriority,
                                       dp := IF dp < parameters!numPriorities THEN dp
                                                     ELSE parameters!numPriorities - 1 FI;
                      ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        TASK      b := portal(p)!inBuffer(dp),
                  buffer(b)!pdu := pdu;
        DECISION p = upward or not merging;
         (True):      CALL     Process_PDU(p, dp);
         (False):     TASK     pInputWait(dp) := Incl(p, pInputWait(dp));
        ENDDECISION;
        CALL      Crank_engine;
        RETURN;
        ENDPROCEDURE;


                                                                    /*-------------------------*/
PROCEDURE         Allocate_buffer;                                  /*  Allocate_buffer  */
FPAR    IN/OUT    b            BufferId;                            /*-------------------------*/

        START
        COMMENT   'Allocate a free buffer id or fail and return 0.
                  ';
```

```
        DECISION b /= 0 and buffer(b)!portals = 0;
         (True):        RETURN;
        ELSE:ENDDECISION;
        TASK            b := 0;
        DECISION bFree = Empty;
         (False):       TASK    b := Pick(bFree),
                                bFree := Del(b, bFree),
                                bUsed := Incl(b, bUsed),
                                buffer(b)!receiver := 0,
                                buffer(b)!dataPriority := 0,
                                buffer(b)!portals := 0;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                                        /*--------------------*/
PROCEDURE       Cast_buffer;                                            /*  Cast_buffer  */
FPAR            b               BufferId,                               /*--------------------*/
                p               PortalId;

        DCL             dp              DataPriority;
        START
        COMMENT         'Send buffer containing an MCSPDU to a single output.
                        ';
        DECISION p = 0;
         (True):        RETURN;
        ELSE:ENDDECISION;
         TASK           dp := buffer(b)!dataPriority;
        DECISION portal(p)!outReady(dp);
         (True):        CALL    Output_buffer(b, p);
                        TASK    portal(p)!outReady(dp) := False;
         (False):       DECISION portal(p)!outCount = 0 and portal(p)!interval > 0;
                         (True):    TASK  portal(p)!outFlow := 0,
                                          portal(p)!elapsed := portal(p)!interval;
                                    SET(NOW + portal(p)!interval, Time.portal(p));
                        ELSE:ENDDECISION;
                        TASK    portal(p)!outQueue(dp) := Push(b, portal(p)!outQueue(dp)),
                                buffer(b)!portals := buffer(b)!portals + 1,
                                portal(p)!outCount := portal(p)!outCount + 1;
        ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                                        /*-------------------------*/
PROCEDURE       Release_buffer;                                         /*  Release_buffer  */
FPAR            b               BufferId;                               /*-------------------------*/

        DCL             p               PortalId,
                        dp              DataPriority;
        START
        COMMENT         'Release a buffer that is no longer needed.
                        ';
        DECISION b = 0 or buffer(b)!portals > 0;
         (True):        RETURN;
        ELSE:ENDDECISION;
        TASK            bUsed := Del(b, bUsed),
                        bFree := Incl(b, bFree),
                        p := buffer(b)!receiver,
                        dp := buffer(b)!dataPriority;
        DECISION p = 0;
         (False):       TASK    portal(p)!inCredit(dp) := portal(p)!inCredit(dp) + 1;
                        DECISION portal(p)!inBuffer(dp) = 0;
                         (True):    TASK  pBufferWait(dp) := Incl(p, pBufferWait(dp));
                        ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;
```

```
                                                                    /*--------------------*/
PROCEDURE       Route_user;                                         /*  Route_user  */
FPAR            u           UserId,                                 /*--------------------*/
      IN/OUT    p           PortalId;

      DCL       c           ChannelId;
      START
      COMMENT   'Return the portal id that leads toward a user.
                ';
      TASK      p := 0;
      DECISION u in uUsed;
       (True):  TASK    c := ChannelId(u),
                        p := chan(c)!portal;
      ELSE:ENDDECISION;
      RETURN;
      ENDPROCEDURE;

                                                                    /*--------------------------*/
PROCEDURE       Multicast_buffer;                                   /*  Multicast_buffer  */
FPAR            b           BufferId,                               /*--------------------------*/
                uSet        UserIdSet;

      DCL       u           UserId,
                p           PortalId,
                pSet        PortalIdSet;
      START
      COMMENT   'Send buffer containing an MCSPDU to multiple users.
                ';
      TASK      pSet := Empty;
       1b :     /* for u in uSet */
      DECISION uSet = Empty;
       (False): TASK    u := Pick(uSet),
                        uSet := Del(u, uSet);
                CALL    Route_user(u, p);
                TASK    pSet := Incl(p, pSet);
                JOIN 1b;
      ELSE:ENDDECISION;
       2b :     /* for p in pSet */
      DECISION pSet = Empty;
       (False): TASK    p := Pick(pSet),
                        pSet := Del(p, pSet);
                CALL    Cast_buffer(b, p);
                JOIN 2b;
      ELSE:ENDDECISION;
      RETURN;
      ENDPROCEDURE;

                                                                    /*----------------------------*/
PROCEDURE       Broadcast_buffer;                                   /*  Broadcast_buffer  */
FPAR            b           BufferId;                               /*----------------------------*/

      DCL       pdu         PDUStruct,
                p           PortalId,
                pSet        PortalIdSet;
      START
      COMMENT   'Send buffer containing an MCSPDU to the whole subtree.
                ';
      TASK      pdu := buffer(b)!pdu;
      DECISION (pdu!kind = PCin and pdu!detachUserIds = Empty
                                and pdu!purgeChannelIds = Empty)
                or (pdu!kind = PTin and pdu!purgeTokenIds = Empty)
                or (pdu!kind = DUin and pdu!userIds = Empty);
       (True):  RETURN;
      ELSE:ENDDECISION;
      TASK      pSet := pUsed;
       1b :     /* for p in pSet */
```

```
        DECISION pSet = Empty;
         (False):       TASK      p := Pick(pSet),
                                  pSet := Del(p, pSet);
                        DECISION portal(p)!kind;
                         (Attached):
                                  DECISION pdu!kind = PDin;
                                   (False):CALL          Cast_buffer(b, p);
                                  ELSE:ENDDECISION;
                         (Downlink):
                                  CALL        Cast_buffer(b, p);
                        ELSE:ENDDECISION;
                        JOIN 1b;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;


                                                                          /*-----------------------*/
PROCEDURE               Crank_engine;                                     /*  Crank_engine  */
                                                                          /*-----------------------*/
        DCL             b               BufferId,
                        p               PortalId,
                        dp              DataPriority;
        START
        COMMENT         'After individual processing of each input signal,
                        look for the most deserving work to do next.
                        This advances incrementally through stages of a merge;
                        else it cleans up users, channels, tokens left behind.
                        Buffers are assigned to accept new inputs, with preference
                        to PDUs flowing downward and to higher priorities first.
                        ';
        TASK            b := 0;
        DECISION pdSend;
         (True):        CALL      Allocate_buffer(b);
                        DECISION b = 0;
                         (True):    RETURN;
                        ELSE:ENDDECISION;
                        TASK      pdSend := False,
                                  buffer(b)!pdu!kind := PDin,
                                  buffer(b)!pdu!heightLimit := parameters!maxHeight - 1;
                        CALL      Broadcast_buffer(b);
        ELSE:ENDDECISION;
        DECISION edSend;
         (True):        CALL      Allocate_buffer(b);
                        DECISION b = 0;
                         (True):    RETURN;
                        ELSE:ENDDECISION;
                        TASK      edSend := False,
                                  buffer(b)!pdu!kind := EDrq,
                                  buffer(b)!pdu!subHeight := height,
                                  buffer(b)!pdu!subInterval := maxInterval;
                        CALL      Cast_buffer(b, upward);
        ELSE:ENDDECISION;
        CALL            Release_buffer(b);
        TASK            dp := 0;
         1b :           /* for dp = 0..? */
        DECISION dp < parameters!numPriorities;
         (True):        DECISION upward in pBufferWait(dp);
                         (False):  TASK        dp := dp + 1;
                                   JOIN 1b;
                        ELSE:ENDDECISION;
                        TASK      p := upward,
                                  b := 0;
                        CALL      Allocate_buffer(b);
                        DECISION b = 0;
                         (True):    RETURN;
                        ELSE:ENDDECISION;
```

```
                TASK    pBufferWait(dp) := Del(p, pBufferWait(dp)),
                        buffer(b)!receiver := p,
                        buffer(b)!dataPriority := dp,
                        portal(p)!inCredit(dp) := portal(p)!inCredit(dp) - 1,
                        portal(p)!inBuffer(dp) := b;
                OUTPUT  PDU.ready(dp) TO portal(p)!pids(dp);
                JOIN 1b;
ELSE:ENDDECISION;
CALL            Drop_portals;
DECISION merging;
 (True):        CALL    Merge_users;
                DECISION uConfirm = uUsed;
                 (True):    CALL        Merge_tokens;
                            DECISION tConfirm = tUsed;
                             (True):    CALL  Merge_channels;
                                        DECISION cConfirm = cUsed;
                                         (True):   TASK     merging := False;
                                        ELSE:ENDDECISION;
                            ELSE:ENDDECISION;
                ELSE:ENDDECISION;
                DECISION merging;
                 (True):    RETURN;
                 (False):   TASK        dp := 0;
                             2b :       /* for dp = 0..? */
                            DECISION dp < parameters!numPriorities;
                             (True):    /* for p in pInputWait(dp) */
                                        DECISION pInputWait(dp) = Empty;
                                         (True):  TASK     dp := dp + 1;
                                                  JOIN 2b;
                                        ELSE:ENDDECISION;
                                        TASK    p := Pick(pInputWait(dp)),
                                                pInputWait(dp) := Del(p, pInputWait(dp));
                                        CALL    Process_PDU(p, dp);
                                        JOIN 2b;
                            ELSE:ENDDECISION;
                ENDDECISION;
ELSE:ENDDECISION;
CALL            Detach_users;
CALL            Leave_channels;
CALL            Disband_channels;
CALL            Reject_tokens;
TASK            dp := 0;
 3b :           /* for dp = 0..? */
DECISION dp < parameters!numPriorities;
 (True):        /* for p in pBufferWait(dp) */
                DECISION pBufferWait(dp) = Empty;
                 (True):    TASK        dp := dp + 1;
                            JOIN 3b;
                ELSE:ENDDECISION;
                TASK    p := Pick(pBufferWait(dp)),
                        b := 0;
                CALL    Allocate_buffer(b);
                DECISION b = 0;
                 (True):    RETURN;
                ELSE:ENDDECISION;
                TASK    pBufferWait(dp) := Del(p, pBufferWait(dp)),
                        buffer(b)!receiver := p,
                        buffer(b)!dataPriority := dp,
                        portal(p)!inCredit(dp) := portal(p)!inCredit(dp) - 1,
                        portal(p)!inBuffer(dp) := b;
                OUTPUT  PDU.ready(dp) TO portal(p)!pids(dp);
                JOIN 3b;
ELSE:ENDDECISION;
RETURN;
ENDPROCEDURE;
```

```
                                                                    /*----------------------*/
PROCEDURE       Drop_portals;                                       /*  Drop_portals  */
                                                                    /*----------------------*/
        DCL     b               BufferId,
                p               PortalId,
                pdu             PDUStruct;
        START
        COMMENT 'Generate DPum MCSPDUs requested by controller.
                ';
        TASK    b := 0,
                pdu!kind := DPum,
                pdu!reason := RN_user_requested;
         1b :   /* for p in pDrop */
        DECISION pDrop = Empty;
         (False): CALL  Allocate_buffer(b);
                DECISION b = 0;
                 (True): RETURN;
                ELSE:ENDDECISION;
                TASK    p := Pick(pDrop),
                        pDrop := Del(p, pDrop),
                        buffer(b)!pdu := pdu;
                CALL    Cast_buffer(b, p);
                JOIN 1b;
        ELSE:ENDDECISION;
        CALL    Release_buffer(b);
        RETURN;
        ENDPROCEDURE;


                                                                    /*----------------------*/
PROCEDURE       Merge_users;                                        /*  Merge_users  */
                                                                    /*----------------------*/
        DCL     b               BufferId,
                u               UserId,
                c               ChannelId,
                pdu             PDUStruct,
                cAttr           ChannelAttributes;
        START
        COMMENT 'Generate MCrq MCSPDUs to merge user ids upward.
                Fill them to maximum size that encoding allows.
                ';
        TASK    b := 0,
                pdu!kind := MCrq,
                pdu!mergeChannels := Empty;
         1b :   /* for u in uMerge */
        DECISION uMerge = Empty;
         (False): CALL  Allocate_buffer(b);
                DECISION b = 0;
                 (True): RETURN;
                ELSE:ENDDECISION;
                TASK    u := Pick(uMerge),
                        uMerge := Del(u, uMerge),
                        c := ChannelId(u),
                        cMerge := Del(c, cMerge),
                        cAttr!channelId := c,
                        cAttr!kind := UserId,
                        cAttr!joined := (chan(c)!joined /= Empty),
                        pdu!mergeChannels := Incl(cAttr, pdu!mergeChannels);
                DECISION'encoding of pdu + 9 <= maxMCSPDUsize';
                 ('True'): JOIN 1b;
                ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        DECISION pdu!mergeChannels = Empty;
         (False): TASK  buffer(b)!pdu := pdu,
                        pdu!mergeChannels := Empty;
                CALL    Cast_buffer(b, upward);
                JOIN 1b;
```

```
        ELSE:ENDDECISION;
        CALL        Release_buffer(b);
        RETURN;
        ENDPROCEDURE;


                                                                    /*----------------------------*/
PROCEDURE           Merge_channels;                                 /*  Merge_channels  */
                                                                    /*----------------------------*/
        DCL         b               BufferId,
                    c               ChannelId,
                    u               UserId,
                    pdu             PDUStruct,
                    cAttr           ChannelAttributes;
        START
        COMMENT     'Generate MCrq MCSPDUs to merge channel ids upward.
                    Fill them to maximum size that encoding allows.
                    ';
        TASK        b := 0,
                    pdu!kind := MCrq,
                    pdu!mergeChannels := Empty,
                    pdu!purgeChannelIds := Empty;
         1b :       /* for c in cMerge */
        DECISION cMerge = Empty;
         (False):   CALL     Allocate_buffer(b);
                    DECISION b = 0;
                     (True):  RETURN;
                    ELSE:ENDDECISION;
                    TASK     c := Pick(cMerge),
                             cMerge := Del(c, cMerge),
                             cAttr!channelId := c,
                             cAttr!kind := chan(c)!kind,
                             cAttr!manager := chan(c)!manager,
                             cAttr!admitted := Empty,
                             cAttr!joined := (chan(c)!joined /= Empty);
                    DECISION (chan(c)!kind = Static or chan(c)!kind = Assigned)
                             and chan(c)!joined = Empty;
                     (True):  CALL       Delete_channel(c);
                             JOIN 1b;
                    ELSE:ENDDECISION;
                    DECISION chan(c)!kind = Private;
                     (True):  DECISION chan(c)!manager in uUsed;
                               (False):   TASK     pdu!purgeChannelIds :=
                                                       Incl(c, pdu!purgeChannelIds);
                                          JOIN 3f;
                                ELSE:ENDDECISION;
                                 2b :       /* for u in chan(c)!uMerge */
                                DECISION chan(c)!uMerge = Empty;
                                 (False):   TASK     u := Pick(chan(c)!uMerge),
                                                     chan(c)!uMerge := Del(u, chan(c)!uMerge),
                                                     cAttr!admitted := Incl(u, cAttr!admitted),
                                                     pdu!mergeChannels :=          /* try this many */
                                                       Incl(cAttr, pdu!mergeChannels);
                                           DECISION'encoding of pdu + 4 <= maxMCSPDUsize';
                                            ('True'): TASK     pdu!mergeChannels := /* try another */
                                                       Del(cAttr, pdu!mergeChannels);
                                                     JOIN 2b;
                                           ELSE:ENDDECISION;
                                ELSE:ENDDECISION;
                    ELSE:ENDDECISION;
                    TASK     pdu!mergeChannels := Incl(cAttr, pdu!mergeChannels);
                     3f :
                    DECISION'encoding of pdu + 23 <= maxMCSPDUsize';
                     ('True'):  JOIN 1b;
                    ELSE:ENDDECISION;
        ELSE:ENDDECISION;
```

```
        DECISION pdu!mergeChannels = Empty and pdu!purgeChannelIds = Empty;
         (False):       TASK    buffer(b)!pdu := pdu,
                                pdu!mergeChannels := Empty,
                                pdu!purgeChannelIds := Empty;
                        CALL    Cast_buffer(b, upward);
                        JOIN 1b;
        ELSE:ENDDECISION;
        CALL            Release_buffer(b);
        RETURN;
        ENDPROCEDURE;



                                                                        /*------------------------*/
PROCEDURE               Merge_tokens;                                   /*  Merge_tokens  */
                                                                        /*------------------------*/
        DCL             b               BufferId,
                        t               TokenId,
                        u               UserId,
                        pdu             PDUStruct,
                        tAttr           TokenAttributes;
        START
        COMMENT         'Generate MTrq MCSPDUs to merge token ids upward.
                        Fill them to maximum size that encoding allows.
                        ';
        TASK            b := 0,
                        pdu!kind := MTrq,
                        pdu!mergeTokens := Empty,
                        pdu!purgeTokenIds := Empty,
         1b :           /* for t in tMerge */
        DECISION tMerge = Empty;
         (False):       CALL    Allocate_buffer(b);
                        DECISION b = 0;
                         (True):    RETURN;
                        ELSE:ENDDECISION;
                        TASK    t := Pick(tMerge),
                                tMerge := Del(t, tMerge),
                                tAttr!tokenId := t,
                                tAttr!kind := token(t)!kind,
                                tAttr!grabber := token(t)!grabber,
                                tAttr!recipient := token(t)!recipient,
                                tAttr!inhibitors := Empty;
                        DECISION token(t)!kind = Inhibited;
                         (True):    2b :        /* for u in token(t)!uMerge */
                                    DECISION token(t)!uMerge = Empty;
                                     (False):   TASK    u := Pick(token(t)!uMerge),
                                                        token(t)!uMerge := Del(u, token(t)!uMerge),
                                                        tAttr!inhibitors := Incl(u, tAttr!inhibitors),
                                                        pdu!mergeTokens :=          /* try this many */
                                                                        Incl(tAttr, pdu!mergeTokens);
                                                DECISION'encoding of pdu + 4 <= maxMCSPDUsize';
                                                 ('True'): TASK     pdu!mergeTokens := /* try another */
                                                                        Del(tAttr, pdu!mergeTokens);
                                                           JOIN 2b;
                                                ELSE:ENDDECISION;
                                    ELSE:ENDDECISION;
                        ELSE:ENDDECISION;
                        TASK    pdu!mergeTokens := Incl(tAttr, pdu!mergeTokens);
                        DECISION'encoding of pdu + 16 <= maxMCSPDUsize';
                         ('True'):  JOIN 1b;
        ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        DECISION pdu!mergeTokens = Empty;
         (False):       TASK    buffer(b)!pdu := pdu,
                                pdu!mergeTokens := Empty;
                        CALL    Cast_buffer(b, upward);
                        JOIN 1b;
```

```
        ELSE:ENDDECISION;
        CALL        Release_buffer(b);
        RETURN;
        ENDPROCEDURE;


                                                                    /*-----------------------*/
PROCEDURE           Detach_users;                                   /*  Detach_users  */
                                                                    /*-----------------------*/
        DCL         b               BufferId,
                    u               UserId,
                    pdu             PDUStruct,
                    diagnostic      Diagnostic;
        START
        COMMENT     'Generate DUrq MCSPDUs to detach users.
                    Fill them to maximum size that encoding allows.
                    ';
        TASK        b := 0,
                    pdu!kind := DUrq,
                    pdu!userIds := Empty;
         1b :       /* for u in uDetach */
        DECISION uDetach = Empty;
         (False):   CALL     Allocate_buffer(b);
                    DECISION b = 0;
                     (True):   RETURN;
                    ELSE:ENDDECISION;
                    TASK     u := Pick(uDetach),
                             uDetach := Del(u, uDetach),
                             uRevoke := Del(u, uRevoke),
                             pdu!reason := RN_domain_disconnected,
                             pdu!userIds := Incl(u, pdu!userIds);
                    DECISION'encoding of pdu + 4 <= maxMCSPDUsize';
                     ('True'):  JOIN 1b;
                    ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        DECISION pdu!userIds = Empty;
         (False):   JOIN 3f;
        ELSE:ENDDECISION;
         2b :       /* for u in uRevoke */
        DECISION uRevoke = Empty;
         (False):   CALL     Allocate_buffer(b);
                    DECISION b = 0;
                     (True):   RETURN;
                    ELSE:ENDDECISION;
                    TASK     u := Pick(uRevoke),
                             uRevoke := Del(u, uRevoke),
                             pdu!reason := RN_channel_purged,
                             pdu!userIds := Incl(u, pdu!userIds);
                    DECISION'encoding of pdu + 4 <= maxMCSPDUsize';
                     ('True'):  JOIN 1b;
                    ELSE:ENDDECISION;
        ELSE:ENDDECISION;
         3f :
        DECISION pdu!userIds = Empty;
         (False):   TASK     buffer(b)!pdu := pdu,
                             pdu!userIds := Empty;
                    DECISION upward = 0;
                     (False):  CALL     Cast_buffer(b, upward);
                     (True):   TASK     buffer(b)!pdu!kind := DUin;
                               CALL     Apply_PDU(0, b, diagnostic);
                    ENDDECISION;
                    JOIN 1b;
        ELSE:ENDDECISION;
        CALL        Release_buffer(b);
        RETURN;
        ENDPROCEDURE;
```

```
                                                                    /*--------------------------*/
PROCEDURE       Leave_channels;                                             / Leave_channels  */
                                                                    /*--------------------------*/
        DCL     b               BufferId,
                c               ChannelId,
                pdu             PDUStruct;
        START
        COMMENT 'Generate CLrq MCSPDUs to leave channels.
                Fill them to maximum size that encoding allows.
                ';
        TASK    b := 0,
                pdu!kind := CLrq,
                pdu!channelIds := Empty;
         1b :   /* for c in cLeave */
        DECISION cLeave = Empty;
         (False):   CALL    Allocate_buffer(b);
                    DECISION b = 0;
                     (True):    RETURN;
                    ELSE:ENDDECISION;
                    TASK    c := Pick(cLeave),
                            cLeave := Del(c, cLeave);
                    DECISION chan(c)!kind;
                     (Static, Assigned):
                            CALL        Delete_channel(c);
                    ELSE:ENDDECISION;
                    TASK    pdu!channelIds := Incl(c, pdu!channelIds);
                    DECISION'encoding of pdu + 4 <= maxMCSPDUsize';
                     ('True'):  JOIN 1b;
                    ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        DECISION pdu!channelIds = Empty;
         (False):   TASK    buffer(b)!pdu := pdu,
                            pdu!channelIds := Empty;
                    CALL    Cast_buffer(b, upward);
                    JOIN 1b;
        ELSE:ENDDECISION;
        CALL        Release_buffer(b);
        RETURN;
        ENDPROCEDURE;
```

```
                                                                    /*----------------------------*/
PROCEDURE       Disband_channels;                                   /*  Disband_channels  */
                                                                    /*----------------------------*/
        DCL     b               BufferId,
                c               ChannelId,
                pdu             PDUStruct,
                diagnostic      Diagnostic;
        START
        COMMENT 'Generate CDrq MCSPDUs to disband channels.
                ';
        TASK    b := 0,
                pdu!kind := CDrq;
         1b :   /* for c in cDisband */
        DECISION cDisband = Empty;
         (False):   CALL    Allocate_buffer(b);
                    DECISION b = 0;
                     (True):    RETURN;
                    ELSE:ENDDECISION;
                    TASK    c := Pick(cDisband),
                            cDisband := Del(c, cDisband),
                            pdu!channelId := c,
                            buffer(b)!pdu := pdu;
```

```
                DECISION upward = 0;
                 (False):  CALL      Cast_buffer(b, upward);
                 (True):   TASK      buffer(b)!pdu!kind := CDin;
                           CALL      Apply_PDU(0, b, diagnostic);
                ENDDECISION;
                JOIN 1b;
        ELSE:ENDDECISION;
        CALL    Release_buffer(b);
        RETURN;
        ENDPROCEDURE;


                                                                  /*------------------------*/
PROCEDURE       Reject_tokens;                                    /*  Reject_tokens  */
                                                                  /*------------------------*/
        DCL     b               BufferId,
                t               TokenId,
                u               UserId,
                p               PortalId,
                pdu             PDUStruct;
        START
        COMMENT 'Generate TVcf MCSPDUs to reject tokens.
                ';
        TASK    b := 0,
                pdu!kind := TVcf,
                pdu!result := RT_no_such_user;
         1b :   /* for t in tReject */
        DECISION tReject = Empty;
         (False):   CALL    Allocate_buffer(b);
                DECISION b = 0;
                 (True):    RETURN;
                ELSE:ENDDECISION;
                TASK    t := Pick(tReject),
                        tReject := Del(t, tReject),
                        token(t)!kind := Grabbed,
                        u := token(t)!grabber,
                        pdu!initiator := u,
                        pdu!tokenId := t;
                CALL    Token_status(pdu);
                TASK    buffer(b)!pdu := pdu;
                CALL    Route_user(u, p);
                CALL    Cast_buffer(b, p);
                JOIN 1b;
        ELSE:ENDDECISION;
        CALL    Release_buffer(b);
        RETURN;
        ENDPROCEDURE;


                                                                  /*------------------------*/
PROCEDURE       Process_PDU;                                      /*  Process_PDU  */
FPAR            r               PortalId,                         /*------------------------*/
                dp              DataPriority;


        DCL     b               BufferId,
                pdu             PDUStruct,
                diagnostic      Diagnostic;
        START
        COMMENT 'This is the main line of processing an MCSPDU.
                If its content is invalid, ignore or reject it.
                Else if this is the top provider, take the special
                actions of Top_provider.  Then whether or not this
                is the top provider, call Apply_PDU.
                ';
        TASK    b := portal(r)!inBuffer(dp),
                pdu := buffer(b)!pdu;
```

```
DECISION pdu!kind = RJum;
 (True):        TASK      diagnostic := pdu!diagnostic;
 (False):       CALL      Validate_input(r, b, diagnostic);
                DECISION diagnostic = DC_OK;
                 (True):   DECISION buffer(b)!pdu!kind;
                            (MCrq):     TASK     mcrqQueue := Push(r, mcrqQueue);
                            (MTrq):     TASK     mtrqQueue := Push(r, mtrqQueue);
                            (AUrq):     TASK     aurqQueue := Push(r, aurqQueue);
                           ELSE:ENDDECISION;
                           DECISION upward = 0;
                            (True):     CALL     Top_provider(r, b);
                           ELSE:ENDDECISION;
                           CALL        Apply_PDU(r, b, diagnostic);
                ELSE:ENDDECISION;
ENDDECISION;
DECISION diagnostic;
 (DC_OK, DC_ignore):
                /* no action */
ELSE:
                OUTPUT  Report.portal(r, diagnostic) TO control;
                DECISION pdu!kind = RJum;
                 (False):   TASK'pdu!initialOctets := truncate pdu';
                ELSE:ENDDECISION;
                TASK      pdu!kind := RJum,
                          pdu!diagnostic := diagnostic,
                          buffer(b)!pdu := pdu,
                          dp := buffer(b)!dataPriority,
                          buffer(b)!dataPriority := 0;
                CALL      Cast_buffer(b, r);
                TASK      buffer(b)!dataPriority := dp;
ENDDECISION;
CALL            Release_buffer(b);
TASK            portal(r)!inBuffer(dp) := 0;
DECISION        portal(r)!inCredit(dp) > 0;
 (True):        TASK      pBufferWait(dp) := Incl(r, pBufferWait(dp));
ELSE:ENDDECISION;
RETURN;
ENDPROCEDURE;


                                                                  /*----------------------*/
PROCEDURE       Validate_input;                                   /*  Validate_input  */
FPAR            r               PortalId,                         /*----------------------*/
                b               BufferId,
        IN/OUT  diagnostic      Diagnostic;


        DCL     pdu             PDUStruct,
                p               PortalId,
                dp              DataPriority,
                u               UserId,
                uSet            UserIdSet,
                c               ChannelId,
                cSet            ChannelIdSet,
                cAttr           ChannelAttributes,
                cAttrSet        ChannelAttributesSet,
                t               TokenId,
                tSet            TokenIdSet,
                tAttr           TokenAttributes,
                tAttrSet        TokenAttributesSet;
        START
        COMMENT 'Validate the direction and user ids of an MCSPDU
                and perform other checks, depending on its type.
                Top_provider and Apply_PDU trust this procedure
                to catch important anomalies and ease their logic.
                ';
        TASK    pdu := buffer(b)!pdu,
                diagnostic := DC_OK;
```

```
DECISION    portal(r)!kind;
 (Downlink):
             DECISION pdu!kind;
              (EDrq, MCrq, MTrq, DPum, RJum,
               AUrq, DUrq, CJrq, CLrq, CCrq,
               CDrq, CArq, CErq, SDrq, USrq,
               TGrq, TIrq, TVrq, TVrs, TPrq,
               TRrq, TTrq):
                        /* OK */
             ELSE:
                        TASK        diagnostic := DC_forbidden_PDU_upward;
                        RETURN;
             ENDDECISION;
 (Uplink):
             DECISION pdu!kind;
              (PDin, MCcf, PCin, MTcf, PTin,
               DPum, RJum, AUcf, DUin, CJcf,
               CCcf, CDin, CAin, CEin, SDin,
               USin, TGcf, TIcf, TVin, TVcf,
               TPin, TRcf, TTcf):
                        /* OK */
             ELSE:
                        TASK        diagnostic := DC_forbidden_PDU_downward;
                        RETURN;
             ENDDECISION;
ELSE:ENDDECISION;
TASK         dp := 0;
DECISION pdu!kind;
 (SDrq, SDin, USrq, USin):
             TASK        dp := pdu!dataPriority,
                         dp := IF dp < parameters!numPriorities THEN dp
                                                ELSE parameters!numPriorities - 1 FI;
ELSE:ENDDECISION;
DECISION buffer(b)!dataPriority = dp;
 (False):    TASK        diagnostic := DC_wrong_transport_priority;
             RETURN;
ELSE:ENDDECISION;
DECISION pdu!kind;
 (MCrq):     TASK        cAttrSet := pdu!mergeChannels,
                         cSet := pdu!purgeChannelIds;
              1b :       /* for cAttr in cAttrSet */
             DECISION cAttrSet = Empty;
              (False):   TASK        cAttr := Pick(cAttrSet),
                                     cAttrSet := Del(cAttr, cAttrSet),
                                     c := cAttr!channelId;
                         DECISION c in cSet;
                          (True):    TASK     diagnostic := DC_inconsistent_merge;
                                     RETURN;
                         ELSE:ENDDECISION;
                         TASK        cSet := Incl(c, cSet);
                         DECISION cAttr!kind = Private;
                          (False):   JOIN 1b;
                         ELSE:ENDDECISION;
                         TASK        pdu!mergeChannels := Del(cAttr, pdu!mergeChannels);
                         CALL        Route_user(cAttr!manager, p);
                         DECISION p = r;
                          (False):   TASK     pdu!purgeChannelIds :=
                                                          Incl(c, pdu!purgeChannelIds);
                                     JOIN 1b;
                         ELSE:ENDDECISION;
                         TASK        uSet := cAttr!admitted;
                          2b :       /* for u in uSet */
                         DECISION uSet = Empty;
                          (False):   TASK     u := Pick(uSet),
                                              uSet := Del(u, uSet);
                                     CALL     Route_user(u, p);
```

```
                                DECISION p = r;
                                 (False): TASK     cAttr!admitted :=
                                                              Del(u, cAttr!admitted);
                                ELSE:ENDDECISION;
                                JOIN 2b;
                        ELSE:ENDDECISION;
                        TASK    pdu!mergeChannels := Incl(cAttr, pdu!mergeChannels);
                        JOIN 1b;
                ELSE:ENDDECISION;
(MTrq):         TASK    tAttrSet := pdu!mergeTokens,
                        tSet := pdu!purgeTokenIds;
                 3b :   /* for tAttr in tAttrSet */
                DECISION tAttrSet = Empty;
                 (False): TASK  tAttr := Pick(tAttrSet),
                                tAttrSet := Del(tAttr, tAttrSet),
                                t := tAttr!tokenId;
                        DECISION t in tSet;
                         (True):    TASK    diagnostic := DC_inconsistent_merge;
                                    RETURN;
                        ELSE:ENDDECISION;
                        TASK    tSet := Incl(t, tSet),
                                pdu!mergeTokens := Del(tAttr, pdu!mergeTokens);
                        DECISION tAttr!kind;
                         (Grabbed, Ungivable):
                                    CALL    Route_user(tAttr!grabber, p);
                                    DECISION p = r;
                                     (False): JOIN 4f;
                                    ELSE:ENDDECISION;
                         (Given):
                                    CALL    Route_user(tAttr!recipient, p);
                                    DECISION p = r;
                                     (False): JOIN 4f;
                                    ELSE:ENDDECISION;
                         (Giving):
                                    CALL    Route_user(tAttr!recipient, p);
                                    DECISION p = r;
                                     (True):  CALL  Route_user(tAttr!grabber, p);
                                              DECISION p = r;
                                               (False):  TASK        tAttr!kind := Given;
                                              ELSE:ENDDECISION;
                                     (False): TASK    tAttr!kind := Ungivable;
                                              CALL    Route_user(tAttr!grabber, p);
                                              DECISION p = r;
                                               (False): 4f :
                                                        TASK        pdu!purgeTokenIds :=
                                                                    Incl(t, pdu!purgeTokenIds);
                                                        JOIN 3b;
                                              ELSE:ENDDECISION;
                                    ENDDECISION;
                         (Inhibited):
                                    TASK    uSet := tAttr!inhibitors;
                                     5b :   /* for u in uSet */
                                    DECISION uSet = Empty;
                                     (False): TASK    u := Pick(uSet),
                                                      uSet := Del(u, uSet);
                                              CALL    Route_user(u, p);
                                              DECISION p = r;
                                               (False):  TASK        tAttr!inhibitors :=
                                                                     Del(u, tAttr!inhibitors);
                                              ELSE:ENDDECISION;
                                              JOIN 5b;
                                    ELSE:ENDDECISION;
                        ENDDECISION;
                        TASK    pdu!mergeTokens := Incl(tAttr, pdu!mergeTokens);
                        JOIN 3b;
                ELSE:ENDDECISION;
```

```
(DUrq):         TASK      uSet := pdu!userIds;
                 6b :     /* for u in uSet */
                DECISION uSet = Empty;
                 (False): TASK      u := Pick(uSet),
                                    uSet := Del(u, uSet);
                          CALL      Route_user(u, p);
                          DECISION p = r;
                           (False):  TASK    pdu!userIds := Del(u, pdu!userIds);
                          ELSE:ENDDECISION;
                          JOIN 6b;
                ELSE:ENDDECISION;
                DECISION pdu!userIds = Empty;
                 (True):  TASK      diagnostic := DC_ignore;
                          RETURN;
                ELSE:ENDDECISION;
(CJrq, CCrq, CDrq, CArq, CErq,
 SDrq, USrq, TGrq, TIrq, TVrq,
 TPrq, TRrq, TTrq):
                CALL      Route_user(pdu!initiator, p);
                DECISION p = r;
                 (False): TASK      diagnostic := DC_ignore;
                          RETURN;
                ELSE:ENDDECISION;
(TVin):         DECISION pdu!recipient in uUsed;
                 (False): TASK      diagnostic := DC_misrouted_user;
                          RETURN;
                ELSE:ENDDECISION;
(TVrs):         CALL      Route_user(pdu!recipient, p);
                DECISION p = r;
                 (False): TASK      diagnostic := DC_ignore;
                          RETURN;
                ELSE:ENDDECISION;
(CJcf, CCcf, TGcf, TIcf, TVcf,
 TRcf, TTcf):
                DECISION pdu!initiator in uUsed;
                 (False): TASK      diagnostic := DC_misrouted_user;
                          RETURN;
                ELSE:ENDDECISION;
ELSE:ENDDECISION;
DECISION pdu!kind;
(CDrq, CArq, CErq):
                TASK      c := pdu!channelId;
                DECISION c in cUsed and chan(c)!kind = Private
                          and pdu!initiator = chan(c)!manager;
                 (False): TASK      diagnostic := DC_ignore;
                          RETURN;
                ELSE:ENDDECISION;
(SDrq, USrq):
                TASK      c := pdu!channelId;
                DECISION c in cUsed and chan(c)!kind = Private;
                 (True):  DECISION pdu!initiator in chan(c)!admitted;
                           (False):  TASK    diagnostic := DC_ignore;
                                     RETURN;
                          ELSE:ENDDECISION;
                ELSE:ENDDECISION;
(MCcf):         DECISION not merging and mcrqQueue = EmptyQueue;
                 (True):  TASK      diagnostic := DC_unrequested_confirm;
                          RETURN;
                ELSE:ENDDECISION;
(MTcf):         DECISION not merging and mtrqQueue = EmptyQueue;
                 (True):  TASK      diagnostic := DC_unrequested_confirm;
                          RETURN;
                ELSE:ENDDECISION;
```

```
        (AUcf):     DECISION aurqQueue = EmptyQueue;
                    (True):  TASK     diagnostic := DC_unrequested_confirm;
                             RETURN;
                    ELSE:ENDDECISION;
        (CJcf, CCcf, TGcf, TIcf, TVcf,
         TRcf, TTcf):
                    DECISION merging;
                    (True):  TASK     diagnostic := DC_unrequested_confirm;
                             RETURN;
                    ELSE:ENDDECISION;
    ELSE:ENDDECISION;
    TASK        buffer(b)!pdu := pdu;
    RETURN
    COMMENT     'Additional tests might be applied to check
                that MCSPDUs flowing downward are consistent
                with user or channel states already recorded.
                But stronger assertions could contain flaws,
                and such defenses are not necessary for the
                continued functioning of this MCS provider.
                The value of an MCS domain involves trust in
                the correct operation of superior providers.
                ';
    ENDPROCEDURE;


                                                                    /*----------------------*/
PROCEDURE       Top_provider;                                       /*  Top_provider  */
FPAR            r           PortalId,                               /*----------------------*/
                b           BufferId;


    DCL         pdu         PDUStruct,
                new         PDUStruct,
                u           UserId,
                c           ChannelId,
                cAttr       ChannelAttributes,
                t           TokenId,
                tAttr       TokenAttributes,
                result      Result,
                diagnostic  Diagnostic;
    START
    COMMENT     'The buffer containing an MCSPDU will be processed next
                by Apply_PDU.  Its contents may first be modified here.
                Changing pdu!kind results in "turning the corner".
                ';
    TASK        pdu := buffer(b)!pdu;
    DECISION pdu!kind;
    (EDrq):     DECISION pdu!subHeight < parameters!maxHeight;
                (False): TASK     pdSend := True;
                ELSE:ENDDECISION;
    (MCrq):     TASK     new!kind := MCcf,
                         new!mergeChannels := Empty,
                         new!purgeChannelIds := pdu!purgeChannelIds;
                1b :     /* for cAttr in mergeChannels */
                DECISION pdu!mergeChannels = Empty;
                (False): TASK     cAttr := Pick(pdu!mergeChannels),
                                  pdu!mergeChannels := Del(cAttr, pdu!mergeChannels),
                                  c := cAttr!channelId;
                         DECISION c in cUsed;
                         (True):  DECISION chan(c)!kind;
                                  (Static):
                                           JOIN 2f;
                                  (Private):
                                           DECISION cAttr!kind = Private
                                                    and cAttr!manager = chan(c)!manager;
                                            (True):  JOIN 2f;
                                           ELSE:ENDDECISION;
                                  ELSE:ENDDECISION;
```

```
                    (False):    DECISION cAttr!kind = UserId;
                                 (True):  CALL     Take_user(c, diagnostic);
                                 (False): CALL     Take_channel(c, diagnostic);
                                ENDDECISION;
                                DECISION diagnostic = DC_OK;
                                 (True):  TASK     chan(c)!kind := cAttr!kind;
                                           2f :
                                          TASK     new!mergeChannels :=
                                                              Incl(cAttr, new!mergeChannels);
                                          JOIN 1b;
                                ELSE:ENDDECISION;
                           ENDDECISION;
                           TASK       new!purgeChannelIds := Incl(c, new!purgeChannelIds);
                           JOIN 1b;
                      ELSE:ENDDECISION;
                      TASK       pdu := new;
(MTrq):               TASK       new!kind := MTcf,
                                 new!mergeTokens := Empty,
                                 new!purgeTokenIds := pdu!purgeTokenIds;
                       3b :      /* for tAttr in mergeTokens */
                      DECISION pdu!mergeTokens = Empty;
                       (False):  TASK       tAttr := Pick(pdu!mergeTokens),
                                 pdu!mergeTokens := Del(tAttr, pdu!mergeTokens),
                                 t := tAttr!tokenId;
                           DECISION t in tUsed;
                            (True):     DECISION token(t)!kind;
                                         (Inhibited):
                                                 DECISION tAttr!kind = Inhibited;
                                                  (True):   JOIN 4f;
                                                 ELSE:ENDDECISION;
                                        ELSE:ENDDECISION;
                            (False):    CALL     Take_token(t, diagnostic);
                                        DECISION diagnostic = DC_OK;
                                         (True):   4f :
                                                 TASK     new!mergeTokens :=
                                                                     Incl(tAttr, new!mergeTokens);
                                                 JOIN 3b;
                                        ELSE:ENDDECISION;
                           ENDDECISION;
                           TASK       new!purgeTokenIds  :=  Incl(t, new!purgeTokenIds);
                           JOIN 3b;
                      ELSE:ENDDECISION;
                      TASK       pdu  :=  new;
(AUrq):               CALL       New_user(u, result);
                      TASK       pdu!kind  :=  AUcf,
                                 pdu!result  :=  result,
                                 pdu!initiator  :=  u;
(DUrq):               TASK       pdu!kind  :=  DUin;
(CJrq):               TASK       pdu!kind  :=  CJcf,
                                 c  :=  pdu!channelId,
                                 pdu!requested  :=  c,
                                 result  :=  RT_successful;
                      DECISION c = 0;
                       (True):   CALL       New_channel(c, result);
                                 DECISION result = RT_successful;
                                  (True):     TASK     chan(c)!kind  :=  Assigned;
                                 ELSE:ENDDECISION;
                                 TASK       pdu!channelId  :=  c;
                       (False):  DECISION c in cUsed;
                                  (False):    DECISION c < 1001;
                                               (False): TASK     result  :=  RT_no_such_channel;
                                               (True):  CALL     Take_chanel(c, diagnostic);
                                                        DECISION diagnostic = DC_OK;
                                                         (False):  TASK       result  :=  RT_too_many_channels;
                                                         (True):   TASK       chan(c)!kind  :=  Static;
                                                        ENDDECISION;
                                               ENDDECISION;
```

```
                (True):  DECISION chan(c)!kind;
                         (UserId):
                                  TASK    u  :=  UserId(c);
                                  DECISION pdu!initiator = u;
                                  (False):TASK    result  :=  RT_other_user_id;
                                  ELSE:ENDDECISION;
                         (Private):
                                  DECISION pdu!initiator in chan(c)!admitted;
                                  (False):TASK    result  :=  RT_not_admitted;
                                  ELSE:ENDDECISION;
                         ELSE:ENDDECISION;
                ENDDECISION;
                DECISION result = RT_successful;
                (False):TASK    pdu!channelId  :=  0;
                ELSE:ENDDECISION;
        ENDDECISION;
        TASK    pdu!result  :=  result;
(CLrq):  /* no special action */
(CCrq):  CALL    New_channel(c, result);
        TASK    pdu!kind  :=  CCcf,
                pdu!result  :=  result,
                pdu!channelId  :=  c;
(CDrq):  TASK    pdu!kind  :=  CDin;
(CArq):  TASK    pdu!kind  :=  CAin;
(CErq):  TASK    pdu!kind  :=  CEin;
(SDrq):  TASK    pdu!kind  :=  SDin;
(USrq):  TASK    pdu!kind  :=  USin;
(TGrq):  TASK    pdu!kind  :=  TGcf,
                pdu!result  :=  RT_successful,
                t  :=  pdu!tokenId,
                u  :=  pdu!initiator;
        DECISION t in tUsed;
        (False): CALL    Take_token(t, diagnostic);
                DECISION diagnostic = DC_OK;
                (False): TASK  pdu!result  :=  RT_too_many_tokens;
                (True):  TASK  token(t)!kind  :=  Grabbed,
                                token(t)!grabber  :=  u;
                ENDDECISION;
        (True):  DECISION token(t)!kind = Inhibited
                        and token(t)!inhibitors = Incl(u, Empty);
                (False):TASK   pdu!result  :=  RT_token_not_available;
                (True): TASK   token(t)!kind  :=  Grabbed,
                                token(t)!grabber  :=  u;
                ENDDECISION;
        ENDDECISION;
        CALL    Token_status(pdu);
(TIrq):  TASK    pdu!kind  :=  TIcf,
                pdu!result  :=  RT_successful,
                t  :=  pdu!tokenId,
                u  :=  pdu!initiator;
        DECISION t in tUsed;
        (False): CALL    Take_token(t, diagnostic);
                DECISION diagnostic = DC_OK;
                (False):TASK   pdu!result  :=  RT_too_many_tokens;
                (True): TASK   token(t)!kind  :=  Inhibited,
                                token(t)!inhibitors  :=  Incl(u, Empty);
                ENDDECISION;
        (True):  DECISION token(t)!kind = Grabbed and token(t)!grabber = u;
                (True): TASK   token(t)!kind  :=  Inhibited,
                                token(t)!inhibitors  :=  Incl(u, Empty);
                ELSE:ENDDECISION;
                DECISION token(t)!kind = Inhibited;
                (False):TASK    pdu!result  :=  RT_token_not_available;
                (True): TASK    token(t)!inhibitors :=
                                        Incl(u, token(t)!inhibitors);
                ENDDECISION;
```

```
                ENDDECISION;
                CALL    Token_status(pdu);
        (TVrq): TASK    pdu!kind  :=  TVcf,
                        pdu!result  :=  RT_token_not_possessed,
                        t  :=  pdu!tokenId,
                        u  :=  pdu!initiator;
                DECISION t in tUsed and token(t)!kind = Grabbed and token(t)!grabber = u;
                (True):  DECISION pdu!recipient in uUsed;
                        (False):TASK   pdu!result  :=  RT_no_such_user;
                        (True): TASK   pdu!kind  :=  TVin,
                                       token(t)!kind  :=  Giving,
                                       token(t)!recipient  :=  pdu!recipient;
                        ENDDECISION;
                ELSE:ENDDECISION;
                CALL    Token_status(pdu);
        (TPrq): TASK    pdu!kind  :=  TPin;
        (TRrq): TASK    pdu!kind  :=  TRcf,
                        pdu!result  :=  RT_token_not_possessed,
                        t  :=  pdu!tokenId,
                        u  :=  pdu!initiator;
                DECISION t in tUsed;
                (True):  DECISION token(t)!kind;
                        (Grabbed, Ungivable):
                                DECISION token(t)!grabber = u;
                                (True): TASK    pdu!result  :=  RT_successful;
                                        CALL    Delete_token(t);
                                ELSE:ENDDECISION;
                        (Giving):
                                DECISION token(t)!grabber = u;
                                (True): TASK    pdu!result  :=  RT_successful,
                                                token(t)!kind  :=  Given;
                                ELSE:ENDDECISION;
                        (Inhibited):
                                DECISION u in token(t)!inhibitors;
                                (True): TASK    pdu!result  :=  RT_successful,
                                                token(t)!inhibitors :=
                                                        Del(u, token(t)!inhibitors);
                                ELSE:ENDDECISION;
                                DECISION token(t)!inhibitors = Empty;
                                (True): CALL    Delete_token(t);
                                ELSE:ENDDECISION;
                        ENDDECISION;
                ELSE:ENDDECISION;
                CALL    Token_status(pdu);
        (TTrq): TASK    pdu!kind  :=  TTcf;
                CALL    Token_status(pdu);
        (TVrs): TASK    t  :=  pdu!tokenId,
                        u  :=  pdu!recipient;
                DECISION t in tUsed and token(t)!recipient = u;
                (True):  DECISION token(t)!kind;
                        (Giving):
                                TASK token(t)!kind  :=  Grabbed,
                                        pdu!kind  :=  TVcf,
                                        pdu!initiator  :=  token(t)!grabber;
                                DECISION pdu!result = RT_successful;
                                (True): TASK    token(t)!grabber  :=  u;
                                ELSE:ENDDECISION;
                                CALL  Token_status(pdu);
                        ELSE:ENDDECISION;
                ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        TASK    buffer(b)!pdu  :=  pdu;
        RETURN;
        ENDPROCEDURE;

                                                                /*--------------------*/
PROCEDURE       Apply_PDU;                                      /*  Apply_PDU  */
FPAR            r               PortalId,                       /*--------------------*/
                b               BufferId,
```

```
IN/OUT   diagnostic      Diagnostic;

DCL      pdu             PDUStruct,
         new             PDUStruct,
         p               PortalId,
         pSet            PortalIdSet,
         x               PortalId,
         u               UserId,
         uSet            UserIdSet,
         c               ChannelId,
         cSet            ChannelIdSet,
         cAttr           ChannelAttributes,
         cAttrSet        ChannelAttributesSet,
         t               TokenId,
         tSet            TokenIdSet,
         tAttr           TokenAttributes,
         tAttrSet        TokenAttributesSet,
         result          Result;
START
COMMENT'This is a large case selection based on MCSPDU kind.
          These actions and updates to the information base
          take place in both the top and subordinate providers.
          ';
TASK     pdu := buffer(b)!pdu;
DECISION pdu!kind;
(MCrq, MTrq, AUrq, DUrq, CCrq,
CDrq,  CArq, CErq, USrq, TGrq,
TIrq,  TVrq, TPrq, TRrq, TTrq):
         CALL   Cast_buffer(b, upward);
(PDin):  DECISION pdu!heightLimit > 0;
         (True):TASK     buffer(b)!pdu!heightLimit := pdu!heightLimit - 1;
         (False):OUTPUT Report.portal(r, DC_height_limit_exceeded) TO control;
         ENDDECISION;
         CALL   Broadcast_buffer(b);
(EDrq):  TASK   portal(r)!subHeight := pdu!subHeight,
                portal(r)!subInterval := pdu!subInterval,
                portal(r)!interval := IF pdu!subInterval = 0 THEN interval
                                      ELSE oneSecond + (pdu!subInterval * 3) FI;
         CALL   Erect_domain;
(MCcf):  DECISION merging;
         (False):TASK    p := Next(mcrqQueue),
                         mcrqQueue := Pull(mcrqQueue);
         ELSE:ENDDECISION;
         TASK   cAttrSet := pdu!mergeChannels;
         1b :   /* for cAttr in cAttrSet */
         DECISION cAttrSet = Empty;
         (False): TASK   cAttr := Pick(cAttrSet),
                         cAttrSet := Del(cAttr, cAttrSet),
                         c := cAttr!channelId;
                  DECISION c in cUsed and chan(c)!kind = cAttr!kind;
                  (False): DECISION cAttr!kind = UserId;
                           (True): CALL    Take_user(c, diagnostic);
                           (False): CALL   Take_channel(c, diagnostic);
                           ENDDECISION;
                           DECISION diagnostic = DC_OK;
                           (False):RETURN;
                           ELSE:ENDDECISION;
                           TASK chan(c)!kind := cAttr!kind;
                  ELSE:ENDDECISION;
                  DECISION merging;
                  (True):  DECISION chan(c)!kind = UserId;
                           (True): TASK    u := UserId(c),
                                           uConfirm := Incl(u, uConfirm);
                           ELSE:ENDDECISION;
```

```
            DECISION chan(c)!kind = Private
                        and  chan(c)!uMerge /= Empty;
            (True): TASK    cMerge  :=  Incl(c, cMerge);
            (False):TASK    cConfirm  :=  Incl(c, cConfirm);
            ENDDECISION;
            JOIN 1b;
ELSE:ENDDECISION;
            TASK    chan(c)!portal  :=  p;
            DECISION cAttr!kind = Static or cAttr!kind = Assigned
                    or cAttr!joined;
            (True): DECISION p = 0;
                    (False):TASK    chan(c)!joined :=
                                                Incl(p, chan(c)!joined),
                                    cLeave  :=  Del(c, cLeave);
                    (True): DECISION chan(c)!joined = Empty;
                            (True): TASK    cLeave  :=  Incl(c, cLeave);
                            ELSE:ENDDECISION;
                    ENDDECISION;
            ELSE:ENDDECISION;
            DECISION cAttr!kind = UserId and p = 0;
            (True): TASK    u  :=  UserId(c),
                            uDetach  :=  Incl(u, uDetach),
                            cLeave  :=  Del(c, cLeave);
            ELSE:ENDDECISION;
            DECISION cAttr!kind = Private;
            (False):JOIN 1b;
            ELSE:ENDDECISION;
            CALL    Route_user(cAttr!manager, x);
            DECISION x = p;
            (False):TASK    chan(c)!manager  :=  0,
                            buffer(b)!pdu!mergeChannels :=
                                    Del(cAttr, buffer(b)!pdu!mergeChannels),
                            buffer(b)!pdu!purgeChannelIds :=
                                    Incl(c, buffer(b)!pdu!purgeChannelIds),
                            cDisband  :=  Incl(c, cDisband);
            (True): TASK    chan(c)!manager  :=  cAttr!manager,
                            uSet  :=  cAttr!admitted and uUsed;
                    2b :    /* for u in uSet */
                    DECISION uSet = Empty;

                    (False):TASK    u  :=  Pick(uSet),
                                    uSet  :=  Del(u, uSet);

                            CALL    Route_user(u, x);
                            DECISION x = p;
                            (True): TASK    chan(c)!admitted :=
                                                    Incl(u, chan(c)!admitted);
                            ELSE:ENDDECISION;
                            JOIN 2b;
                    ELSE:ENDDECISION;
            ENDDECISION;
            JOIN 1b;
ELSE:ENDDECISION;
DECISION merging;
(False): CALL  Cast_buffer(b, p);
(True):  TASK  new!kind  :=  PCin,
                    new!detachUserIds  :=  Empty,
                    new!purgeChannelIds  :=  Empty,
                    cSet  :=  pdu!purgeChannelIds and cUsed;
            3b :    /* for c in cSet */
            DECISION cSet = Empty;
(False): TASK  c  :=  Pick(cSet),
                    cSet  :=  Del(c, cSet);
            DECISION chan(c)!kind;
            (UserId):
                    TASK    u  :=  UserId(c),
                            new!detachUserIds :=
                                    Incl(u, new!detachUserIds);
```

```
                    (Static, Assigned, Private):
                        TASK    new!purgeChannelIds :=
                                        Incl(c, new!purgeChannelIds);
                    ENDDECISION;
                    JOIN 3b;
                ELSE:ENDDECISION;
                CALL    Purge_users(new!detachUserIds);
                CALL    Purge_channels(new!purgeChannelIds);
                TASK    buffer(b)!pdu  :=  new;
                CALL    Broadcast_buffer(b);
            ENDDECISION;
(PCin):     TASK    cSet  :=  pdu!purgeChannelIds and cUsed,
                    buffer(b)!pdu!purgeChannelIds  :=  cSet;
            DECISION merging;
            (True):  TASK    buffer(b)!pdu!detachUserIds  :=  pdu!detachUserIds
                                                        and uConfirm,
                    buffer(b)!pdu!purgeChannelIds  :=  cSet and cConfirm;
                4b :        /* for c in cSet */
                DECISION cSet = Empty;
                (False): TASK  c  :=  Pick(cSet),
                            cSet  :=  Del(c, cSet);
                    DECISION c in cConfirm;
                    (False):TASK    chan(c)!uMerge  :=  chan(c)!admitted;
                    ELSE:ENDDECISION;
                    JOIN 4b;
                ELSE:ENDDECISION;
            ELSE:ENDDECISION;
            CALL    Purge_users(buffer(b)!pdu!detachUserIds);
            CALL    Purge_channels(buffer(b)!pdu!purgeChannelIds);
            CALL    Broadcast_buffer(b);
(MTcf):     DECISION merging;
            (False):TASK     p  :=  Next(mtrqQueue),
                    mtrqQueue  :=  Pull(mtrqQueue);
            ELSE:ENDDECISION;
            TASK    tAttrSet  :=  pdu!mergeTokens;
              5b :    /* for tAttr in tAttrSet */
            DECISION tAttrSet = Empty;
            (False): TASK    tAttr  :=  Pick(tAttrSet),
                    tAttrSet  :=  Del(tAttr, tAttrSet),
                    t  :=  tAttr!tokenId;
                DECISION t in tUsed;
                (False): CALL  Take_token(t, diagnostic);
                    DECISION diagnostic = DC_OK;
                    (False):RETURN;
                    ELSE:ENDDECISION;
                ELSE:ENDDECISION;
                TASK    token(t)!kind  :=  tAttr!kind,
                    token(t)!grabber  :=  tAttr!grabber,
                    token(t)!recipient  :=  tAttr!recipient;
                DECISION merging;
                (True):  DECISION token(t)!kind = Inhibited
                            and token(t)!uMerge /= Empty;
                    (True): TASK    tMerge  :=  Incl(t, tMerge);
                    (False):TASK    tConfirm  :=  Incl(t, tConfirm);
                        DECISION token(t)!kind = Inhibited
                                and token(t)!inhibitors = Empty;
                        (True): CALL    Delete_token(t);
                        ELSE:ENDDECISION;
                    ENDDECISION;
                    JOIN 5b;
                ELSE:ENDDECISION;
                DECISION upward = 0 and token(t)!kind = Ungivable;
                (True): TASK    tReject  :=  Incl(t, tReject);
                ELSE:ENDDECISION;
                DECISION tAttr!kind;
```

```
            (Grabbed, Ungivable):
                    CALL   Token_route(tAttr!grabber, x, p);
                    DECISION x = p;
                    (False):JOIN 6f;
                    ELSE:ENDDECISION;
            (Given):
                    CALL   Token_route(tAttr!recipient, x, p);
                    DECISION x = p;
                    (False):JOIN 6f;
                    ELSE:ENDDECISION;
            (Giving):
                    CALL    Token_route(tAttr!recipient, x, p);
                    DECISION x = p;
                    (True): CALL    Token_route(tAttr!grabber, x, p);
                            DECISION x = p;
                            (False):TASK    token(t)!kind  :=  Given;
                            ELSE:ENDDECISION;
                    (False):CALL    Token_route(tAttr!grabber, x, p);
                            DECISION x = p;
                            (True): TASK    token(t)!kind  :=  Ungivable;
                            (False): 6f :
                                    TASK    buffer(b)!pdu!mergeTokens :=
                                         Del(tAttr, buffer(b)!pdu!mergeTokens),
                                              buffer(b)!pdu!purgeTokenIds :=
                                         Incl(t, buffer(b)!pdu!purgeTokenIds);
                                    CALL    Delete_token(t);
                            ENDDECISION;
                    ENDDECISION;
            (Inhibited):
                    TASK    uSet  :=  tAttr!inhibitors and uUsed;
                      7b :    /* for u in uSet */
                    DECISION uSet = Empty;
                            (False):TASK    u  :=  Pick(uSet),
                                             uSet  :=  Del(u, uSet);
                                    CALL    Token_route(u, x, p);
                                    DECISION x = p;
                                    (True): TASK    token(t)!inhibitors :=
                                                        Incl(u, token(t)!inhibitors);
                                    ELSE:ENDDECISION;
                                    JOIN 7b;
                            ELSE:ENDDECISION;
                            DECISION token(t)!inhibitors = Empty;
                            (True): CALL    Delete_token(t);
                            ELSE:ENDDECISION;
                    ENDDECISION;
                    JOIN 5b;
            ELSE:ENDDECISION;
            DECISION merging;
            (False): CALL    Cast_buffer(b, p);
            (True):  CALL    Purge_tokens(pdu!purgeTokenIds);
                    TASK    buffer(b)!pdu!kind  :=  PTin;
                    CALL    Broadcast_buffer(b);
            ENDDECISION;
    (PTin):  TASK    tSet  :=  pdu!purgeTokenIds and tUsed,
                    buffer(b)!pdu!purgeTokenIds  :=  tSet;
            DECISION merging;
            (True):  TASK    buffer(b)!pdu!purgeTokenIds  :=  tSet and tConfirm;
                      8b :    /* for t in tSet */
                    DECISION tSet = Empty;
                    (False):TASK    t  :=  Pick(tSet),
                                     tSet  :=  Del(t, tSet);
                            DECISION t in tConfirm;
                            (False):TASK    token(t)!uMerge  :=  token(t)!inhibitors;
                            ELSE:ENDDECISION;
                            JOIN 8b;
                    ELSE:ENDDECISION;
```

```
                ELSE:ENDDECISION;
                CALL    Purge_tokens(buffer(b)!pdu!purgeTokenIds);
                CALL    Broadcast_buffer(b);
        (DPum): OUTPUT  Drop.portal(r, pdu!reason) TO control;
        (AUcf): TASK    p := Next(aurqQueue),
                        aurqQueue := Pull(aurqQueue),
                        c := ChannelId(u),
                        u := pdu!initiator;
                DECISION pdu!result = RT_successful;
                (True): DECISION upward = 0;
                        (False):CALL    Take_user(c, diagnostic);
                                DECISION diagnostic = DC_OK;
                                (False):RETURN;
                                ELSE:ENDDECISION;
                        ELSE:ENDDECISION;
                        TASK    chan(c)!portal := p;
                        DECISION p = 0;
                        (True): TASK    uDetach := Incl(u, uDetach),
                                        cLeave := Del(c, cLeave);
                        ELSE:ENDDECISION;
                ELSE:ENDDECISION;
                CALL    Cast_buffer(b, p);
        (DUin): DECISION merging;
                (True): TASK    buffer(b)!pdu!userIds := pdu!userIds and uConfirm;
                ELSE:ENDDECISION;
                CALL    Purge_users(buffer(b)!pdu!userIds);
                CALL    Broadcast_buffer(b);
        (CJrq): TASK    c := pdu!channelId,
                        result := RT_successful,
                        p := upward;
                DECISION c in cUsed;
                (True): DECISION chan(c)!kind;
                        (UserId):
                                TASK    u := UserId(c);
                                DECISION pdu!initiator = u;
                                (False):TASK    result := RT_other_user_id;
                                ELSE:ENDDECISION;
                        (Private):
                                DECISION pdu!initiator in chan(c)!admitted;
                                (False):TASK    result := RT_not_admitted;
                                ELSE:ENDDECISION;
                        ELSE:ENDDECISION;
                        DECISION result = RT_successful;
                        (False):TASK    buffer(b)!pdu!kind := CJcf,
                                buffer(b)!pdu!requested := c,
                                buffer(b)!pdu!result := result,
                                buffer(b)!pdu!channelId := 0,
                                p := r;
                (True): DECISION chan(c)!joined /= Empty or c in cLeave;
                        (True): TASK    chan(c)!joined :=
                                                Incl(r, chan(c)!joined),
                                        cLeave := Del(c, cLeave),
                                        buffer(b)!pdu!kind := CJcf,
                                        buffer(b)!pdu!requested := c,
                                        buffer(b)!pdu!result := result,
                                        p := r;
                        ELSE:ENDDECISION;
                ENDDECISION;
        ELSE:ENDDECISION;
        CALL    Cast_buffer(b, p);
(CJcf): TASK    c := pdu!channelId,
                u := pdu!initiator;
        CALL    Route_user(u, p);
        DECISION pdu!result = RT_successful;
        (True): DECISION c in cUsed;
                (False): CALL   Take_channel(c, diagnostic);
```

```
                    DECISION diagnostic = DC_OK;
                    (False):RETURN;
                    ELSE:ENDDECISION;
                    TASK   chan(c)!kind  :=  IF c < 1001 THEN Static
                                             ELSE Assigned FI;
               ELSE:ENDDECISION;
               DECISION p = 0;
               (False): TASK   chan(c)!joined  :=  Incl(p, chan(c)!joined),
                               cLeave  :=  Del(c, cLeave);
               (True):  DECISION chan(c)!joined = Empty;
                        (True): TASK   cLeave  :=  Incl(c, cLeave);
                        ELSE:ENDDECISION;
               ENDDECISION;
          ELSE:ENDDECISION;
          CALL   Cast_buffer(b, p);
(CLrq):   TASK   cSet  :=  pdu!channelIds and cUsed;
            9b :   /* for c in cSet */
          DECISION cSet = Empty;
          (False): TASK   c  :=  Pick(cSet),
                          cSet  :=  Del(c, cSet);
                   DECISION r in chan(c)!joined;
                   (True):  TASK   chan(c)!joined  :=  Del(r, chan(c)!joined);
                            DECISION chan(c)!joined = Empty;
                            (True): TASK   cLeave  :=  Incl(c, cLeave);
                            ELSE:ENDDECISION;
                   ELSE:ENDDECISION;
                   JOIN 9b;
          ELSE:ENDDECISION;
(CCcf):   TASK   c  :=  pdu!channelId,
                 u  :=  pdu!initiator;
          DECISION pdu!result = RT_successful;
          (True):  DECISION upward = 0;
                   (False): CALL   Take_channel(c, diagnostic);
                            DECISION diagnostic = DC_OK;
                            (False):RETURN;
                            ELSE:ENDDECISION;
                   ELSE:ENDDECISION;
                   TASK   chan(c)!kind  :=  Private,
                            chan(c)!manager  :=  u,
                            chan(c)!admitted  :=  Incl(u, Empty);
          ELSE:ENDDECISION;
          CALL   Route_user(u, p);
          CALL   Cast_buffer(b, p);
(CDin):   TASK   c  :=  pdu!channelId;
          DECISION c in cUsed;
          (False): RETURN;
          (True):  DECISION merging and not c in cConfirm;
                   (True):  TASK   chan(c)!uMerge  :=  chan(c)!admitted;
                            RETURN;
                   ELSE:ENDDECISION;
                   DECISION chan(c)!kind = Private;
                   (False): TASK   diagnostic  :=  DC_channel_id_conflict;
                            RETURN;
                   ELSE:ENDDECISION;
          ENDDECISION;
          TASK   uSet  :=  Incl(chan(c)!manager, chan(c)!admitted);
          CALL   Delete_channel(c);
          CALL   Multicast_buffer(b, uSet);
(CAin):   TASK   c  :=  pdu!channelId,
                 uSet  :=  pdu!userIds and uUsed,
                 buffer(b)!pdu!userIds  :=  uSet;
          DECISION merging;
          (True):  TASK    uSet  :=  uSet and uConfirm,
                           buffer(b)!pdu!userIds  :=  uSet;
                    10b :   /* for u in uSet */
                   DECISION uSet = Empty;
```

```
                (False): TASK   u  :=  Pick(uSet),
                                uSet  :=  Del(u, uSet),
                                c  :=  ChannelId(u),
                                chan(c)!portal = 0;
                         JOIN 10b;
                ELSE:ENDDECISION;
                TASK    buffer(b)!pdu!kind  :=  DUrq,
                        buffer(b)!pdu!reason  :=  RN_channel_purged;
                CALL    Cast_buffer(b, r);
                RETURN;
        ELSE:ENDDECISION;
        DECISION uSet = Empty;
        (False): DECISION c in cUsed and chan(c)!kind = Private;
                (False): CALL   Take_channel(c, diagnostic);
                         DECISION diagnostic = DC_OK;
                         (False):RETURN;
                         ELSE:ENDDECISION;
                ELSE:ENDDECISION;
                TASK    chan(c)!kind  :=  Private,
                        chan(c)!manager  :=  pdu!initiator,
                        chan(c)!admitted  :=  chan(c)!admitted or uSet;
                CALL    Multicast_buffer(b, uSet);
        ELSE:ENDDECISION;
(CEin):  TASK   c  :=  pdu!channelId;
        DECISION c in cUsed;
        (False): RETURN;
        (True):  DECISION merging and not c in cConfirm;
                (True):  TASK   chan(c)!uMerge  :=  chan(c)!admitted;
                         RETURN;
                ELSE:ENDDECISION;
                DECISION chan(c)!kind = Private;
                (False): TASK   diagnostic  :=  DC_channel_id_conflict;
                         RETURN;
                ELSE:ENDDECISION;
        ENDDECISION;
        TASK    uSet  :=  chan(c)!admitted,
                pSet  :=  Empty;
         11b :   /* for u in uSet */
        DECISION uSet = Empty;
        (False): TASK    u  :=  Pick(uSet),
                         uSet  :=  Del(u, uSet);
                DECISION u in pdu!userIds;
                (False): CALL   Route_user(u, p);
                         TASK   pSet  :=  Incl(p, pSet);
                ELSE:ENDDECISION;
                JOIN 11b;
        ELSE:ENDDECISION;
        TASK    uSet  :=  pdu!userIds and chan(c)!admitted,
                buffer(b)!pdu!userIds  :=  uSet;
         12b :    /* for u in uSet */
        DECISION uSet = Empty;
        (False): TASK    u  :=  Pick(uSet),
                         uSet  :=  Del(u, uSet),
                         chan(c)!admitted  :=  Del(u, chan(c)!admitted);
                CALL    Route_user(u, p);
                DECISION p in chan(c)!joined and not p in pSet;
                (False): TASK   chan(c)!joined  :=  Del(p, chan(c)!joined);
                         DECISION chan(c)!joined = Empty;
                         (True): TASK   cLeave  :=  Incl(c, cLeave);
                         ELSE:ENDDECISION;
                ELSE:ENDDECISION;
                JOIN 12b;
        ELSE:ENDDECISION;
        DECISION chan(c)!admitted /= Empty or chan(c)!manager in uUsed;
        (False): CALL   Delete_channel(c);
        ELSE:ENDDECISION;
```

```
            TASK    uSet  :=  buffer(b)!pdu!userIds;
            CALL    Multicast_buffer(b, uSet);
(SDrq):     TASK     buffer(b)!pdu!kind  :=  SDin,
                     pSet  :=  Incl(upward, Empty);
            JOIN 13f;
(SDin):     TASK    pSet  :=  Empty;
            JOIN 13f;
(USin):     TASK    pSet  :=  Empty;
              13f :
            TASK    c  :=  pdu!channelId;
            DECISION c in cUsed and not merging;
            (True):  TASK    pSet  :=  pSet or chan(c)!joined;
                     DECISION chan(c)!kind = UserId;
                     (True):  TASK    pSet  :=  Del(upward, pSet);
                     ELSE:ENDDECISION;
            ELSE:ENDDECISION;
            DECISION buffer(b)!pdu!kind = SDin;
            (True):  TASK    pSet  :=  Del(r, pSet);
            ELSE:ENDDECISION;
              14b :    /* for p in pSet */
            DECISION pSet = Empty;
            (False): TASK     p  :=  Pick(pSet),
                              pSet  :=  Del(p, pSet);
                     CALL     Cast_buffer(b, p);
                     JOIN 14b;
            ELSE:ENDDECISION;
(TGcf, Tlcf, TVcf, TRcf, TTcf):
            TASK    t  :=  pdu!tokenId,
                    u  :=  pdu!initiator;
            DECISION pdu!tokenStatus;
            (SelfGrabbed):
                     DECISION t in tUsed;
                     (False): CALL    Take_token(t, diagnostic);
                              DECISION diagnostic = DC_OK;
                              (False):RETURN;
                              ELSE:ENDDECISION;
                     ELSE:ENDDECISION;
                     TASK    token(t)!kind  :=  Grabbed,
                              token(t)!grabber  :=  u;
            (SelfInhibited):
                     DECISION t in tUsed;
                     (False): CALL    Take_token(t, diagnostic);
                              DECISION diagnostic = DC_OK;
                              (False):RETURN;
                              ELSE:ENDDECISION;
                     ELSE:ENDDECISION;
                     TASK    token(t)!kind  :=  Inhibited,
                              token(t)!inhibitors  :=  Incl(u, token(t)!inhibitors);
            (SelfRecipient):
                     TASK    token(t)!recipient  :=  u;
            (SelfGiving):
                     TASK    token(t)!grabber  :=  u;
            (NotInUse):
                     CALL    Delete_token(t);
            ELSE:
                     DECISION token(t)!kind;
                     (Grabbed, Ungivable):
                              DECISION token(t)!grabber = u;
                              (True): CALL    Delete_token(t);
                              ELSE:ENDDECISION;
                     (Giving, Given):
                              DECISION token(t)!grabber = u;
                              (True): TASK    token(t)!kind  :=  Given;
                              ELSE:ENDDECISION;
                              DECISION token(t)!recipient = u;
                              (True): CALL    Delete_token(t);
```

```
                    ELSE:ENDDECISION;
               (Inhibited):
                    TASK    token(t)!inhibitors :=
                                   Del(u, token(t)!inhibitors);
                    DECISION token(t)!inhibitors = Empty;
                    (True): CALL    Delete_token(t);
                    ELSE:ENDDECISION;
               ENDDECISION;
          ENDDECISION;
          CALL    Route_user(u, p);
          CALL    Cast_buffer(b, p);
(TVin):   TASK    t  :=  pdu!tokenId,
               u  :=  pdu!recipient;
          DECISION merging;
          (True):  TASK    buffer(b)!pdu!kind  :=  TVrs,
                    buffer(b)!pdu!result  :=  RT_domain_merging;
               CALL    Cast_buffer(b, r);
               RETURN;
          ELSE:ENDDECISION;
          DECISION t in tUsed;
          (False): CALL    Take_token(t, diagnostic);
               DECISION diagnostic = DC_OK;
               (False):RETURN;
               ELSE:ENDDECISION;
          ELSE:ENDDECISION;
          TASK    token(t)!kind  :=  Giving,
               token(t)!grabber  :=  pdu!initiator,
               token(t)!recipient  :=  u;
          CALL    Route_user(u, p);
          CALL    Cast_buffer(b, p);
(TVrs):   TASK    t  :=  pdu!tokenId,
               u  :=  pdu!recipient;
          DECISION t in tUsed and token(t)!recipient = u;
          (True):  DECISION token(t)!kind;
               (Giving):
                    TASK    token(t)!kind  :=  Grabbed;
                    DECISION pdu!result = RT_successful;
                    (True): TASK    token(t)!grabber  :=  u;
                    (False):DECISION token(t)!grabber in uUsed;
                         (False):CALL    Delete_token(t);
                         ELSE:ENDDECISION;
                    ENDDECISION;
                    CALL    Cast_buffer(b, upward);
               (Given):
                    TASK    token(t)!kind  :=  Grabbed,
                              token(t)!grabber  :=  u;
                    DECISION pdu!result = RT_successful;
                    (False):CALL    Delete_token(t);
                    ELSE:ENDDECISION;
                    CALL    Cast_buffer(b, upward);
               ELSE:ENDDECISION;
          ELSE:ENDDECISION;
(TPin):   TASK    t  :=  pdu!tokenId;
          DECISION t in tUsed and not merging;
          (True):  DECISION token(t)!kind;
               (Grabbed, Ungivable):
                    CALL    Route_user(token(t)!grabber, p);
                    CALL    Cast_buffer(b, p);
               (Given):
                    CALL    Route_user(token(t)!recipient, p);
                    CALL    Cast_buffer(b, p);
               (Giving):
                    TASK    uSet  :=  Incl(token(t)!grabber, Empty);
                    TASK    uSet  :=  Incl(token(t)!recipient, uSet);
                    CALL    Multicast_buffer(b, uSet);
```

```
                    (Inhibited):
                        TASK    uSet  :=  token(t)!inhibitors;
                        CALL    Multicast_buffer(b, uSet);
                    ENDDECISION;
                ELSE:ENDDECISION;
        ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                                    /*----------------------*/
PROCEDURE       Token_status;                                       /*  Token_status  */
FPAR    IN/OUT  pdu             PDUStruct;                          /*----------------------*/

        DCL     t       TokenId,
                u       UserId,
                status  TokenStatus;
        START
        COMMENT'Calculate for an MCSPDU the relationship between
                 the initiator user id and token id it contains.
                ';
        TASK    t  :=  pdu!tokenId,
                u  :=  pdu!initiator;
        DECISION t in tUsed;
        (False):TASK    status  :=  NotInUse;
        (True): DECISION token(t)!kind;
                (Grabbed):
                        DECISION token(t)!grabber = u;
                        (True): TASK    status  :=  SelfGrabbed;
                        (False):TASK    status  :=  OtherGrabbed;
                        ENDDECISION;
                (Ungivable):
                        DECISION token(t)!grabber = u;
                        (True): TASK    status  :=  SelfGiving;
                        (False):TASK    status  :=  OtherGiving;
                        ENDDECISION;
                (Given):
                        DECISION token(t)!recipient = u;
                        (True): TASK    status  :=  SelfRecipient;
                        (False):TASK    status  :=  OtherGiving;
                        ENDDECISION;
                (Giving):
                        DECISION token(t)!recipient = u;
                        (True): TASK    status  :=  SelfRecipient;
                        (False):DECISION token(t)!grabber = u;
                                (True): TASK    status  :=  SelfGiving;
                                (False):TASK    status  :=  OtherGiving;
                                ENDDECISION;
                        ENDDECISION;
                (Inhibited):
                        DECISION u in token(t)!inhibitors;
                        (True): TASK    status  :=  SelfInhibited;
                        (False):TASK    status  :=  OtherInhibited;
                        ENDDECISION;
                ENDDECISION;
        ENDDECISION;
        TASK    pdu!tokenStatus  :=  status;
        RETURN;
        ENDPROCEDURE;

                                                                    /*----------------------*/
PROCEDURE       Token_route;                                        /*  Token_route  */
FPAR            u               UserId,                             /*----------------------*/
        IN/OUT  x               PortalId,
                p               PortalId;

        START
        COMMENT'Revoke token user if its route x is no longer via p.
                ';
        CALL    Route_user(u, x);
```

```
        DECISION x = p;
        (False):  DECISION u in uUsed;
                  (True): TASK   uRevoke  :=  Incl(u, uRevoke);
                  ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;
                                                                    /*---------------------*/
PROCEDURE       Delete_user;                                        /*  Delete_user  */
FPAR            u               UserId;                             /*---------------------*/

        DCL     p               PortalId,
                c               ChannelId,
                cSet            ChannelIdSet,
                a               UserId,
                aSet            UserIdSet,
                q               PortalId,
                t               TokenId,
                tSet            TokenIdSet;
        START
        COMMENT'Update the information base to delete a user id.
                 This has consequences for its channels and tokens.
                 Any data still in transit is left undisturbed.
                 ';
        DECISION u in uUsed;
        (False):RETURN;
        ELSE:ENDDECISION;
        TASK    c  :=  ChannelId(u),
                p  :=  chan(c)!portal,
                uUsed  :=  Del(u, uUsed),
                numUserIds  :=  numUserIds - 1,
                cUsed  :=  Del(c, cUsed),
                cFree  :=  Incl(c, cFree),
                numChannelIds  :=  numChannelIds - 1,
                uConfirm  :=  Del(u, uConfirm),
                cConfirm  :=  Del(c, cConfirm),
                uDetach  :=  Del(u, uDetach),
                uRevoke  :=  Del(u, uRevoke),
                cLeave  :=  Del(c, cLeave);
        TASK    cSet  :=  cUsed;
          1b :  /* for c in cSet */
        DECISION cSet = Empty;
        (False):  TASK   c  :=  Pick(cSet),
                         cSet  :=  Del(c, cSet);
                  DECISION portal(p)!kind = Attached and p in chan(c)!joined;
                  (True):  TASK   chan(c)!joined  :=  Del(p, chan(c)!joined);
                           DECISION chan(c)!joined = Empty;
                           (True): TASK   cLeave  :=  Incl(c, cLeave);
                           ELSE:ENDDECISION;
                  ELSE:ENDDECISION;
                  DECISION chan(c)!kind = Private;
                  (True):  DECISION chan(c)!manager = u;
                           (True):  TASK   chan(c)!manager  :=  0;
                                    DECISION upward = 0;
                                    (True): TASK   cDisband  :=  Incl(c, cDisband);
                                    ELSE:ENDDECISION;
                           ELSE:ENDDECISION;
                           TASK    chan(c)!admitted  :=  Del(u, chan(c)!admitted),
                                   chan(c)!uMerge  :=  Del(u, chan(c)!uMerge);
                           DECISION chan(c)!admitted /= Empty or chan(c)!manager in uUsed;
                           (False): DECISION not merging or c in cConfirm;
                                    (True): CALL   Delete_channel(c);
                                    ELSE:ENDDECISION;
                           (True):  DECISION p in chan(c)!joined;
                                    (True): TASK   aSet  :=  chan(c)!admitted;
                                              2b :  /* for a in aSet */
```

```
                                        DECISION aSet = Empty;
                                        (False):TASK    a  :=  Pick(aSet),
                                                        aSet  :=  Del(a, aSet);
                                                CALL    Route_user(a, q);
                                                DECISION q = p;
                                                (False):JOIN 2b;
                                                (True): JOIN 1b;
                                                ENDDECISION;
                                        ELSE:ENDDECISION;
                                        TASK    chan(c)!joined  :=  Del(p, chan(c)!joined);
                                        DECISION chan(c)!joined = Empty;
                                        (True): TASK    cLeave  :=  Incl(c, cLeave);
                                        ELSE:ENDDECISION;
                                ELSE:ENDDECISION;
                        ENDDECISION;
                ELSE:ENDDECISION;
                JOIN 1b;
        ELSE:ENDDECISION;
        TASK    tSet  :=  tUsed;
          3b :  /* for t in tSet */
        DECISION tSet = Empty;
        (False):  TASK    t  :=  Pick(tSet),
                        tSet  :=  Del(t, tSet);
                DECISION token(t)!kind;
                (Grabbed, Ungivable):
                        DECISION token(t)!grabber = u;
                        (True): JOIN 3f;
                        ELSE:ENDDECISION;
                (Given):
                        DECISION token(t)!recipient = u;
                        (True): JOIN 3f;
                        ELSE:ENDDECISION;
                (Giving):
                        DECISION token(t)!recipient = u;
                        (True):  TASK    token(t)!kind  :=  Ungivable;
                                DECISION token(t)!grabber in uUsed;
                                (False):JOIN 3f;
                                (True): DECISION upward = 0;
                                        (True): TASK    tReject  :=  Incl(t, tReject);
                                        ELSE:ENDDECISION;
                                ENDDECISION;
                        (False): DECISION token(t)!grabber = u;
                                (True): TASK    token(t)!kind  :=  Given;
                                ELSE:ENDDECISION;
                        ELSE:ENDDECISION;
                (Inhibited):
                        TASK    token(t)!inhibitors  :=  Del(u, token(t)!inhibitors),
                                token(t)!uMerge  :=  Del(u, token(t)!uMerge);
                        DECISION token(t)!inhibitors = Empty;
                        (True):   3f :
                                DECISION not merging or t in tConfirm;
                                (True): CALL    Delete_token(t);
                                (False):TASK    token(t)!kind  :=  Inhibited,
                                                token(t)!inhibitors  :=  Empty;
                                ENDDECISION;
                        ELSE:ENDDECISION;
                ENDDECISION;
                JOIN 3b;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;
                                                                        /*-------------------------*/
PROCEDURE       Delete_channel;                                         /*  Delete_channel  */
FPAR            c               ChannelId;                              /*-------------------------*/
```

```
        DCL     u               UserId;
        START
        COMMENT'Update the information base to delete a channel id.
                ';
        DECISION c in cUsed;
        (False):RETURN;
        ELSE:ENDDECISION;
        DECISION chan(c)!kind = UserId;
        (True):  TASK   u := UserId(c);
                 CALL   Delete_user(u);
        (False): TASK   cUsed := Del(c, cUsed),
                        cFree := Incl(c, cFree),
                        numChannelIds := numChannelIds - 1,
                        cConfirm := Del(c, cConfirm),
                        cLeave := Del(c, cLeave),
                        cDisband := Del(c, cDisband);
        ENDDECISION;
        RETURN;
        ENDPROCEDURE;
                                                                /*----------------------*/
PROCEDURE       Delete_token;                                   /*  Delete_token  */
FPAR            t               TokenId;                        /*----------------------*/

        START
        COMMENT'Update the information base to delete a token id.
                ';
        DECISION t in tUsed;
        (False):RETURN;
        ELSE:ENDDECISION;
        TASK    tUsed := Del(t, tUsed),
                tFree := Incl(t, tFree),
                numTokenIds := numTokenIds - 1,
                tConfirm := Del(t, tConfirm),
                tReject := Del(t, tReject);
        RETURN;
        ENDPROCEDURE;
                                                                /*----------------------*/
PROCEDURE       Purge_users;                                    /*  Purge_users  */
FPAR            uSet            UserIdSet;                      /*----------------------*/

        DCL     u               UserId;
        START
        COMMENT'Delete a set of user ids.
                ';
          1b :   /* for u in uSet */
        DECISION uSet = Empty;
        (False): TASK   u := Pick(uSet),
                        uSet := Del(u, uSet);
                 CALL   Delete_user(u);
                 JOIN 1b;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;
                                                                /*---------------------------*/
PROCEDURE       Purge_channels;                                         /*Purge_channels  */
FPAR            cSet            ChannelIdSet;                   /*---------------------------*/

        DCL     c               ChannelId;
        START
        COMMENT'Delete a set of channel ids.
                ';
          1b :   /* for c in cSet */
        DECISION cSet = Empty;
        (False): TASK   c := Pick(cSet),
                        cSet := Del(c, cSet);
                 CALL   Delete_channel(c);
```

```
                JOIN 1b;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;
                                                                    /*----------------------*/
PROCEDURE       Purge_tokens;                                       /*  Purge_tokens  */
FPAR            tSet            TokenIdSet;                         /*----------------------*/

        DCL     t               TokenId;
        START
        COMMENT'Delete a set of token ids.
                ';
          1b :      /* for t in tSet */
        DECISION tSet = Empty;
        (False):  TASK    t  :=  Pick(tSet),
                          tSet  :=  Del(t, tSet);
                  CALL    Delete_token(t);
                  JOIN 1b;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;
                                                                    /*----------------------*/
PROCEDURE       Output_buffer;                                      /*  Output_buffer  */
FPAR            b               BufferId,                           /*----------------------*/
                p               PortalId;

        DCL     dp              DataPriority,
                pdu             PDUStruct,
                pid             PId;
        START
        COMMENT'Send the output signal representing an MCSPDU.
                 The next must wait until PDU.ready is received.
                ';
        TASK    dp  :=  buffer(b)!dataPriority,
                pdu  :=  buffer(b)!pdu,
                pid  :=  portal(p)!pids(dp);
        DECISION p = upward and pdu!kind = SDin;
        (True):   TASK    pdu!kind  :=  SDrq;
        ELSE:ENDDECISION;
        DECISION pdu!kind;


        (PDin)   : OUTPUT  PDin(pdu)  TO pid;
        (EDrq)   : OUTPUT  EDrq(pdu)  TO pid;
        (MCrq)   : OUTPUT  MCrq(pdu)  TO pid;
        (MCcf)   : OUTPUT  MCcf(pdu)  TO pid;
        (PCin)   : OUTPUT  PCin(pdu)  TO pid;
        (MTrq)   : OUTPUT  MTrq(pdu)  TO pid;
        (MTcf)   : OUTPUT  MTcf(pdu)  TO pid;
        (PTin)   : OUTPUT  PTin(pdu)  TO pid;
        (DPum)   : OUTPUT  DPum(pdu)  TO pid;
        (RJum)   : OUTPUT  RJum(pdu)  TO pid;
        (AUrq)   : OUTPUT  AUrq(pdu)  TO pid;
        (AUcf)   : OUTPUT  AUcf(pdu)  TO pid;
        (DUrq)   : OUTPUT  DUrq(pdu)  TO pid;
        (DUin)   : OUTPUT  DUin(pdu)  TO pid;
        (CJrq)   : OUTPUT  CJrq(pdu)  TO pid;
        (CJcf)   : OUTPUT  CJcf(pdu)  TO pid;
        (CLrq)   : OUTPUT  CLrq(pdu)  TO pid;
        (CCrq)   : OUTPUT  CCrq(pdu)  TO pid;
        (CCcf)   : OUTPUT  CCcf(pdu)  TO pid;
        (CDrq)   : OUTPUT  CDrq(pdu)  TO pid;
        (CDin)   : OUTPUT  CDin(pdu)  TO pid;
        (CArq)   : OUTPUT  CArq(pdu)  TO pid;
        (CAin)   : OUTPUT  CAin(pdu)  TO pid;
        (CErq)   : OUTPUT  CErq(pdu)  TO pid;
        (CEin)   : OUTPUT  CEin(pdu)  TO pid;
        (SDrq)   : OUTPUT  SDrq(pdu)  TO pid;
```

```
        (SDin)   : OUTPUT  SDin(pdu)  TO pid;
        (USrq)   : OUTPUT  USrq(pdu)  TO pid;
        (USin)   : OUTPUT  USin(pdu)  TO pid;
        (TGrq)   : OUTPUT  TGrq(pdu)  TO pid;
        (TGcf)   : OUTPUT  TGcf(pdu)  TO pid;
        (TIrq)   : OUTPUT  TIrq(pdu)  TO pid;
        (TIcf)   : OUTPUT  TIcf(pdu)  TO pid;
        (TVrq)   : OUTPUT  TVrq(pdu)  TO pid;
        (TVin)   : OUTPUT  TVin(pdu)  TO pid;
        (TVrs)   : OUTPUT  TVrs(pdu)  TO pid;
        (TVcf)   : OUTPUT  TVcf(pdu)  TO pid;
        (TPrq)   : OUTPUT  TPrq(pdu)  TO pid;
        (TPin)   : OUTPUT  TPin(pdu)  TO pid;
        (TRrq)   : OUTPUT  TRrq(pdu)  TO pid;
        (TRcf)   : OUTPUT  TRcf(pdu)  TO pid;
        (TTrq)   : OUTPUT  TTrq(pdu)  TO pid;
        (TTcf)   : OUTPUT  TTcf(pdu)  TO pid;
        ENDDECISION;
        RETURN;
        ENDPROCEDURE;

        /* Input transitions */

        DCL       p        PortalId,
                  dp       DataPriority,
                  pdu      PDUStruct,
                  pKind    PortalKind,
                  pids     PIdByPri,
                  reason   Reason;
        START
        COMMENT'The state machine contains a single state.
                 ';
        CALL      Initialize_resources;
        NEXTSTATE ~;

STATE ~; INPUT    Open.portal(p, pKind, pids);
        CALL      Open_portal(p, pKind, pids);
        NEXTSTATE -;

STATE ~; INPUT    Time.portal(p);
        CALL      Time_portal(p);
        NEXTSTATE -;

STATE ~; INPUT    Drop.portal(p, reason);
        CALL      Drop_portal(p, reason);
        NEXTSTATE -;

STATE ~; INPUT    Shut.portal(p);
        CALL      Shut_portal(p);
        DECISION pUsed = Empty;
        (False):NEXTSTATE -;
        (True): STOP;
        ENDDECISION;

STATE ~; INPUT    PDU.ready(dp);
        CALL      PDU_ready(dp);
        NEXTSTATE -;

STATE ~; INPUT    PDin(pdu); TASK pdu!kind  := PDin;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    EDrq(pdu); TASK pdu!kind  := EDrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    MCrq(pdu); TASK pdu!kind  := MCrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    MCcf(pdu); TASK pdu!kind  := MCcf;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    PCin(pdu); TASK pdu!kind  := PCin;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    MTrq(pdu); TASK pdu!kind  := MTrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    MTcf(pdu); TASK pdu!kind  := MTcf;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    PTin(pdu);  TASK pdu!kind  := PTin;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    DPum(pdu); TASK pdu!kind := DPum; CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    RJum(pdu); TASK pdu!kind := RJum; CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    AUrq(pdu); TASK pdu!kind  := AUrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
```

```
STATE ~; INPUT    AUcf(pdu); TASK pdu!kind  :=  AUcf;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    DUrq(pdu); TASK pdu!kind  :=  DUrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    DUin(pdu); TASK pdu!kind  :=  DUin;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    CJrq(pdu); TASK pdu!kind  :=  CJrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    CJcf(pdu); TASK pdu!kind  :=  CJcf;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    CLrq(pdu); TASK pdu!kind  :=  CLrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    CCrq(pdu); TASK pdu!kind  :=  CCrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    CCcf(pdu); TASK pdu!kind  :=  CCcf;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    CDrq(pdu); TASK pdu!kind  :=  CDrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    CDin(pdu); TASK pdu!kind  :=  CDin;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    CArq(pdu); TASK pdu!kind  :=  CArq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    CAin(pdu); TASK pdu!kind  :=  CAin;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    CErq(pdu); TASK pdu!kind  :=  CErq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    CEin(pdu); TASK pdu!kind  :=  CEin;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    SDrq(pdu); TASK pdu!kind  :=  SDrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    SDin(pdu); TASK pdu!kind  :=  SDin;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    USrq(pdu); TASK pdu!kind  :=  USrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    USin(pdu); TASK pdu!kind  :=  USin;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    TGrq(pdu); TASK pdu!kind  :=  TGrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    TGcf(pdu); TASK pdu!kind  :=  TGcf;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    TIrq(pdu);  TASK pdu!kind  :=  TIrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    TIcf(pdu);  TASK pdu!kind  :=  TIcf;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    TVrq(pdu); TASK pdu!kind  :=  TVrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    TVin(pdu); TASK pdu!kind  :=  TVin;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    TVrs(pdu); TASK pdu!kind  :=  TVrs;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    TVcf(pdu); TASK pdu!kind  :=  TVcf;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    TPrq(pdu); TASK pdu!kind  :=  TPrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    TPin(pdu); TASK pdu!kind  :=  TPin;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    TRrq(pdu); TASK pdu!kind  :=  TRrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    TRcf(pdu); TASK pdu!kind  :=  TRcf;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    TTrq(pdu); TASK pdu!kind  :=  TTrq;  CALL Input_PDU(pdu);   NEXTSTATE –;
STATE ~; INPUT    TTcf(pdu); TASK pdu!kind  :=  TTcf;  CALL Input_PDU(pdu);   NEXTSTATE –;

ENDPROCESS;
```

Apéndice V

Especificación en SDL del proceso punto extremo

(Este apéndice no es parte integrante de la presente Recomendación)

```
PROCESS Endpoint;
FPAR     tcId          TCEndpointId,          /* incoming TC if not Null */
         localTSAP     TSAPAddress,           /* own transport address */
         remoteTSAP    TSAPAddress,           /* other transport address */
         givenQOS      TransportQOS,          /* target or offered QOS */
         minQOS        TransportQOS,                                  /* minimum acceptable
QOS */
         parameters    DomainParameters;      /* values established or null */

        /* Data declarations */

DCL      control       PId,                   /* single Control process */
         domain        PId;                   /* selected Domain process */

        /* Input transitions */

        DCL     localDomain      DomainSelector,
                remoteDomain     DomainSelector,
                upward           Boolean,
                targetParms      DomainParameters,
                minParms         DomainParameters,
                maxParms         DomainParameters,
                userData         UserData,
                result           Result,
                ccId             Natural,
                label            Natural,
                tsdu             TSDU,
                dp               DataPriority,
                pdu              PDUStruct;
```

```
        START
        COMMENT'State machine:                  NEXTSTATE
                        1  connecting          * . 2 . . . 6
                        2  connbusy            * . 2 3 . 5 6
                        3  connready            . 2 3 . 5 6
                        4  busy                 . . . 4 5 6
                        5  ready                . . . 4 5 6
                        6  disconnected         . . . . . 6
                ';
        TASK    control  :=  PARENT,
                label  :=  0;
        DECISION tcId = Null;
        (True):  OUTPUT  T.Connect.request(label, localTSAP, remoteTSAP,
                                           givenQOS, minQOS);
                NEXTSTATE connecting;
        (False): OUTPUT  T.Connect.response(tcId, givenQOS);
                OUTPUT  T.ready(tcId);
                NEXTSTATE connbusy;
        ENDDECISION;

STATE *;
        INPUT    Exit;
        STOP;

STATE *;
        INPUT    T.Disconnect.indication(label, tcId);
        OUTPUT  Quit TO control;
        NEXTSTATE disconnected;

STATE connecting;
        INPUT    T.Connect.confirm(label, tcId, givenQOS);
        OUTPUT  T.ready(tcId);
        NEXTSTATE connbusy;

STATE connecting;
        SAVE     *;  /* await outcome of TC */

STATE connbusy, connready, busy, ready;
        INPUT    *;
        OUTPUT  T.Disconnect.request(tcId);
        OUTPUT  Quit TO control;
        NEXTSTATE disconnected;

STATE connbusy;
        INPUT    T.ready(tcId);
        NEXTSTATE connready;

STATE connbusy;
        SAVE     Connect.Initial, Connect.Response, Connect.Additional, Connect.Result;

STATE connready;
        INPUT    Connect.Initial(localDomain, remoteDomain, upward,
                 targetParms, minParms, maxParms, userData);
        TASK'tsdu  :=  encode Connect-Initial using BER';
          1b :
        OUTPUT  T.Data.request(tcId, tsdu);
        NEXTSTATE connbusy;

STATE connready;
        INPUT    Connect.Response(result, ccId, parameters, userData);
        TASK'tsdu  :=  encode Connect-Response using BER';
        JOIN 1b;

STATE connready;
        INPUT    Connect.Additional(ccId, dp);
        TASK'tsdu  :=  encode Connect-Additional using BER';
        JOIN 1b;

STATE connready;
        INPUT    Connect.Result(result);
        TASK'tsdu  :=  encode Connect-Result using BER';
        JOIN 1b;
```

```
STATE connbusy, connready;
        INPUT     T.Data.indication(tcId, tsdu);
        TASK'connect MCSPDU  :=  decode tsdu using BER';
        DECISION'connect MCSPDU';
        ('Connect-Initial'):
                OUTPUT  Connect.Initial(localDomain, remoteDomain, upward,
                        targetParms, minParms, maxParms, userData) TO control;
                NEXTSTATE -;
        ('Connect-Response'):
                OUTPUT  Connect.Response(result, ccId, parameters, userData) TO control;
                NEXTSTATE -;
        ('Connect-Additional'):
                OUTPUT  Connect.Additional(ccId, dp) TO control;
                NEXTSTATE -;
        ('Connect-Result'):
                OUTPUT  Connect.Result(result) TO control;
                NEXTSTATE -;
        ELSE:
                OUTPUT  T.Disconnect.request(tcId);
                OUTPUT  Quit TO control;
                NEXTSTATE disconnected;
        ENDDECISION;

STATE connbusy;
        INPUT     PDU.ready(dp);
        TASK      domain  :=  SENDER;
        OUTPUT  T.ready(tcId);
        NEXTSTATE ready;

STATE connready;
        INPUT     PDU.ready(dp);
        TASK      domain  :=  SENDER;
        OUTPUT  PDU.ready(dp) TO domain;
        OUTPUT  T.ready(tcId);
        NEXTSTATE ready;

STATE busy;
        INPUT     PDU.ready(dp);
        OUTPUT  T.ready(tcId);
        NEXTSTATE ready;

STATE busy, ready;
        INPUT     T.ready(tcId);
        OUTPUT  PDU.ready(dp) TO domain;
        NEXTSTATE -;

STATE busy, ready;
        INPUT   PDin(pdu); TASK pdu!kind  :=  PDin;   JOIN  2f;
        INPUT   EDrq(pdu); TASK pdu!kind  :=  EDrq;   JOIN  2f;
        INPUT   MCrq(pdu); TASK pdu!kind  :=  MCrq;   JOIN  2f;
        INPUT   MCcf(pdu); TASK pdu!kind  :=  MCcf;   JOIN  2f;
        INPUT   PCin(pdu); TASK pdu!kind  :=  PCin;   JOIN  2f;
        INPUT   MTrq(pdu); TASK pdu!kind  :=  MTrq;   JOIN  2f;
        INPUT   MTcf(pdu); TASK pdu!kind  :=  MTcf;   JOIN  2f;
        INPUT   PTin(pdu);  TASK pdu!kind  :=  PTin;   JOIN  2f;
        INPUT   DPum(pdu);TASK pdu!kind  :=  DPum; JOIN  2f;
        INPUT   RJum(pdu);TASK pdu!kind  :=  RJum;JOIN   2f;
        INPUT   AUrq(pdu); TASK pdu!kind  :=  AUrq;  JOIN   2f;
        INPUT   AUcf(pdu); TASK pdu!kind  :=  AUcf;  JOIN   2f;
        INPUT   DUrq(pdu); TASK pdu!kind  :=  DUrq;  JOIN   2f;
        INPUT   DUin(pdu); TASK pdu!kind  :=  DUin;  JOIN   2f;
        INPUT   CJrq(pdu); TASK pdu!kind  :=  CJrq;  JOIN   2f;
        INPUT   CJcf(pdu);  TASK pdu!kind  :=  CJcf;  JOIN   2f;
        INPUT   CLrq(pdu); TASK pdu!kind  :=  CLrq;  JOIN   2f;
        INPUT   CCrq(pdu); TASK pdu!kind  :=  CCrq;  JOIN   2f;
        INPUT   CCcf(pdu); TASK pdu!kind  :=  CCcf;  JOIN   2f;
        INPUT   CDrq(pdu); TASK pdu!kind  :=  CDrq;  JOIN   2f;
        INPUT   CDin(pdu); TASK pdu!kind  :=  CDin;  JOIN   2f;
```

```
        INPUT   CArq(pdu); TASK pdu!kind  :=  CArq; JOIN   2f;
        INPUT   CAin(pdu); TASK pdu!kind  :=  CAin; JOIN   2f;
        INPUT   CErq(pdu); TASK pdu!kind  :=  CErq; JOIN   2f;
        INPUT   CEin(pdu); TASK pdu!kind  :=  CEin; JOIN   2f;
        INPUT   SDrq(pdu); TASK pdu!kind  :=  SDrq; JOIN   2f;
        INPUT   SDin(pdu); TASK pdu!kind  :=  SDin; JOIN   2f;
        INPUT   USrq(pdu); TASK pdu!kind  :=  USrq; JOIN   2f;
        INPUT   USin(pdu); TASK pdu!kind  :=  USin; JOIN   2f;
        INPUT   TGrq(pdu); TASK pdu!kind  :=  TGrq; JOIN   2f;
        INPUT   TGcf(pdu); TASK pdu!kind  :=  TGcf; JOIN   2f;
        INPUT   TIrq(pdu);  TASK pdu!kind  :=  TIrq;  JOIN   2f;
        INPUT   TIcf(pdu);  TASK pdu!kind  :=  TIcf;  JOIN   2f;
        INPUT   TVrq(pdu); TASK pdu!kind  :=  TVrq; JOIN   2f;
        INPUT   TVin(pdu); TASK pdu!kind  :=  TVin; JOIN   2f;
        INPUT   TVrs(pdu); TASK pdu!kind  :=  TVrs; JOIN   2f;
        INPUT   TVcf(pdu); TASK pdu!kind  :=  TVcf; JOIN   2f;
        INPUT   TPrq(pdu); TASK pdu!kind  :=  TPrq; JOIN   2f;
        INPUT   TPin(pdu); TASK pdu!kind  :=  TPin; JOIN   2f;
        INPUT   TRrq(pdu); TASK pdu!kind  :=  TRrq; JOIN   2f;
        INPUT   TRcf(pdu); TASK pdu!kind  :=  TRcf; JOIN   2f;
        INPUT   TTrq(pdu); TASK pdu!kind  :=  TTrq; JOIN   2f;
        INPUT   TTcf(pdu); TASK pdu!kind  :=  TTcf;

          2f :
        TASK'tsdu  :=  encode pdu using BER or PER,
              depending on parameters!protocolVersion';
        OUTPUT  T.Data.request(tcId, tsdu);
        NEXTSTATE -;

STATE ready;

        INPUT     T.Data.indication(tcId, tsdu);
        TASK'pdu  :=  decode tsdu using BER or PER,
              depending on parameters!protocolVersion';
        DECISION pdu!kind;
        (RJum):   OUTPUT  RJum(pdu) TO control;
                  OUTPUT  T.ready(tcId);
                  NEXTSTATE -;

        (PDin):   OUTPUT  PDin(pdu)  TO domain;  NEXTSTATE busy;
        (EDrq):   OUTPUT  EDrq(pdu)  TO domain;  NEXTSTATE busy;
        (MCrq):   OUTPUT  MCrq(pdu)  TO domain;  NEXTSTATE busy;
        (MCcf):   OUTPUT  MCcf(pdu)  TO domain;  NEXTSTATE busy;
        (PCin):   OUTPUT  PCin(pdu)  TO domain;  NEXTSTATE busy;
        (MTrq):   OUTPUT  MTrq(pdu)  TO domain;  NEXTSTATE busy;
        (MTcf):   OUTPUT  MTcf(pdu)  TO domain;  NEXTSTATE busy;
        (PTin):   OUTPUT  PTin(pdu)  TO domain;  NEXTSTATE busy;
        (DPum):   OUTPUT  DPum(pdu)  TO domain;  NEXTSTATE busy;
        (AUrq):   OUTPUT  AUrq(pdu)  TO domain;  NEXTSTATE busy;
        (AUcf):   OUTPUT  AUcf(pdu)  TO domain;  NEXTSTATE busy;
        (DUrq):   OUTPUT  DUrq(pdu)  TO domain;  NEXTSTATE busy;
        (DUin):   OUTPUT  DUin(pdu)  TO domain;  NEXTSTATE busy;
        (CJrq):   OUTPUT  CJrq(pdu)  TO domain;  NEXTSTATE busy;
        (CJcf):   OUTPUT  CJcf(pdu)  TO domain;  NEXTSTATE busy;
        (CLrq):   OUTPUT  CLrq(pdu)  TO domain;  NEXTSTATE busy;
        (CCrq):   OUTPUT  CCrq(pdu)  TO domain;  NEXTSTATE busy;
        (CCcf):   OUTPUT  CCcf(pdu)  TO domain;  NEXTSTATE busy;
        (CDrq):   OUTPUT  CDrq(pdu)  TO domain;  NEXTSTATE busy;
        (CDin):   OUTPUT  CDin(pdu)  TO domain;  NEXTSTATE busy;
        (CArq):   OUTPUT  CArq(pdu)  TO domain;  NEXTSTATE busy;
        (CAin):   OUTPUT  CAin(pdu)  TO domain;  NEXTSTATE busy;
        (CErq):   OUTPUT  CErq(pdu)  TO domain;  NEXTSTATE busy;
        (CEin):   OUTPUT  CEin(pdu)  TO domain;  NEXTSTATE busy;
        (SDrq):   OUTPUT  SDrq(pdu)  TO domain;  NEXTSTATE busy;
        (SDin):   OUTPUT  SDin(pdu)  TO domain;  NEXTSTATE busy;
        (USrq):   OUTPUT  USrq(pdu)  TO domain;  NEXTSTATE busy;
        (USin):   OUTPUT  USin(pdu)  TO domain;  NEXTSTATE busy;
        (TGrq):   OUTPUT  TGrq(pdu)  TO domain;  NEXTSTATE busy;
        (TGcf):   OUTPUT  TGcf(pdu)  TO domain;  NEXTSTATE busy;
```

```
        (TIrq):    OUTPUT  TIrq(pdu)   TO domain;  NEXTSTATE busy;
        (TIcf):    OUTPUT  TIcf(pdu)   TO domain;  NEXTSTATE busy;
        (TVrq):    OUTPUT  TVrq(pdu)   TO domain;  NEXTSTATE busy;
        (TVin):    OUTPUT  TVin(pdu)   TO domain;  NEXTSTATE busy;
        (TVrs):    OUTPUT  TVrs(pdu)   TO domain;  NEXTSTATE busy;
        (TVcf):    OUTPUT  TVcf(pdu)   TO domain;  NEXTSTATE busy;
        (TPrq):    OUTPUT  TPrq(pdu)   TO domain;  NEXTSTATE busy;
        (TPin):    OUTPUT  TPin(pdu)   TO domain;  NEXTSTATE busy;
        (TRrq):    OUTPUT  TRrq(pdu)   TO domain;  NEXTSTATE busy;
        (TRcf):    OUTPUT  TRcf(pdu)   TO domain;  NEXTSTATE busy;
        (TTrq):    OUTPUT  TTrq(pdu)   TO domain;  NEXTSTATE busy;
        (TTcf):    OUTPUT  TTcf(pdu)   TO domain;  NEXTSTATE busy;
        ELSE:
                   TASK'pdu!initialOctets  :=  truncate tsdu';
                   TASK'pdu!diagnostic  :=  DC_invalid_?ER_encoding';
                   OUTPUT  RJum(pdu) TO domain;
                   NEXTSTATE busy;
        ENDDECISION;

STATE disconnected;
        INPUT   Quit;
        OUTPUT  Quit TO control;
        NEXTSTATE disconnected;

ENDPROCESS;
```

Apéndice VI

Especificación en SDL del proceso de anexión

(Este apéndice no es parte integrante de la presente Recomendación)

```
PROCESS Attachment;
FPAR    label           Natural,                    /* for MCS.Attach.User.confirm */
        parameters      DomainParameters;           /* values established in domain */

       /* Type definitions */

NEWTYPE         MCSRequest
STRUCT
        kind            PDUKind;                    /* SDrq, USrq, CArq, CErq */
        channelId       ChannelId;                  /* parameter of request */
        segmentation    Segmentation;               /* parameter of request */
        userData        UserData;                   /* parameter of request */
        offset          Integer;                    /* octets sent so far */
        userIds         UserIdSet;                  /* remaining to affect */
ENDNEWTYPE;

NEWTYPE         PrioritySet         SetOf(DataPriority);
ENDNEWTYPE;

NEWTYPE         MCSRequestByPri     Array(DataPriority, MCSRequest);
ENDNEWTYPE;
NEWTYPE         PDUStructByPri      Array(DataPriority, PDUStruct);
ENDNEWTYPE;

       /* Data declarations */

DCL     maId            MCSAttachmentId,            /* this Attachment process */
        control         PId,                        /* single Control process */
        domain          PId;                        /* selected Domain process */

DCL     user            UserId;                     /* unique id of attached user */

DCL     cJoined         ChannelIdSet,               /* channels the user has joined */
        cConvened       ChannelIdSet,               /* channels the user has convened */
        cAdmitted       ChannelIdSet;               /* channels user was admitted to */

DCL     tPossessed      TokenIdSet,                 /* tokens grabbed or inhibited */
        tRecipient      TokenIdSet;                 /* tokens given waiting response */
```

```
DCL     uPending        PrioritySet,                    /* request is pending at 0..3 */
        mcsreq          MCSRequestByPri,                /* content of segmented request */
        dReady          PrioritySet;                    /* domain MCSPDU allowed 0..? */

DCL     dPending        PrioritySet,                    /* SDin or USin pending at 0..3 */
        mcspdu          PDUStructByPri,                 /* content of the domain MCSPDU */
        uReady          PrioritySet;                    /* data indication allowed 0..3 */

        /* Procedure decomposition */

        /*      Segment_request         (dp)
                Indicate_data
                Track_token             (t, status)     */
                                                        /*-----------------------------*/
PROCEDURE       Segment_request;                        /*  Segment_request  */
FPAR            dp      DataPriority;                   /*-----------------------------*/

        DCL     udp     DataPriority,
                req     MCSRequest,
                pdu     PDUStruct,
                b       Boolean,
                m       Integer,
                n       Integer,
                u       UserId;
        START
        COMMENT'Feed domain process the next segment of user data
                  or the next subset of private channel user ids,
                  if ready, for the specified transport priority.
                ';
        DECISION dp in dReady;
        (False):RETURN;
        ELSE:ENDDECISION;
        TASK    udp  :=  dp;
          1b :  /* for udp = dp..? */
        DECISION udp = dp or (udp >= parameters!numPriorities and udp < 4);
        (False):  RETURN;
        (True):   DECISION udp in uPending;
                  (False): TASK   udp  :=  udp + 1;
                           JOIN 1b;
                  ELSE:ENDDECISION;
        ENDDECISION;
        TASK    dReady  :=  Del(dp, dReady),
                dp  :=  udp,
                req  :=  mcsreq(dp),
                pdu!initiator  :=  user,
                pdu!channelId  :=  req!channelId;
        DECISION req!kind;
        (SDrq, USrq):
                TASK    pdu!dataPriority  :=  dp,
                        b  :=  req!segmentation!begin,
                        pdu!segmentation!begin  :=  IF req!offset = 0 THEN b ELSE False FI,
                        m  :=  parameters!maxMCSPDUsize,
                        m  :=  IF parameters!protocolVersion = 1 THEN m - 24 ELSE m - 8 FI,
                        n  :=  Length(req!userData) - req!offset,
                        n  :=  IF n > m THEN m ELSE n FI,
                        pdu!userData  :=  Substring(req!userData, 1 + req!offset, n),
                        req!offset  :=  req!offset + n,
                        n  :=  Length(req!userData) - req!offset,
                        b  :=  req!segmentation!end,
                        pdu!segmentation!end  :=  IF n = 0 THEN b ELSE False FI,
                        mcsreq(dp)!offset  :=  req!offset;
        (CArq, CErq):
                TASK    pdu!userIds  :=  Empty,
                        n  :=  0,
                        m  :=  parameters!maxMCSPDUsize,
                        m  :=  IF parameters!protocolVersion = 1 THEN (m - 16) / 4
                                        ELSE (m - 7) / 2 FI;
                  2b :  /* while n < m */
```

```
                DECISION req!userIds = Empty;
                (True):  TASK    n  :=  0;
                (False): TASK    u  :=  Pick(req!userIds),
                                 req!userIds  :=  Del(u, req!userIds);
                         DECISION u >= 1001;
                         (True):  TASK    pdu!userIds  :=  Incl(u, pdu!userIds);
                         ELSE:ENDDECISION;
                         TASK    n  :=  n + 1;
                         DECISION n < m;
                         (True): JOIN 2b;
                         ELSE:ENDDECISION;
                ENDDECISION;
        ENDDECISION;
        DECISION req!kind;
        (SDrq):  OUTPUT  SDrq(pdu) TO domain;
        (USrq):  OUTPUT  USrq(pdu) TO domain;
        (CArq):  OUTPUT  CArq(pdu) TO domain;
        (CErq):  OUTPUT  CErq(pdu) TO domain;
        ENDDECISION;
        DECISION n = 0;
        (True):  TASK    uPending  :=  Del(dp, uPending);
                 DECISION req!kind;
                 (SDrq, USrq):
                         OUTPUT  MCS.ready(mald, dp);
                 ELSE:ENDDECISION;
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;

                                                                  /*----------------------*/
PROCEDURE       Indicate_data;                                    /*  Indicate_data  */
                                                                  /*----------------------*/

        DCL     dp DataPriority,
                pdu     PDUStruct;
        START
        COMMENT'Indicate the receipt of user data, if ready
                 and none pending at higher priorities.
                 ';
        TASK    dp  :=  0;
          1b :  /* for dp = 0..3 */
        DECISION dp < 4 and (dp in uReady or not dp in dPending);
        (False):  RETURN;
        (True):   DECISION dp in dPending;
                  (False): TASK    dp  :=  dp + 1;
                           JOIN 1b;
                  ELSE:ENDDECISION;
        ENDDECISION;
        TASK    dPending  :=  Del(dp, dPending),
                pdu  :=  mcspdu(dp);
        DECISION pdu!kind;
        (SDin):   OUTPUT  MCS.Send.Data.indication(mald, pdu!channelId,
                          pdu!dataPriority, pdu!initiator, pdu!segmentation, pdu!userData);
        (USin):   OUTPUT  MCS.Uniform.Send.Data.indication(mald, pdu!channelId,
                          pdu!dataPriority, pdu!initiator, pdu!segmentation, pdu!userData);
        ENDDECISION;
        TASK    uReady  :=  Del(dp, uReady),
                dp  :=  IF dp < parameters!numPriorities THEN dp
                                ELSE parameters!numPriorities - 1 FI;
        OUTPUT  PDU.ready(dp) TO domain;
        JOIN 1b;
        ENDPROCEDURE;

                                                                  /*----------------------*/
PROCEDURE               Track_token;                              /*  Track_token  */
FPAR                    t          TokenId,                       /*----------------------*/
                        status  TokenStatus;
```

```
        START
        COMMENT'Condense status into possessed or recipient.
                ';
        TASK    tPossessed  :=  Del(t, tPossessed),
                tRecipient  :=  Del(t, tRecipient);
        DECISION status;
        (SelfGrabbed, SelfInhibited, SelfGiving):
                TASK    tPossessed  :=  Incl(t, tPossessed);
        (SelfRecipient):
                TASK    tRecipient  :=  Incl(t, tRecipient);
        ELSE:ENDDECISION;
        RETURN;
        ENDPROCEDURE;

        /* Input transitions */

        DCL     dp              DataPriority,
                pdu             PDUStruct,
                c               ChannelId,
                cSet            ChannelIdSet,
                t               TokenId,
                u               UserId,
                uSet            UserIdSet,
                segmentation    Segmentation,
                userData        UserData,
                result          Result,
                kind            PDUKind,
                reason          Reason;
        START
        COMMENT'State machine:                  NEXTSTATE
                        1  initial              * . 2 . . . 6
                        2  attaching              . 2 3 4 . 6
                        3  busy                   . . 3 4 5 6
                        4  ready                  . . 3 4 . 6
                        5  detaching               . . . . 5 6
                        6  detached               . . . . . 6
                ';
        TASK    mald  :=  SELF,
                control  :=  PARENT,
                user  :=  0;
        NEXTSTATE initial;

STATE *;
        INPUT   Exit;
        STOP;

STATE initial, attaching;
        INPUT   *;
        OUTPUT  MCS.Attach.User.confirm(label, RT_unspecified_failure, mald, user);
        OUTPUT  Quit TO control;
        NEXTSTATE detached;

STATE initial, attaching;
        INPUT   Quit;
        OUTPUT  MCS.Attach.User.confirm(label, RT_domain_disconnected, mald, user);
        OUTPUT  Quit TO control;
        NEXTSTATE detached;

STATE initial;
        INPUT   PDU.ready(dp);
        TASK    domain  :=  SENDER;
        DECISION dp = 0;
        (False):  TASK  dReady  :=  Incl(dp, dReady);
                NEXTSTATE -;
        ELSE:ENDDECISION;
        OUTPUT  PDU.ready(0) TO domain;
        OUTPUT  AUrq(pdu) TO domain;
        NEXTSTATE attaching;
```

```
STATE attaching;
        INPUT     PDU.ready(dp);
        TASK      dReady  :=  Incl(dp, dReady);
        NEXTSTATE -;

STATE attaching;
        INPUT     PCin(pdu), PTin(pdu), DUin(pdu);
        /* no action */
        NEXTSTATE -;

STATE attaching;
        INPUT     AUcf(pdu);
        TASK      user  :=  pdu!initiator;
        OUTPUT    MCS.Attach.User.confirm(label, pdu!result, mald, user);
        DECISION pdu!result = RT_successful;
        (False):  OUTPUT  Quit TO control;
                  NEXTSTATE detached;
        (True):   TASK    dp  :=  0;
                    1b :    /* for dp = 0..3 */
                  DECISION dp < 4;
                  (True):  OUTPUT  MCS.ready(mald, dp);
                           DECISION dp < parameters!numPriorities;
                           (True):  OUTPUT  PDU.ready(dp) TO domain;
                           ELSE:ENDDECISION;
                           TASK    dp  :=  dp + 1;
                           JOIN 1b;
                  ELSE:ENDDECISION;
                  DECISION 0 in dReady;
                  (False): NEXTSTATE busy;
                  (True):  NEXTSTATE ready;
                  ENDDECISION;
        ENDDECISION;

STATE busy, ready;
        INPUT     Quit;
        OUTPUT    MCS.Detach.User.indication(mald, user, RN_provider_initiated);
        OUTPUT    Quit TO control;
        NEXTSTATE detached;

STATE busy, ready;
        INPUT     MCS.ready(mald, dp);
        TASK      uReady  :=  Incl(dp, uReady);
        CALL      Indicate_data;
        NEXTSTATE -;

STATE busy;
        INPUT     MCS.Detach.User.request(mald);
        TASK      reason  :=  RN_user_requested;
        NEXTSTATE detaching;

STATE busy;
        SAVE      *;  /* defer MCS other */

STATE ready;
        INPUT     MCS.Detach.User.request(mald);
        TASK      pdu!reason  :=  RN_user_requested,
                  pdu!userIds  :=  Incl(user, Empty);
        OUTPUT    DUrq(pdu) TO domain;
        NEXTSTATE detached;

STATE ready;
        INPUT     MCS.Channel.Join.request(mald, c);
        TASK      pdu!initiator  :=  user,
                  pdu!channelId  :=  c;
        OUTPUT    CJrq(pdu) TO domain;
          2b :
        TASK      dReady  :=  Del(0, dReady);
        NEXTSTATE busy;
```

```
STATE ready;
        INPUT    MCS.Channel.Leave.request(mald, c);
        TASK     cJoined := Del(c, cJoined),
                 pdu!channelIds := Incl(c, Empty);
        OUTPUT   CLrq(pdu) TO domain;
        JOIN 2b;

STATE ready;
        INPUT    MCS.Channel.Convene.request(mald);
        TASK     pdu!initiator := user;
        OUTPUT   CCrq(pdu) TO domain;
        JOIN 2b;

STATE ready;
        INPUT    MCS.Channel.Disband.request(mald, c);
        DECISION c in cConvened;
        (True):  TASK   cAdmitted := Del(c, cAdmitted),
                        cJoined := Del(c, cJoined);
        ELSE:ENDDECISION;
        TASK     cConvened := Del(c, cConvened),
                 pdu!initiator := user,
                 pdu!channelId := c;
        OUTPUT   CDrq(pdu) TO domain;
        JOIN 2b;

STATE ready;
        INPUT    MCS.Channel.Admit.request(mald, c, uSet);
        TASK     kind := CArq;
        JOIN 3f;

STATE ready;
        INPUT    MCS.Channel.Expel.request(mald, c, uSet);
        TASK     kind := CErq;
          3f :
        TASK     mcsreq(0)!kind := kind,
                 mcsreq(0)!channelId := c,
                 mcsreq(0)!userIds := uSet,
                 uPending := Incl(0, uPending);
        CALL     Segment_request(0);
        DECISION 0 in dReady;
        (False):NEXTSTATE busy;
        (True): NEXTSTATE ready;
        ENDDECISION;

STATE ready;
        INPUT    MCS.Send.Data.request(mald, c, dp, segmentation, userData);
        TASK     kind := SDrq;
        JOIN 4f;

STATE ready;
        INPUT    MCS.Uniform.Send.Data.request(mald, c, dp, segmentation, userData);
        TASK     kind := USrq;
          4f :
        DECISION dp >= 4 or dp in uPending;
        (True):  OUTPUT  MCS.Detach.User.indication(mald, user, RN_provider_initiated);
                 TASK    pdu!reason := RN_provider_initiated,
                         pdu!userIds := Incl(user, Empty);
                 OUTPUT  DUrq(pdu) TO domain;
                 NEXTSTATE detached;
        (False): TASK    mcsreq(dp)!kind := kind,
                         mcsreq(dp)!channelId := c,
                         mcsreq(dp)!segmentation := segmentation,
                         mcsreq(dp)!userData := userData,
                         mcsreq(dp)!offset := 0,
                         uPending := Incl(dp, uPending),
                         dp := IF dp < parameters!numPriorities THEN dp
                                       ELSE parameters!numPriorities - 1 FI;
                 CALL    Segment_request(dp);
                 DECISION 0 in dReady;
                 (False): NEXTSTATE busy;
```

```
                (True):  NEXTSTATE ready;
                ENDDECISION;
        ENDDECISION;

STATE ready;
        INPUT   MCS.Token.Grab.request(mald, t);
        TASK    pdu!initiator := user,
                pdu!tokenId := t;
        OUTPUT  TGrq(pdu) TO domain;
        JOIN 2b;

STATE ready;
        INPUT   MCS.Token.Inhibit.request(mald, t);
        TASK    pdu!initiator := user,
                pdu!tokenId := t;
        OUTPUT  TIrq(pdu) TO domain;
        JOIN 2b;

STATE ready;
        INPUT   MCS.Token.Give.request(mald, t, u);
        DECISION u >= 1001;
        (False):  OUTPUT  MCS.Token.Give.confirm(mald, t, RT_no_such_user);
                NEXTSTATE -;
        (True):   TASK    pdu!initiator := user,
                          pdu!tokenId := t,
                          pdu!recipient := u;
                OUTPUT  TVrq(pdu) TO domain;
                JOIN 2b;
        ENDDECISION;

STATE ready;
        INPUT   MCS.Token.Give.response(mald, t, result);
        DECISION result  = RT_successful;
        (False):TASK      result := RT_user_rejected;
        ELSE:ENDDECISION;
        DECISION t in tRecipient and result = RT_successful;
        (True):   TASK  tPossessed := Incl(t, tPossessed);
        ELSE:ENDDECISION;
        TASK    tRecipient := Del(t, tRecipient),
                pdu!result := result,
                pdu!recipient := user,
                pdu!tokenId := t;
        OUTPUT  TVrs(pdu) TO domain;
        JOIN 2b;

STATE ready;
        INPUT   MCS.Token.Please.request(mald, t);
        TASK    pdu!initiator := user,
                pdu!tokenId := t;
        OUTPUT  TPrq(pdu) TO domain;
        JOIN 2b;

STATE ready;
        INPUT   MCS.Token.Release.request(mald, t);
        TASK    tPossessed := Del(t, tPossessed),
                pdu!initiator := user,
                pdu!tokenId := t;
        OUTPUT  TRrq(pdu) TO domain;
        JOIN 2b;

STATE ready;
        INPUT   MCS.Token.Test.request(mald, t);
        TASK    pdu!initiator := user,
                pdu!tokenId := t;
        OUTPUT  TTrq(pdu) TO domain;
        JOIN 2b;

STATE busy, ready;
        INPUT   PDU.ready(dp);
        TASK    dReady := Incl(dp, dReady);
        CALL    Segment_request(dp);
```

```
        DECISION 0 in dReady;
        (False):   NEXTSTATE busy;
        (True):    NEXTSTATE ready;
        ENDDECISION;

STATE busy, ready;
        INPUT      PCin(pdu);
        TASK       reason  :=  RN_channel_purged;
        DECISION user in pdu!detachUserIds;
        (True):    OUTPUT  MCS.Detach.User.indication(mald, user, reason);
                   OUTPUT  Quit TO control;
                   NEXTSTATE detached;
        (False):   TASK    uSet  :=  pdu!detachUserIds;
                     5b :  /* for u in uSet */
                   DECISION uSet = Empty;
                   (False): TASK    u  :=  Pick(uSet),
                                    uSet  :=  Del(u, uSet);
                            OUTPUT  MCS.Detach.User.indication(mald, u, reason);
                            JOIN 5b;
                   ELSE:ENDDECISION;
                   TASK    cSet  :=  pdu!purgeChannelIds;
                     6b :  /* for c in cSet */
                   DECISION cSet = Empty;
                   (False): TASK    c  :=  Pick(cSet),
                                    cSet  :=  Del(c, cSet);
                            DECISION c in cConvened;
                            (True):  TASK  cConvened  :=  Del(c, cConvened),
                                           cAdmitted  :=  Del(c, cAdmitted),
                                           cJoined  :=  Del(c, cJoined);
                                     OUTPUT  MCS.Channel.Disband.indication(mald, c, reason);
                            ELSE:ENDDECISION;
                            DECISION c in cAdmitted;
                            (True):  TASK  cAdmitted  :=  Del(c, cAdmitted),
                                           cJoined  :=  Del(c, cJoined);
                                     OUTPUT  MCS.Channel.Expel.indication(mald, c, reason);
                            ELSE:ENDDECISION;
                            DECISION c in cJoined;
                            (True):  TASK  cJoined  :=  Del(c, cJoined);
                                     OUTPUT  MCS.Channel.Leave.indication(mald, c, reason);
                            ELSE:ENDDECISION;
                            JOIN 6b;
                   ELSE:ENDDECISION;
                   OUTPUT  PDU.ready(0) TO domain;
                   NEXTSTATE -;
        ENDDECISION;

STATE busy;
        INPUT      PTin(pdu);
        DECISION (pdu!purgeTokenIds and (tPossessed or tRecipient)) = Empty;
        (False):   OUTPUT  MCS.Detach.User.indication(mald, user, RN_token_purged);
                   TASK    reason  :=  RN_token_purged;
                   NEXTSTATE detaching;
        (True):    OUTPUT  PDU.ready(0) TO domain;
                   NEXTSTATE -;
        ENDDECISION;

STATE ready;
        INPUT      PTin(pdu);
        DECISION (pdu!purgeTokenIds and (tPossessed or tRecipient)) = Empty;
        (False):   OUTPUT  MCS.Detach.User.indication(mald, user, RN_token_purged);
                   TASK    pdu!reason  :=  RN_token_purged,
                           pdu!userIds  :=  Incl(user, Empty);
                   OUTPUT  DUrq(pdu) TO domain;
                   NEXTSTATE detached;
        (True):    OUTPUT  PDU.ready(0) TO domain;
                   NEXTSTATE -;
        ENDDECISION;
```

```
STATE busy, ready;
        INPUT     AUcf(pdu);
        OUTPUT  MCS.Detach.User.indication(mald, user, RN_unspecified);
        OUTPUT  Quit TO control;
        NEXTSTATE detached;

STATE busy, ready;
        INPUT     DUin(pdu);
        DECISION user in pdu!userIds;
        (True):    OUTPUT  MCS.Detach.User.indication(mald, user, pdu!reason);
                   OUTPUT  Quit TO control;
        NEXTSTATE detached;
        (False):   TASK    uSet  :=  pdu!userIds;
                     7b :    /* for u in uSet */
                   DECISION uSet = Empty;
                   (False): TASK    u  :=  Pick(uSet),
                                    uSet  :=  Del(u, uSet);
                            OUTPUT  MCS.Detach.User.indication(mald, u, pdu!reason);
                            JOIN 7b;
                   ELSE:ENDDECISION;
                   OUTPUT  PDU.ready(0) TO domain;
                   NEXTSTATE -;
        ENDDECISION;

STATE busy, ready;
        INPUT     CJcf(pdu);
        TASK      c  :=  pdu!channelId;
        DECISION pdu!result = RT_successful;
        (True):    TASK    cJoined  :=  Incl(c, cJoined);
        ELSE:ENDDECISION;
        OUTPUT  MCS.Channel.Join.confirm(mald, pdu!requested, pdu!result, c);
          8b :
        OUTPUT  PDU.ready(0) TO domain;
        NEXTSTATE -;

STATE busy, ready;
        INPUT     CCcf(pdu);
        TASK      c  :=  pdu!channelId;
        DECISION pdu!result = RT_successful;
        (True):    TASK    cConvened  :=  Incl(c, cConvened),
                           cAdmitted  :=  Incl(c, cAdmitted);
        ELSE:ENDDECISION;
        OUTPUT  MCS.Channel.Convene.confirm(mald, pdu!result, c);
        JOIN 8b;

STATE busy, ready;
        INPUT     CDin(pdu);
        TASK      c  :=  pdu!channelId,
                  reason  :=  RN_channel_disbanded;
        DECISION c in cConvened;
        (True):    TASK    cConvened  :=  Del(c, cConvened),
                           cAdmitted  :=  Del(c, cAdmitted),
                           cJoined  :=  Del(c, cJoined);
                   OUTPUT  MCS.Channel.Disband.indication(mald, c, reason);
        ELSE:ENDDECISION;
        DECISION c in cAdmitted;
        (True):    TASK    cAdmitted  :=  Del(c, cAdmitted),
                           cJoined  :=  Del(c, cJoined);
                   OUTPUT  MCS.Channel.Expel.indication(mald, c, reason);
        ELSE:ENDDECISION;
        JOIN 8b;

STATE busy, ready;
        INPUT     CAin(pdu);
        TASK      c  :=  pdu!channelId,
                  cAdmitted  :=  Incl(c, cAdmitted);
        OUTPUT  MCS.Channel.Admit.indication(mald, c, pdu!initiator);
        JOIN 8b;
```

```
STATE busy, ready;
        INPUT    CEin(pdu);
        TASK     c := pdu!channelId,
                 reason := RN_user_requested;
        DECISION c in cAdmitted;
        (True):  TASK   cAdmitted := Del(c, cAdmitted),
                        cJoined := Del(c, cJoined);
                 OUTPUT MCS.Channel.Expel.indication(mald, c, reason);
        ELSE:ENDDECISION;
        JOIN 8b;

STATE busy, ready;
        INPUT    SDin(pdu);
        TASK     pdu!kind := SDin;
        JOIN 9f;

STATE busy, ready;
        INPUT    USin(pdu);
        TASK     pdu!kind := USin;
          9f :
        TASK     c := pdu!channelId,
                 dp := pdu!dataPriority;
        DECISION c in cJoined;
        (True):  TASK   mcspdu(dp) := pdu,
                        dPending := Incl(dp, dPending);
                 CALL   Indicate_data;
        (False): TASK   dp := IF dp < parameters!numPriorities THEN dp
                                    ELSE parameters!numPriorities - 1 FI;
                 OUTPUT PDU.ready(dp) TO domain;
        ENDDECISION;
        NEXTSTATE -;

STATE busy, ready;
        INPUT    TGcf(pdu);
        CALL     Track_token(pdu!tokenId, pdu!tokenStatus);
        OUTPUT   MCS.Token.Grab.confirm(mald, pdu!tokenId, pdu!result);
        JOIN 8b;

STATE busy, ready;
        INPUT    TIcf(pdu);
        CALL     Track_token(pdu!tokenId, pdu!tokenStatus);
        OUTPUT   MCS.Token.Inhibit.confirm(mald, pdu!tokenId, pdu!result);
        JOIN 8b;

STATE busy, ready;
        INPUT    TVin(pdu);
        TASK     tRecipient := Incl(pdu!tokenId, tRecipient);
        OUTPUT   MCS.Token.Give.indication(mald, pdu!tokenId, pdu!initiator);
        JOIN 8b;

STATE busy, ready;
        INPUT    TVcf(pdu);
        CALL     Track_token(pdu!tokenId, pdu!tokenStatus);
        OUTPUT   MCS.Token.Give.confirm(mald, pdu!tokenId, pdu!result);
        JOIN 8b;

STATE busy, ready;
        INPUT    TPin(pdu);
        TASK     t := pdu!tokenId;
        DECISION t in tPossessed or t in tRecipient;
        (True):  OUTPUT MCS.Token.Please.indication(mald, t, pdu!initiator);
        ELSE:ENDDECISION;
        JOIN 8b;

STATE busy, ready;
        INPUT    TRcf(pdu);
        CALL     Track_token(pdu!tokenId, pdu!tokenStatus);
        OUTPUT   MCS.Token.Release.confirm(mald, pdu!tokenId, pdu!result);
        JOIN 8b;
```

```
STATE busy, ready;
        INPUT     TTcf(pdu);
        OUTPUT  MCS.Token.Test.confirm(mald, pdu!tokenId, pdu!tokenStatus);
        JOIN 8b;

STATE detaching, detached;
        INPUT     *;
        NEXTSTATE -;

STATE detaching, detached;
        INPUT     Quit;
        OUTPUT  Quit TO control;
        NEXTSTATE detached;

STATE detaching;
        INPUT     PDU.ready(dp);
        DECISION dp = 0;
        (False):  NEXTSTATE -;
        (True):   TASK    pdu!reason  :=  reason,
                          pdu!userIds  :=  Incl(user, Empty);
                  OUTPUT  DUrq(pdu) TO domain;
                  NEXTSTATE detached;
        ENDDECISION;

STATE detaching, detached;
        INPUT     PCin(pdu);
        DECISION user in pdu!detachUserIds;
        (False):  NEXTSTATE -;
        (True):   OUTPUT  Quit TO control;
                  NEXTSTATE detached;
        ENDDECISION;

STATE detaching, detached;
        INPUT     DUin(pdu);
        DECISION user in pdu!userIds;
        (False):  NEXTSTATE -;
        (True):   OUTPUT  Quit TO control;
                  NEXTSTATE detached;
        ENDDECISION;

ENDPROCESS;
```

Apéndice VII

Características de la realización de referencia

(Este apéndice no es parte integrante de la presente Recomendación)

VII.1 Descomposición SDL

La Figura II.1/T.125 describe un proveedor MCS en el contexto de un sistema que está enlazado con su entorno mediante tres canales: Control.MCSAP a una sola aplicación de controlador, MCSAP a cero o más usuarios anexionados, y TSAP a cero o más proveedores de servicio de transporte. En una descomposición del proveedor, estos canales externos conectan con rutas de señales hacia y desde procesos componentes.

Cada elemento octagonal representa un tipo de proceso y da los límites (máximo, mínimo) impuestos al número de sus instancias. Un proceso control existe permanentemente. Las líneas de trazo discontinuo indican que crea instancias de los tres procesos restantes. Los procesos anexión (Attachment), dominio (Domain), y punto extremo (Endpoint) no existen inicialmente pero su número puede ser ilimitado.

Las rutas de señales hacia y desde procesos componentes transportan señales SDL. El conjunto de señales transmitidas en un sentido dado se muestra cerca de la correspondiente punta de flecha. Los identificadores entre paréntesis son listas de señales conexas. Las expansiones de estas listas no se incluyen en la figura por ser demasiado largas. Los detalles se dan en el texto del Apéndice II.

VII.2 Definiciones de servicio

Las señales transmitidas por canales externos representan utilizaciones de MCS y los servicios de transporte definidos de forma abstracta en la Recomendación UIT-T T.122 y en la Recomendación X.214 del CCITT. Su modelado en SDL requiere supuestos adicionales sobre detalles de las interacciones. Dos aspectos merecen destacarse especialmente: una utilización de identificadores de punto extremo para especificar el contexto de una primitiva de servicio y el uso de señales preparado (ready) de control de flujo para invitar a la transferencia de datos.

Los tres identificadores empleados son identificador de conexión MCS (MCSConnectionId), identificador de anexión MCS (MCSAttachmentId), e identificador de punto extremo de TC (TCEndpointId). Estos parámetros son implícitos (no se representan) en los documentos de definiciones de servicio. En el modelo aquí presentado se supone que son asignados por el proveedor responsable. Por ejemplo, TCEndpoint aparece en T.Connect.indication y T.Connect.confirm. Para permitir al usuario de un servicio relacionar una confirmación con una petición precedente, el usuario puede especificar una etiqueta arbitraria para que se devuelva en eco.

El control de flujo se modela aquí como un mecanismo de «ventana de uno». Hay que enviar una señal de preparado en un sentido para poder enviar una señal de transferencia de datos en el sentido opuesto. Antes de la transferencia siguiente deberá haberse recibido otra señal de preparado. En una realización práctica, la señal de preparado podría tomar la forma del movimiento de un puntero o la incrementación de un contador. La construcción de preparado como una señal no entraña que tenga tanto peso como una transferencia de datos. Hay tres señales de preparado: T.preparado (T.ready), PDU.preparado (PDU.ready), y MCS.preparado (MCS.ready). Estas señales permiten la transferencia, respectivamente, de una TSDU, una MCSPDU de dominio, y una unidad de datos de servicio MCS. El control de flujo se aplica independientemente a los dos sentidos de transmisión de una ruta de señalización.

La calidad del servicio de transporte en una TC se modela aquí con parámetros para el caudal, el retardo de tránsito y la prioridad de datos. Esto es una amalgama de normas que no concuerdan entre sí. En la práctica, los detalles seguramente diferirán. La especificación de la calidad de servicio es una carga que las aplicaciones de controlador deben soportar.

VII.3 Portales para un dominio

El proceso dominio es esencial en esta descomposición de un proveedor MCS. Instancias diferentes representan dominios diferentes atendidos por el mismo proveedor. Un proceso dominio se crea con un conjunto de parámetros que conservan sus valores (se mantienen «congelados») durante toda su existencia. Estos parámetros de dominio son proporcionados por el proceso control. El proceso control determina qué selectores de dominio son válidos y cómo deberán configurarse los correspondientes procesos dominio. Si hay margen para la negociación de parámetros durante MCS.Conexión.Proveedor, el proceso control supervisa la interacción.

Para mantener el proceso dominio lo más sencillo posible, las anexiones MCS y las conexiones MCS con las que dicho proceso está interconectado se han representado en forma abstracta, de modo que aparezcan como portales con un número de prestaciones comunes. Un portal es un proceso anexión individual o un conjunto de procesos punto extremo, uno para cada una de las TC que forman una conexión MCS. El proceso control asigna identificadores de portal y abre y cierra sus asociaciones a un proceso dominio. Oculta los detalles de los procesos de aparición y desaparición, y oculta el intercambio de las MCSPDU de conexión.

El proceso dominio percibe un portal como un conjunto de rutas de señales, una para cada prioridad de datos implementada. Cada ruta de señales transporta MCSPDU sometidas a control de flujo por PDU.listo. Donde sea inevitable recurrir a diferenciaciones en el procesamiento, ello se hará clasificando los portales en tres clases: anexionado, enlace ascendente o enlace descendente.

Las Figuras VII.1 a VII.8 ilustran flujos de señales en operaciones típicas. Se supone un usuario anexionado localmente a un dominio con una conexión MCS a otro proveedor. Aparecen dos puntos extremos (1 y 2) porque se ha partido del supuesto de que en el dominio se han aplicado dos prioridades de datos.

La señalización para cerrar un portal es quizás excesivamente larga. Protege contra la posibilidad de que uno de los procesos participantes pudiera enviar una señal a otro que ya se hubiera detenido, lo que constituiría un error durante la ejecución en SDL. Los procesos anexión y punto extremo prevén que «Quit» sea la última señal y se detienen al recibir «Exit». El proceso dominio no envía más señales a un portal después de contestar a control con la señal cerrar.portal, y se detiene después de haber indicado así que el último portal está cerrado.

En cuanto a las diversas señales restantes, dejar.portal (Drop.portal) es bidireccional y corresponde a **DPum**. Informar.portal (Report.portal) avisa a control que se envió un diagnóstico a otro proveedor MCS. Una **RJum** originada en un punto extremo se envía a través de dominio hasta llegar a la TC inicial. Una **RJum** recibida del exterior se despacha directamente de punto extremo a control.

Obsérvese que petición.MCS.Anexión.Usuario e indicación.T.Conexión están dirigidas a control, pues son estímulos para crear portales correspondientes. Control puede rehusar transmitiendo una confirmación.MCS.Anexión.Usuario negativa o una petición.T.Desconexión.

El presente modelo no limita el número de anexiones MCS o conexiones MCS en cada dominio, salvo un límite local que impone al número de portales a través de todos los dominios atendidos por el proveedor. Muchos aspectos de la configuración del dominio tienen que gestionarse localmente hasta que se normalicen más adelante.

VII.4 Alineación de las MCSPDU

Para mayor claridad, las MCSPDU definidas en la cláusula 7 se representan individualmente por señales SDL. Sin embargo, para conseguir una presentación sencilla del proceso dominio es necesario un tratamiento más unificado. La solución consiste en definir un tipo de datos estructura PDU (PDUStruct) con componentes que pueden seleccionarse de modo que concuerden con cualquiera de las MCSPDU de dominio. Cada una de las MCSPDU de conexión, que se tratan fuera del proceso dominio, conserva su propia estructura.

Un componente clave de PDUStruct es su clase, que identifica la MCSPDU de dominio pretendida. En base a la clase, otros campos se hacen entonces relevantes: los listados en ASN.1 como componentes de la MCSPDU. Los tipos de datos definidos en SDL concuerdan lo más exactamente posible con los tipos de datos definidos en ASN.1 de la cláusula 7. Debe quedar clara la intención de que una PDUStruct sea la representación interna, decodificada, de una MCSPDU de dominio.

Tres MCSPDU, **AUcf**, **CJcf** y **CCcf**, tienen componentes facultativos. En SDL su ausencia se indica codificando con cero el campo correspondiente. Este es un valor ilegal para un identificador de canal estático o dinámico.

Las señales enviadas entre procesos anexión, dominio, y punto extremo, con excepción de PDU.listo, toma una PDUStruct como su único parámetro. En una implementación práctica, este tipo de comunicación podría ser tan sencillo como el desplazamiento de un puntero de memoria tampón.

VII.5 Método para la utilización de SDL

La realización de referencia emplea la representación textual de SDL como un lenguaje de programación potenciado con la aptitud para definir tipos de datos abstractos. En los procesos control y dominio los objetos gestionados son demasiado complejos y numerosos para aprovechar las ventajas del tipo de máquina de estados finitos soportada por SDL. Estos procesos permanecen en un solo estado durante el procesamiento de las señales de entrada. Los procesos anexión y punto extremo, cuyo alcance es menor, pueden sacar más ventaja de la utilización de algunos estados.

El Apéndice II contiene una definición del generador conjunto de (SetOf), que surte efecto en todo el proveedor. Este generador toma cualquier otro tipo definido, por ejemplo identificadores de canal, y formaliza el concepto de subconjuntos de valores de ese tipo. SetOf extiende el generador incorporado Powerset mediante la adición de un nuevo operador que elige un elemento arbitrario tomándolo de un subconjunto no vacío. El significado de este operador, Pick, se define mediante axiomas simples.

En la realización se utilizan mucho los conjuntos. Por ejemplo, una parte de la base de información está destinada a registrar qué subconjunto de identificadores de canal se están utilizando y qué subconjunto de identificadores de portal se han incorporado a un canal dado.

En SDL, se utilizan formaciones (arrays) para cubrir la totalidad de la gama del tipo de datos concernientes a la indexación. Existen, por ejemplo, estructuras de canal para cada identificador de canal de 0 a 65535. Una realización práctica tiene que trabajar con formaciones más esparcidas. Con este fin se mantienen conjuntos de identificadores distintos para registrar los canales y otros recursos que se están utilizando en un momento dado. Un principio del diseño es que los valores de las formaciones no tienen que estar almacenadas en la base de información para identificadores que no están explícitamente marcados que están en uso.

Muchas iteraciones siguen una patrón usual: se crea un conjunto candidato deseado, y entonces, repetidamente, se escoge y se suprime un miembro del conjunto, hasta que no quede ninguno. Tales iteraciones se componen de decisiones y uniones. Mediante comentarios se indica la estructura de control de alto nivel deseada. Las etiquetas se numeran secuencialmente dentro de un procedimiento, agregándoseles una letra para indicar si las ramas correspondientes son de ida o de retorno.

Antes de la primera definición de procedimiento en cada proceso hay una tabla de contenidos que muestra las secuencias de llamada. Los procedimientos SDL pueden retornar resultados mediante parámetros formales que se declaran como entrada/salida.

Dispersas dentro del código aparecen algunas constantes como el número máximo de prioridades de datos (4) y la línea de demarcación entre los identificadores de canal estático y dinámico (1001). Estos no son parámetros cuyos valores pueden redefinirse.

Una noción importante en MCS es que los identificadores de usuario son un subconjunto de los identificadores de canal. Sin embargo, los contextos en que respectivamente se utilizan son diferentes, por lo que es apropiado considerar dos tipos distintos. El Apéndice II formaliza un par de operadores para el «casting» entre ellas.

Los procesos de los Apéndices III a VI cooperan en el marco especificado por el Apéndice II. Se constriñe a las señales a seguir las rutas especificadas, y no se permite un comportamiento incorrecto a través de interfaces. Una codificación defensiva concentra su atención en las aplicaciones de controlador y de usuario que residen en el entorno fuera del proveedor y en las MCSPDU recibidas externamente de proveedores pares.

La realización de referencia permite disponer que las señales de entrada se transmitan en una secuencia arbitraria cuando más de una de ellas está pendiente. No se parte de ningún supuesto en cuanto a la prioridad relativa de los procesos. Sin embargo, si confía en que las distintas transiciones de entrada se ejecuten hasta su conclusión sin preapropiación.

VII.6 Observaciones sobre el proceso dominio

El generador cola (Queue) se utiliza considerablemente, además de SetOf. Las memorias tampón se introducen en cola, en la secuencia en que fueron transmitidas por el usuario, hacia un portal de salida, y los identificadores de portal se introducen en cola en el mismo orden de las MCSPDU **MCrq**, **MTrq**, y **AUrq** no contestadas.

Para no tener que copiar datos de usuario, las MCSPDU se manipulan en memorias tampón, que pueden pasarse a múltiples puertos. El número de memorias tampón disponibles por un proceso dominio se modela como un parámetro externo fijo. Este parámetro podría cambiarse fácilmente, haciéndolo variar, mediante configuración, de un dominio a otro. La realización se degrada paulatinamente, mediante control de flujo global, si el número de memorias tampón proporcionadas es pequeño. Aplica las memorias tampón, según se liberan, al flujo de datos con prioridad más alta.

Cuando una MCSPDU llega como una entrada, el proceso dominio necesita saber el portal de origen y la prioridad de datos. El procedimiento identificar emisor (Identify_sender) efectúa una búsqueda exhaustiva en los portales abiertos para hallar el identificador de proceso comunicado por SDL. La información necesaria podría, por definición, formar parte del conjunto de parámetros de señal, pero esto tendría el inconveniente de que el significado natural de las señales no quedaría entonces claro. En una realización práctica, este modelado no es una cuestión importante.

El núcleo del proceso dominio es el procedimiento procesar PDU (Process_PDU), que invoca la secuencia validar entrada (Validate_input), proveedor superior (Top_provider), y aplicar PDU (Apply_PDU). Cada uno de ellos es una amplia selección de caso basada en el género de PDU de que se trata. Los detalles relativos a cualquier clase dada están por lo general muy claros. Un buen método para abordar estos procedimientos es avanzar horizontalmente siguiendo el curso de las MCSPDU especialmente importantes, como **SDrq**, una por una.

La codificación de las MCSPDU, mediante BER o PER, según la versión de protocolo, se confía por lo general al proceso punto extremo. Todo lo que el proceso dominio necesita saber es el tamaño que puede dar a las MCSPDU de contenido variable, como **DUrq** con sus múltiples identificadores de usuario. Esas consideraciones se expresan mediante enunciados de decisión con texto informal, utilizando constantes del caso más desfavorable para la codificación BER. En la práctica, algunos límites al contenido variable pueden ser previamente calculados en términos que dominio entiende, como el número máximo de identificadores. Otros pueden requerir un conocimiento más directo de la codificación.

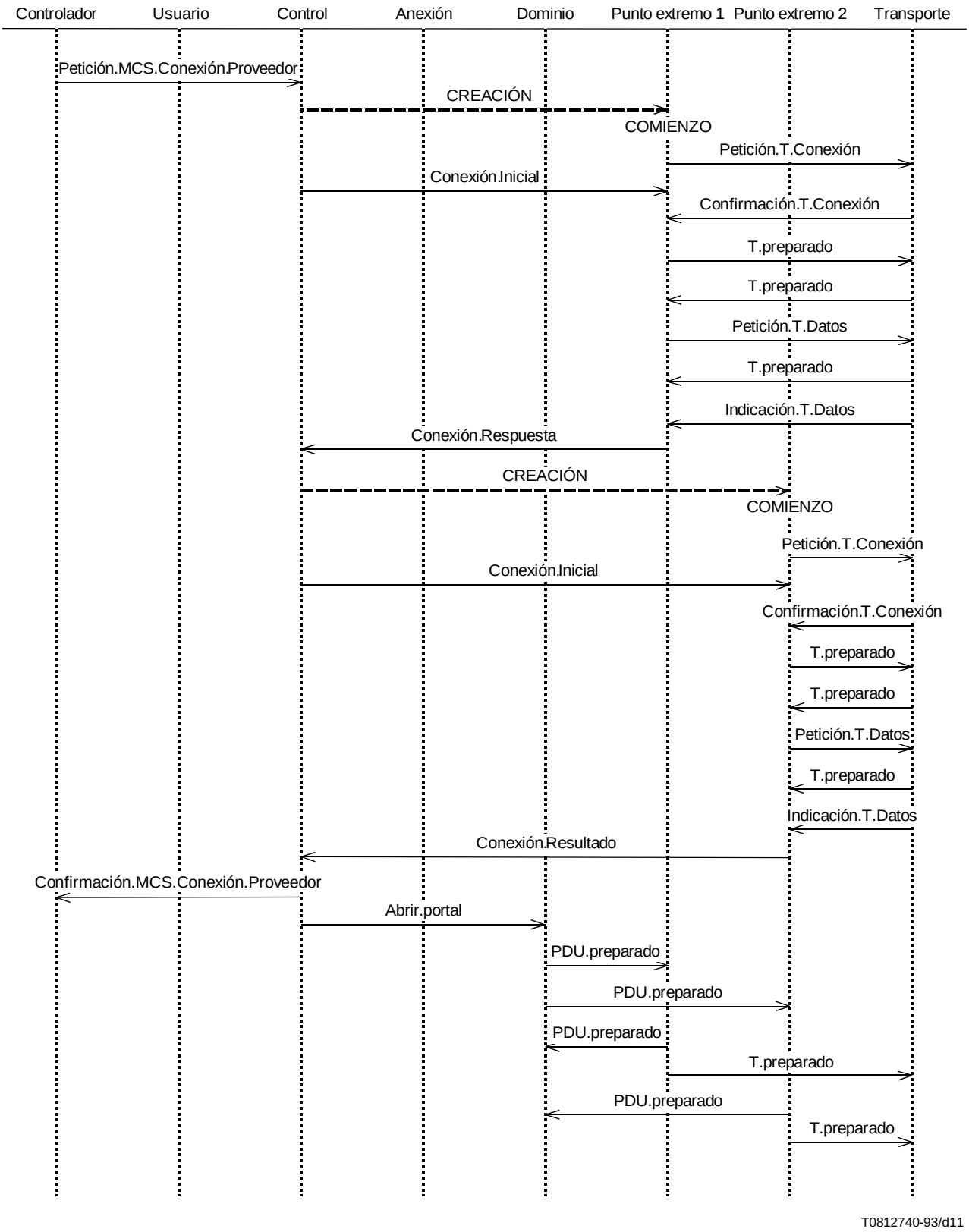


FIGURA VII.1/T.125

Proveedor MCS llamante

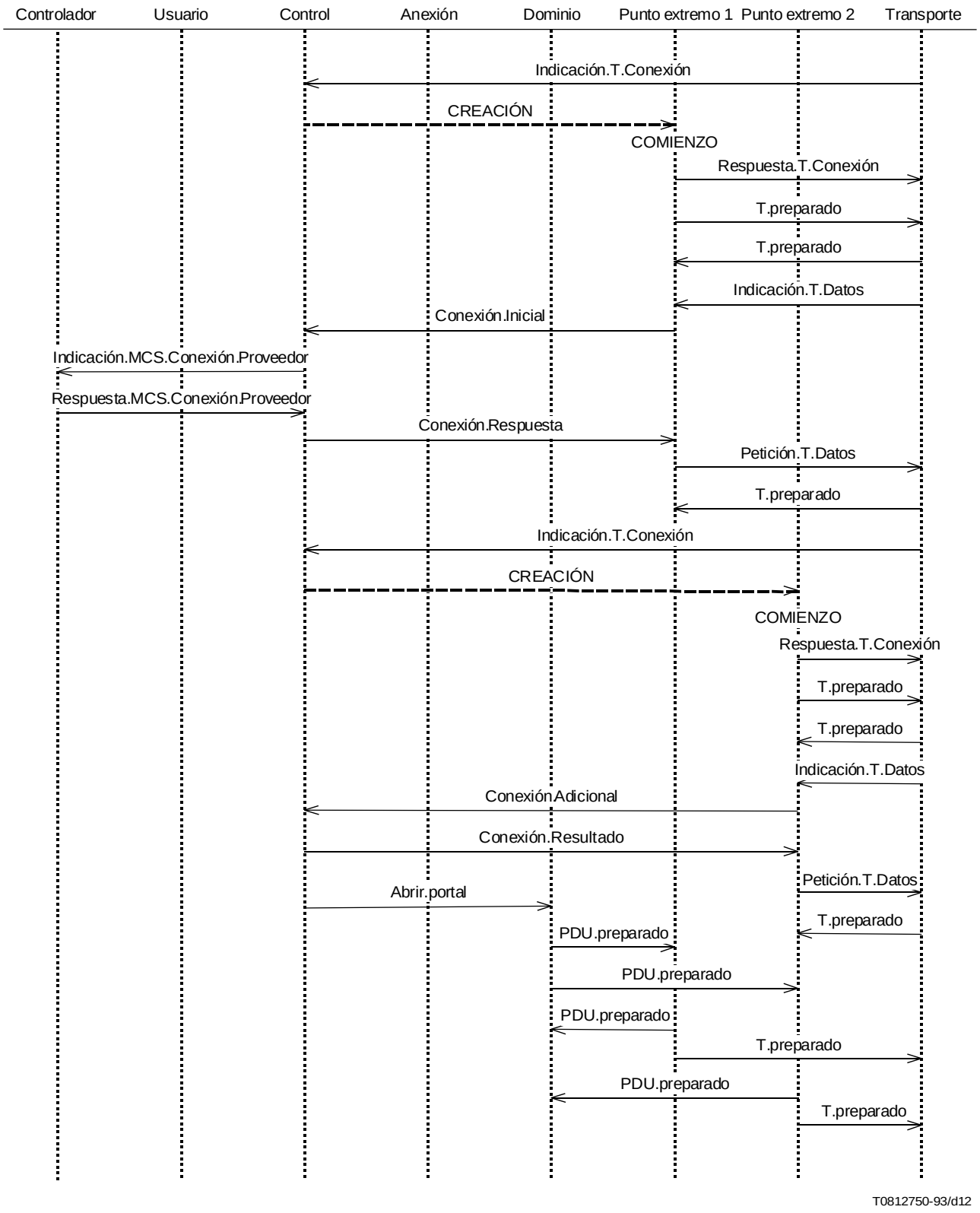


FIGURA VII.2/T.125

Proveedor MCS llamado

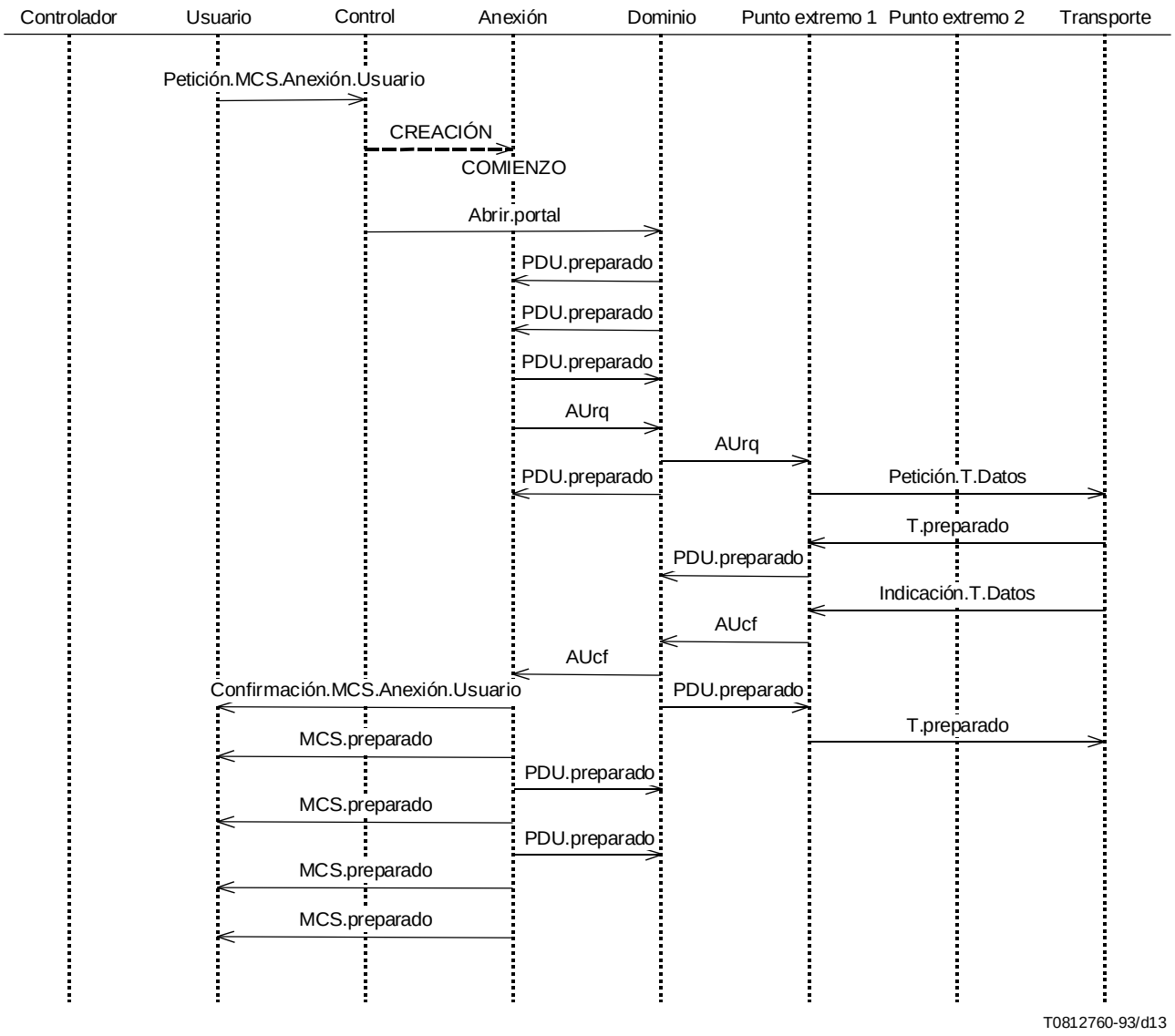


FIGURA VII.3/T.125

Anexión de usuario

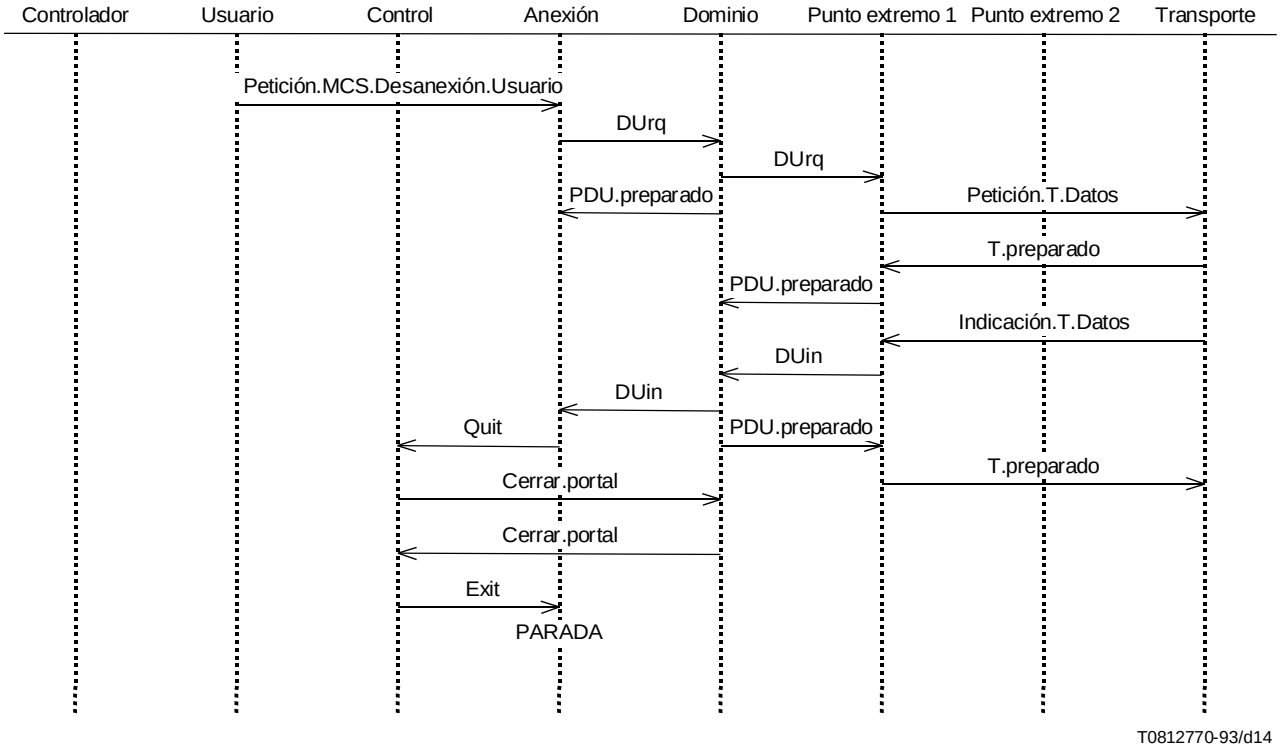


FIGURA VII.4/T.125

Desanexión solicitada por el usuario

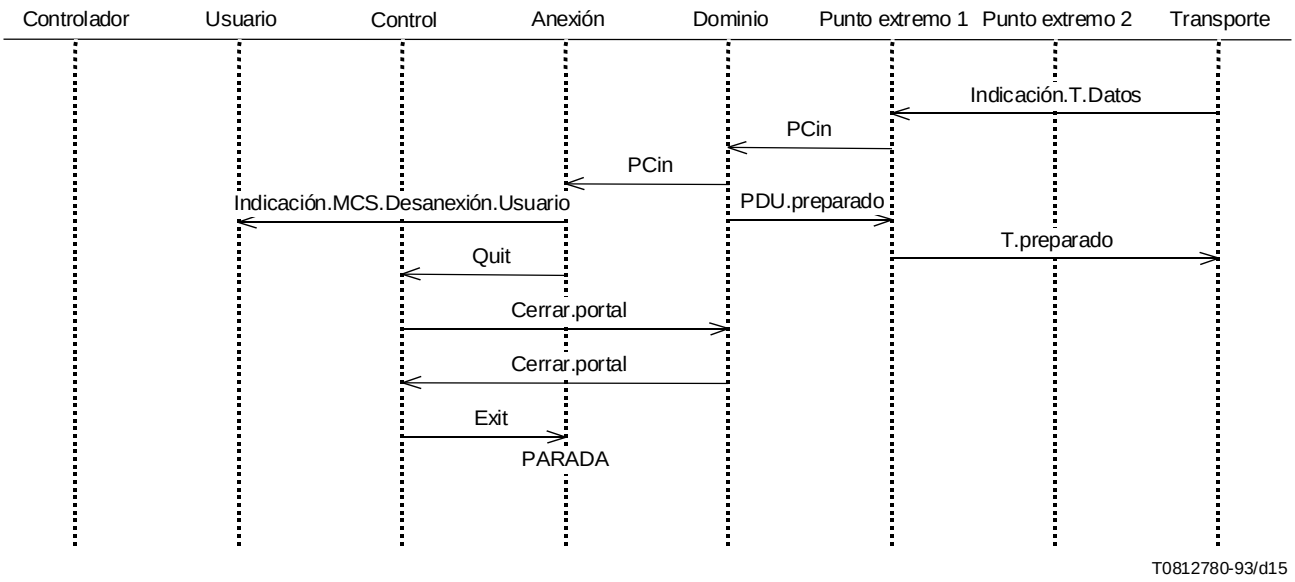


FIGURA VII.5/T.125

Desanexión iniciada por el proveedor

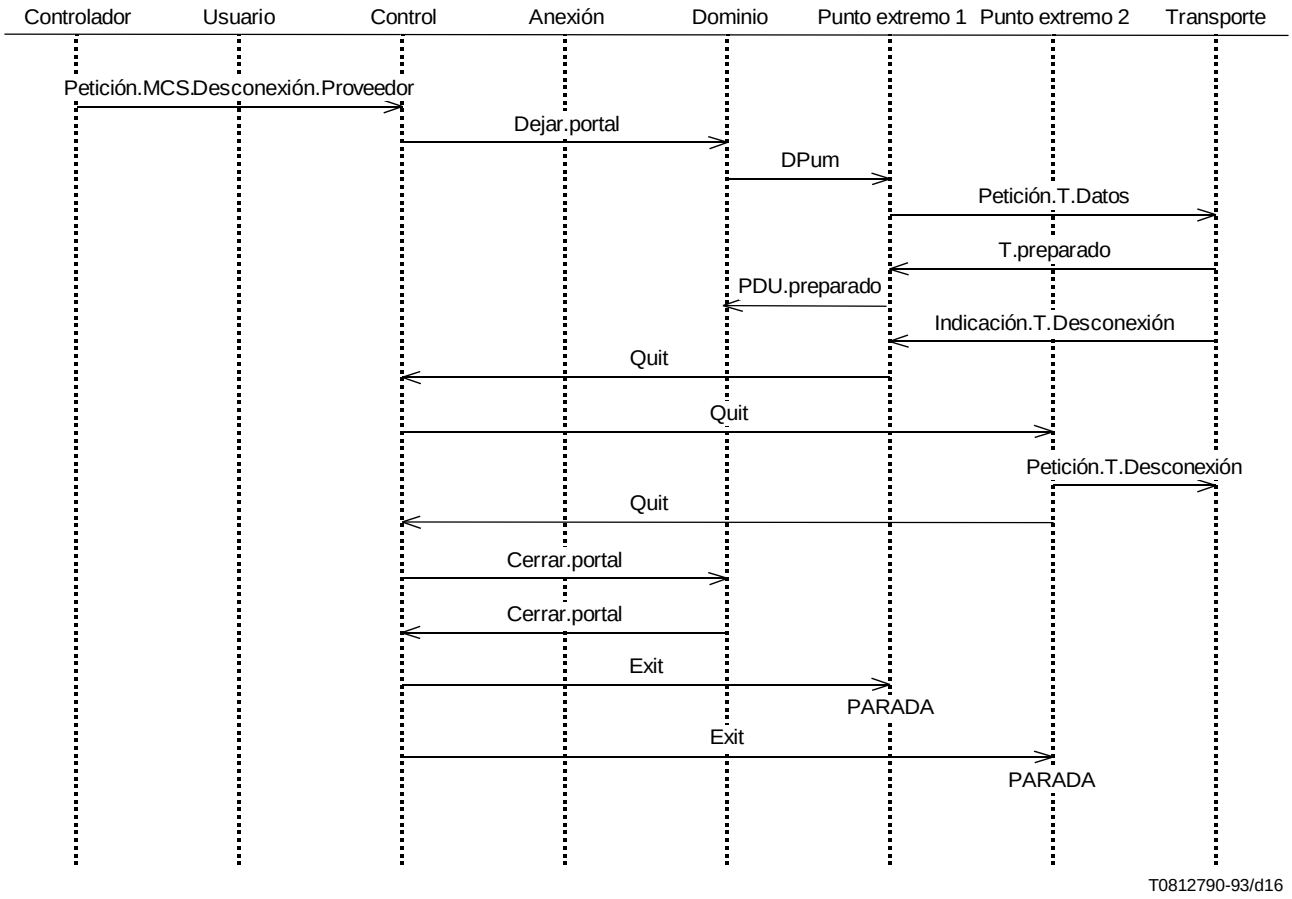


FIGURA VII.6/T.125

Desconexión solicitada por el controlador local

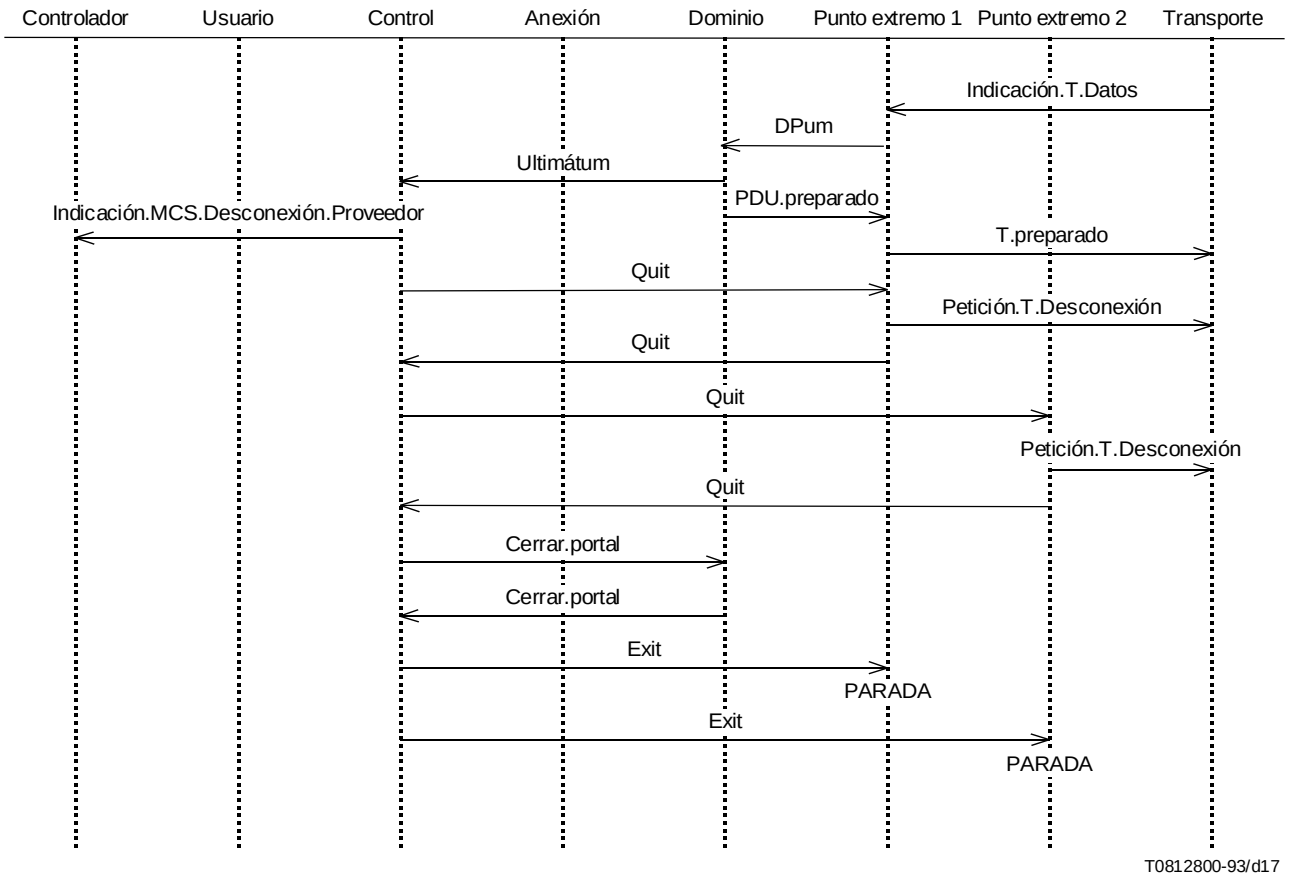


FIGURA VII.7/T.125

Desconexión solicitada por el proveedor distante

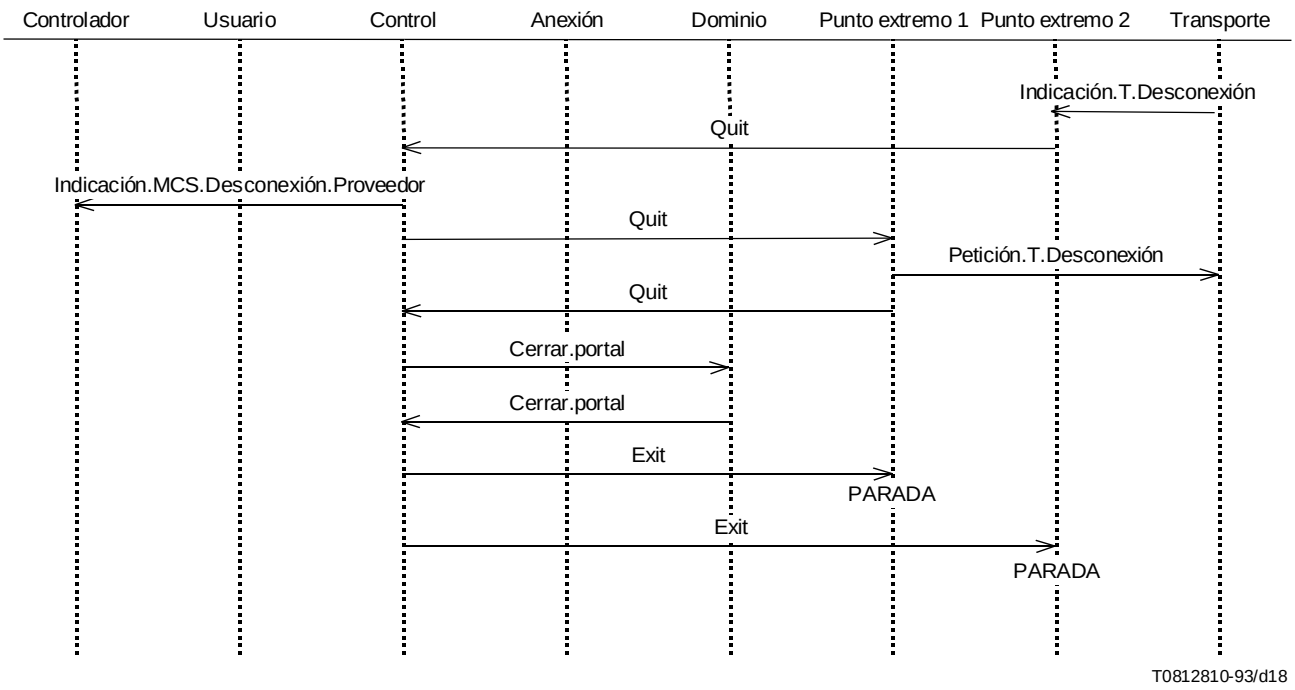


FIGURA VII.8/T.125

Desconexión iniciada por el proveedor