Remote Laboratory Systems Interoperability Standard: Interface Definitions. GOLC

Nicolás Hock Isaza - 200727001010

Fuente: Document URL - GOLC Website

El documento de interoperabilidad entre sistemas de laboratorios en línea (reales o virtuales) **no** establece una *forma* de implementación de servicios. La idea es encontrar servicios (interfaces o mensajes) comunes y estandarizarlos, dejando la implementación a criterio de cada plataforma de laboratorios.

Terminología

Antes de entrar en detalles sobre el documento, es necesario tener claros algunos términos:

Interfaz: Un *canal* de comunicación común entre dos sistemas de laboratorios (un *consumidor* y un *proveedor*) por el cual se pueden comunicar.

Consumidor: Una organización que ha negociado (tiene acceso) con un *proveedor* a nombre de sus propios usuarios (los del consumidor).

Grupo Consumidor: Un grupo de usuarios que hace uso de un acuerdo entre un *consumidor* y un *proveedor*. Es importante resaltar que este grupo tiene unos permisos específicos en un contexto específico.

Usuario: Una persona *identificable* en un laboratorio en línea. Es importante resaltar que los *proveedores* autentican al nivel de *consumidor* (no de usuarios

individuales). Es responsabilidad de los *consumidores* autenticar al usuario y certificar los grupos a los que pertenece. El proveedor solo necesitará la información del usuario individual para estadísticas y análisis de uso del laboratorio.

Proveedor: Una organización que comparte (o hace disponible) sus laboratorios en línea (algunos o todos) para los *consumidores*.

Rig y RigSet (Rig, Tipo de Rig y Pools): Un *rig* es una instancia de un *aparato* (equipo de laboratorio) sobre la cual se puede ejecutar un experimento.

Un *RigSet* puede ser un conjunto de *Rigs* y contener más *RigSets*, creando un grafo dirigido donde las hojas son, probablemente, *RigSets* con un solo *Rig*, esto se necesita pues no todos los laboratorios se comportan igual. Un estudiante puede elegir un *rig* específico solo por la calibración (y consistencia en la calibración), mientras que en otros experimentos se podría utilizar cualquier *rig* disponible.

Principios del estándar

En la sección de principios se definen 6 principios sobre los cuales se basará el estándar.

Principio 1

El **primer principio** define que el estándar definirá un *baseline*, que es la interfaz mínima que debe ofrecer un sistema de laboratorios para implementar el estándar. Se define también que se pueden crear interfaces más amplias para funcionalidad específica de un laboratorio, pero, como no todos tendrán la misma interfaz, debe existir un servicio que permite preguntar por la interfaz del laboratorio.

Creo que este principio es muy válido, pero solo mientras se mantenga la parte de "preguntar por la interfaz". No tipo WSDL, pero los consumidores que se conecten a los laboratorios, deberían ser capaces de obtener la información

sobre las funciones que ofrece el laboratorio.

Es más, los consumidores deberían poder **interactuar** con los laboratorios, solo conociendo lo básico de la respuesta del servidor. Esto, conocido como HATEOAS (*Hypermedia as the Engine of Application State*). Esto implica que los clientes interactúan con la aplicación (los laboratorios en este caso) **completamente** por medio de ***Hypermedia* brindada por las aplicaciones**.

Para mí, *HATEOAS*, es una idea impresionante. Es **muy** escalable (así funciona WWW) y pueden interactuar con los laboratorios conociendo solo **una** URL dentro del laboratorio. A partir de ahí, todas las interacciones se hacen por medio de *links* encontrados en la respuesta del servidor. Para verlo de una manera más clara, si yo entro a una página de internet, como la de EAFIT, solo debo saberme la dirección principal. Después, toda la interacción que yo tengo con la página (como mirar un Pénsum de una carrera) lo hago basado en las respuestas de la primera petición. Así, no importa si las URLs internas de EAFIT cambian, mi consumidor, siempre podrá ir al Pénsum deseado (siempre y cuando las respuestas tengan el camino).

Para lograr esto, es necesario modelar las interacciones como una máquina de estados finitos. Donde las transiciones entre estados se hacen al recibir peticiones desde los *consumidores*.

Principios 2 & 3

El **segundo y tercer principio** indican que el estándar deberá brindar interoperabilidad entre sistemas que hayan adoptado versiones diferentes del estándar (segundo). Y que el estándar debe ser compatible con versiones futuras (*upwardly compatible*) cuándo sea posible.

Yo no estoy completamente de acuerdo con esto (especialmente con el segundo). Creo que es muy importante brindar **cierta** compatibilidad entre versiones, pero no total. A largo plazo mantener versiones muy antiguas es muy costoso.

Un buen mecanismo para esto, es mantener un sistema de versionamiento que

permita reconocer cuáles versiones son compatibles. Un ejemplo es **Semantic Version**. La forma más fácil de explicar *SemVer* es que cada versión está compuesta por 3 números: **Major.Minor.Patch**.

- El primer número (*major*) indica la versión 'principal' del estándar. **Cualquier aumento en este número, implica que se han introducido cambios que no son compatibles con la versión anterior**.
- El segundo número (*minor*) indica la 'sub-versión' del estándar. El incremento en este número indica que **se han agregado funciones pero todas son compatibles con la versión actual del estándar**.
- El tercer número (*patch*) indica que no se ha agregado nada nuevo, pero se **han corregido bugs** encontrados en la versión actual.

Utilizando *SemVer*, un consumidor 3.2.1 puede comunicarse con un proveedor 3.5.3 sin ningún problema. El "*minor version*" permite evolucionar el estándar sin afectar los clientes. Si se agregan cambios incompatibles, se cambia el "*major version*" y preferiblemente, se da soporte para las **últimas dos versiones major** y no para todas las que ha habido.

Esto también implica que uno puede depender de la versión `~> 3.2.1` que quiere decir que sirve cualquier versión `3.x.x` mayor o igual a `3.2.1` pero menor a `4.0.0`.

Principio 4

El **cuarto** principio determina que el estándar debe brindar diferentes opciones a los proveedores de acuerdo al *nivel de servicio* que desean brindar (tanto entre varios sistemas como al interior del propio proveedor). Ponen algunos ejemplos como:

- Varios niveles de garantía de disponibilidad del laboratorio.
- La disponibilidad de algunos datos recogidos al usar el laboratorio, como que hardware y software se utilizó en la sesión. Es importante resaltar que estos datos son **complementarios** y no son los resultados de los

experimentos.

Personalmente me parece fantástico este punto. Cada proveedor del servicio debe tener autonomía para decidir qué presta y con qué calidad lo presta. Forzar un nivel de servicio muy alto sería una barrera muy grande para que nuevos miembros ofrecieran sus laboratorios.

Adicionalmente se han definido mensajes para preguntar el nivel de servicio de un proveedor.

Principio 5

El **quinto** principio asegura que el estándar proveerá **autenticación y autorización** para consumidores, grupos y usuarios, basados en estándares actuales. El estándar que se utilizará para esto debe soportar autenticación tanto a nivel de consumidor (entre universidades) como por grupos de consumidor (a nivel de usuarios). Esto es indispensable para poder asegurar el principio 4 (del nivel de prestación del servicio).

Creo que actualmente el estándar que se debe utilizar es ***OAuth 2.0***. Hay mucho trabajo y dinero en mejorarlo y tiene el apoyo de las empresas más importantes de internet.

Perfiles e Interacciones

En esta sección el estándar define las interacciones básicas que un Proveedor debe soportar. Se define sólo el perfil básico pero se mencionan "futuros perfiles".

Perfil Básico

Este perfil define la funcionalidad mínima requerida para que un consumidor pueda utilizar un laboratorio (interactivo o *batch*) de un proveedor.

Conocimiento del Sistema (*System Enqiry*)

Se definen diferentes mensajes para diferentes interacciones, por ejemplo, se da una sección de **conocimiento del sistema** (*System Enquiry*), donde un consumidor puede preguntarle al proveedor por los perfiles que soporta y qué versión de un perfil específico soporta. También se puede preguntar por el nivel de servicio que se puede esperar de ese proveedor.

En esta sección se dan varios ejemplos de uso de estos mensajes. Por ejemplo, se pregunta por un nivel de servicio específico. Dentro del ejemplo se definen atributos por los que uno puede preguntar (como *ServiceAvailability* o *InformationRetention*). Personalmente pienso que hay un problema con esto (no con el ejemplo, sino con definir las cosas dentro del ejemplo). Si se van a definir los atributos por los que puedo preguntar, deben ser una sección específica del estándar.

Pero, adicionalmente a esto, me parece que esta definición **no** debe ser así. Debería existir un *endpoint* donde se pregunte por los atributos por los que puedo preguntar (volviendo a la idea de *HATEOAS*). Con el nuevo *endpoint*, yo podría tener una respuesta así:

```
attributes: [
    {
      name: "Service Availability",
      endPointQuery: "QueryServiceLevel(Attrib='ServiceAvailabil
    },
    {
      name: "Information Retention",
      endPointQuery: "QueryServiceLevel(Attrib='InformationReten
    }
]
```

Esto sería aún mejor si cada atributo tuviera su propio *endpoint*, entonces la respuesta podría ser algo como:

```
attributes: [
    {
      name: "Service Availability",
      url: "https://example.com/service-availability"
```

```
    },
    {
      name: "Information Retention",
      url: "https://example.com/information-retention"
    }
]
```

Donde los parámetros se pueden mandar en el *request* de HTTP. Nuevamente, las ventajas que esto genera es que si el proveedor cambia de tal manera que, por ejemplo, un servidor atiende las peticiones sobre el servicio de seguridad y otro servidor sobre la consistencia de los datos, los clientes **no** tienen que cambiar, pues siempre han leído el "lugar a preguntar" de la respuesta del servidor.

Esta observación se mantiene en todos los ejemplos presentados en el estándar, por esto, solo la pondré una vez.

Conocimiento de los laboratorios

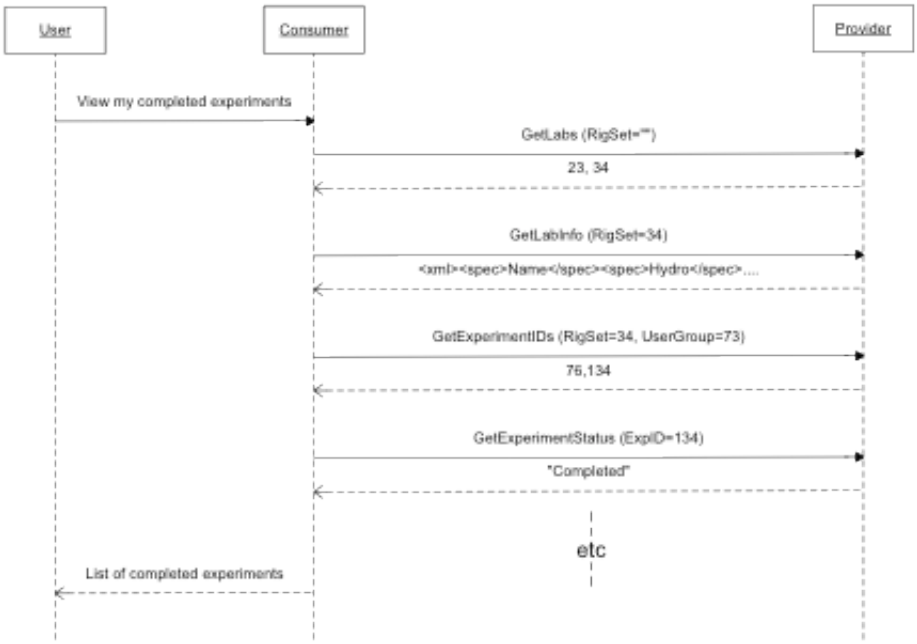
En esta sección se definen interacciones para preguntar sobre el estado de un *RigSet* específico o de un experimento que se ha llevado a cabo en ese *RigSet*.

Se plantean las siguientes interacciones:

- `GetLabs()` - Retorna una lista de los laboratorios en el proveedor. Si se envía un parámetro con un *RigSet*, la lista se limita a los hijos de ese *RigSet* específico.
- `GetLabStatus()` - Retorna el estado de un *RigSet* y si se está funcionando en ese momento.
- `GetLabInfo()` - Para obtener información propietaria sobre ese *RigSet*.
- `GetExperimentIDs()` - Retorna una lista de identificadores para los experimentos asociados con un *Rig* en ese *RigSet* que están activos. Estos experimentos deben estar relacionados a un *consumidor*, *grupo consumidor* o *usuario individual*. Un laboratorio **activo** puede estar en cola, corriendo o completo con los resultados aún disponibles.
- `GetExperimentStatus()` - Provee el estado actual de un experimento

asociado a un `experimentad`. Indica si dicho experimento ya corrió, o está corriendo (y cuánto lleva corriendo) o si está en un estado de espera.

El siguiente diagrama muestra la interacción entre un *consumidor* y un *proveedor* donde se obtiene la lista de todos los experimentos completados.



Personalmente pienso que las interacciones están bien definidas pero tengo ciertas dudas sobre los mensajes como tal. Estas llamadas son muy características de *SOAP* o *RCP*, yo soy más partidario de *REST*. Me parece que si se modelaran los laboratorios y los experimentos como recursos, se pueden tener llamadas más *limpias* y se pueden (deben) aprovechar los métodos de petición HTTP. Por ejemplo:

`GET /laboratories` - Retornaría la lista de *RigSets* y `GET /laboratories/<lab_id>` retornaría la información sobre ese laboratorio específico. Ahora, los recursos pueden *tener* otros recursos y así `GET /laboratories/<lab_id>/experiments` retornaría la lista de experimentos para ese laboratorio.

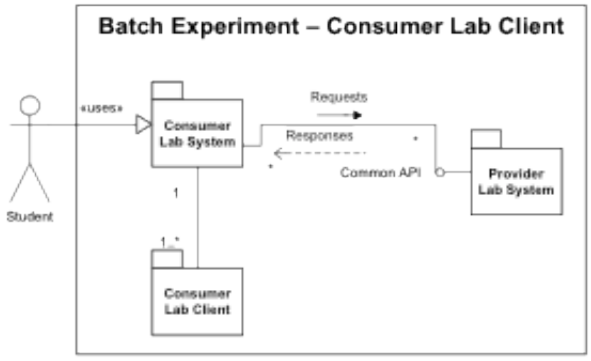
De esta manera, la URL muestra también la jerarquía de los recursos que existe en el sistema.

Laboratorios Batch (o en lotes)

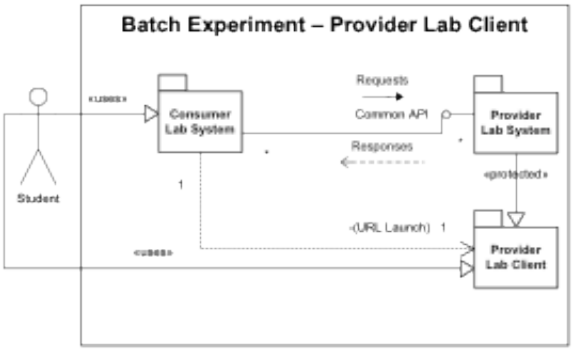
Los laboratorios `match` son aquellos donde los estudiantes configuran el equipo, envían la configuración y se corre en el laboratorio después de pasar por una cola de espera. Se llaman así pues no hay ninguna interacción *mientras* se corre el experimento.

Si un estudiante desea cambiar los parámetros del experimento (por ejemplo, la carga utilizada) debe volver a correr el experimento, pasando nuevamente por la cola de espera.

En esta sección del estándar, se definen dos formas de uso para los laboratorios batch. Una en la que el *consumidor* crea su propia interfaz del laboratorio y otra donde el *proveedor* presta la interfaz. Claramente, si un *consumidor* necesita cierta consistencia visual en todo el proceso, la mejor opción es presentar la interfaz propia, pero este acercamiento puede presentar más problemas de rendimiento y es más difícil de implementar (lo que aumenta el costo). El diagrama ilustra este escenario:



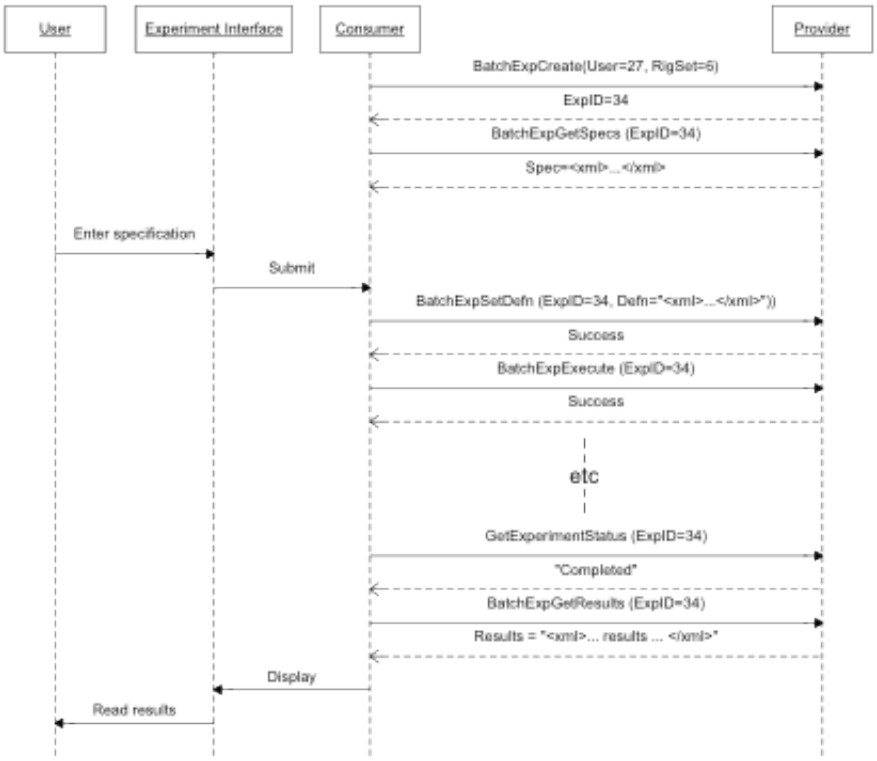
Ahora se muestra un *consumidor* que usará la interfaz del *proveedor*, esto simplifica el diseño del laboratorio pero la interfaz gráfica que verán los estudiantes, posiblemente, no será consistente.



Los mensajes definidos en el estándar para laboratorios *batch*, son:

- `BatchExpCreate` - Crea un experimento nuevo en el proveedor.
- `BatchExpDelete` - Elimina un experimento en el proveedor.
- `BatchExpGetSpecs` - Obtiene la información sobre los parámetros necesarios en el experimento. Tanto requeridos como opcionales.
- `BatchExpSetDefn` - Configura un experimento en el proveedor.
- `BatchExpGetDefn` - Recupera la configuración actual del experimento.
- `BatchExpExecute` - Envía una petición de ejecución de un experimento previamente configurado.
- `BatchExpLaunchProvIF` - Retorna una URL para usar el laboratorio utilizando la interfaz del proveedor.
- `BatchExpGetResults` - Obtiene los resultados de un experimento.

A continuación se muestra un diagrama de secuencias para utilizar una interfaz del consumidor. En el estándar se muestran ambos casos, pero solo pongo la del consumidor pues es la más compleja.



Laboratorios Interactivos

El estándar también especifica las interacciones necesarias para utilizar laboratorios interactivos. Estos laboratorios son en los cuales el usuario puede cambiar (interactuar) con el laboratorio *mientras* el experimento está corriendo. Nuevamente, la interfaz que se le presenta a los estudiantes puede ser creada por el consumidor o el proveedor. En caso de presentar una interfaz del consumidor, el rendimiento debe ser tenido en cuenta (aún más que en los laboratorios match) y se debe intentar optimizar la interfaz para una buena interacción del estudiante con el laboratorio.

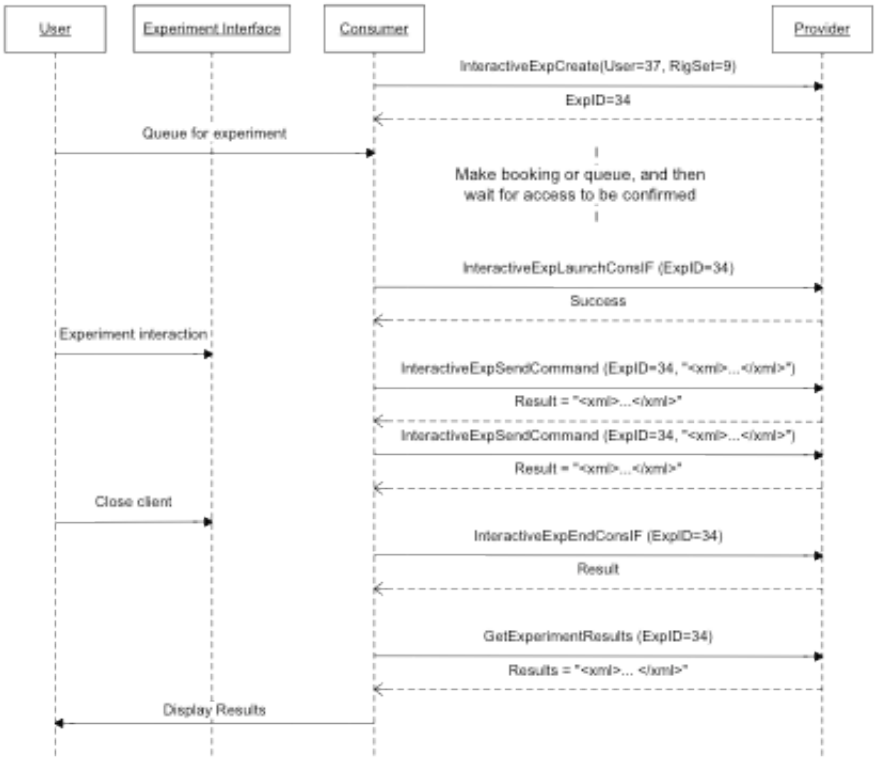
Los mensajes que se definen son:

- `InteractiveExpCreate` - Crea un experimento interactivo en blanco en el proveedor.
- `InteractiveExpDelete` - Elimina un experimento interactivo del

proveedor.

- `InteractiveExpGetSpecs` - Retorna una copia del esquema de interacciones para este experimento. Esto es muy único de cada experimento.
- `InteractiveExpLaunchConsIF` - Crea una petición para iniciar un experimento interactivo con la interfaz del consumidor.
- `InteractiveExpSendCommand` - Envía un XML con los comandos para la interacción con el laboratorio (para controlar el laboratorio mientras corre).
- `InteractiveExpEndConsIF` - Termina un laboratorio que estaba corriendo con la interfaz del consumidor.
- `InteractiveExpLaunchProvIF` - Obtiene una URL con la interfaz del proveedor para este experimento. Si el consumidor no tiene permiso para correr el laboratorio (por ejemplo la reserva aún no ha comenzado o expiró) se retorna un error.
- `GetExperimentResults` - Obtiene los resultados del experimento.

A continuación se muestra un diagrama de secuencias donde se utiliza la interfaz del consumidor para acceder a un laboratorio interactivo. Nuevamente se muestra esta secuencia pues es la situación más compleja.



Manejo del Acceso (Reservas & Colas)

Las interacciones que el estándar presenta en esta sección son funciones administrativas. Estas interacciones soportan la negociación para el acceso a los laboratorios entre el consumidor (o un grupo/usuario) y el proveedor (a un RigSet específico).

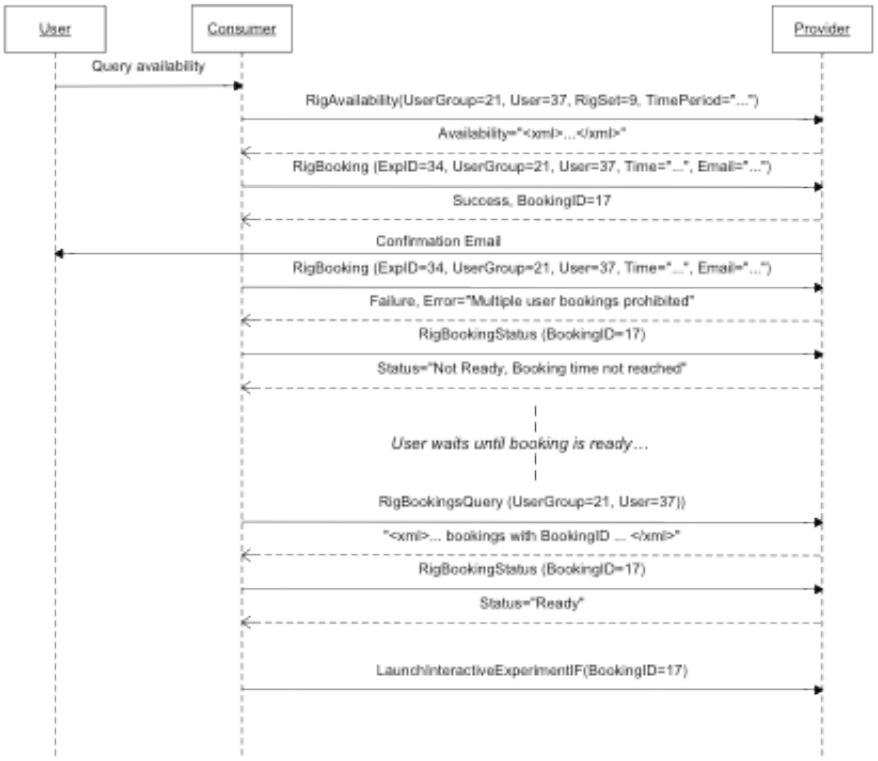
Se presentan dos modelos principales de acceso. Por **colas**, donde el usuario puede pedir acceso y esperar en una fila hasta que el laboratorio esté disponible (normalmente para laboratorios match); y por **reservas** donde el usuario puede separar un intervalo de tiempo con anterioridad y así garantizar la disponibilidad del laboratorio cuándo se va a utilizar. Los mensajes que se especifican son:

- `RigAccessType` - Permite que el consumidor sepa si el laboratorio (RigSet) específico es por colas o por reservas.
- `RigAvailability` - Para un RigSet que sea administrado por reservas,

este mensaje permite preguntar por intervalos disponibles en un rayo específico.

- `RigBooking` - Realiza una petición de reserva para un RigSet específico.
- `RigBookingCancel` - Cancela una reserva.
- `RigBookingStatus` - Obtiene el estado de una reserva. Particularmente permite saber si la reserva se puede redimir o no.
- `RigBookingsQuery` - Obtiene una lista de las reservas realizadas para un consumidor, grupo consumidor o usuario específico.
- `RigQueue` - Agrega un experimento a la fila para ser ejecutado.
- `RigQueueCancel` - Elimina un experimento de la fila de espera.
- `RigQueueStatus` - Obtiene el estado del experimento en la fila. Es la posición de espera en la que se encuentra el experimento.

A continuación se muestra un diagrama en el cual se realiza una reserva.



Personalmente la separación entre los dos escenarios me parece buena. Más

aún, teniendo que actualmente las plataformas de laboratorios en línea utilizan sistemas diferentes para esto. Por ejemplo, en Deusto solo se manejan colas de prioridad, mientras que en iLabs existen reservas y colas.

Esta separación permite tener algo común a "corto plazo" para integrar las diferentes plataformas. Por esto es clave que un consumidor pueda preguntar qué sistema utiliza cierto laboratorio (`RigAccessType`).

Comentarios Personales

Pienso que el estándar es un gran esfuerzo y en la dirección correcta, pero aún cuándo en el documento sale que ha decidido el protocolo (si hacer SOAP o REST), pienso que como está es muy difícil implementar REST.

Si se pueden desacoplar un poco más los consumidores de los proveedores, sería una gran ventaja que permitiría evolucionar las dos partes del sistema de manera independiente. Esto es obviamente un trabajo muy difícil pero creo es que es lo que se debería buscar. Aún sin ser necesariamente guiados por hipermedia, en las respuestas que sea posible se debe buscar esto y buscar en un futuro ser guiado siempre por las respuestas del servidor.

El versionamiento también me parece un punto muy importante. Poder calcular automáticamente qué versiones son compatibles con otras sería otra forma de desacoplar sistemas. Así, si un proveedor no es compatible con un consumidor, puede "sugerir" otro proveedor que tenga esta misma versión. Este puede ser un experimento interesante tomando como ejemplo DNS.

En cuanto al uso de XML (y SOAP) para las llamadas, me preocupa un poco que limite el uso de Javascript. Actualmente no se pueden hacer llamados (GET sobretodo) a servidores diferentes desde Javascript. Hay muchos "trucos" para esto pero son situaciones para saltarse el problema, no lo arreglan. Existe algo llamado JSONP, que busca, de alguna manera más limpia, resolver este problema. La forma como funciona es que la petición se hace y se envía una función `callback` que se ejecuta con la respuesta. Esto es un punto importante para analizar ya que si se pueden hacer llamados y cálculos desde el

navegador de los estudiantes, los servidores perderían un poco de complejidad y se desacopla aún más consumidor y proveedor. Sobretodo cuándo se quiera utilizar una interfaz de un consumidor para los experimentos. Y no sobra decir la gran cantidad de dinero invertida en HTML5 y Javascript para crear aplicaciones interactivas pesadas en el browser.