Service Location Service

Hal Hildebrand

Problem Statement

Currently there is no real standard service which provides the system wide ability to discover, manage and maintain groups of services which reside in different processes, on different host machines in a large distributed system. Manual configuration is always possible, but this is an extremely brittle option that requires a great deal of effort to maintain - not to mention the escalating effort required to manage these manual configurations across multiple data centers and systems. Static configuration also makes it hard to dynamically maintain the services within the system after startup. Processes may come and go, requiring additional protocols for discovery and death detection. This service which provides service location, discovery and lifecycle services must itself be a robust, easy to configure subsystem which is reliable and continuously available.

Proposed Solution

The proposal is to provide a solution in the form of a *Service Location Service* (here after referred to as *SLS*). This service provides a simple, high level API that any process in the system may use to

- discover other services within a large number of processes
- discover how to connect to these service providers
- monitor the lifecycle of the individual service providers
- provide describing metadata for services that clients can use for discovery

This service is layered on top of a more general group membership and state dissemination service, which provides the continuous availability and robust communications required to implement the SLS.

Service Location Protocol

The service reuses the concepts pioneered by the Service Location Protocol (SLP). The main concept we're reusing from SLP is the concept of a Service. In SLP, a service is described by two pieces of information

- The Service URL
- The Service Attributes

The Service URL is of the form: **service:printer:lpr://myprinter/myqueue**. These URLs follow a particular form so that the services they represent can be queried by clients.   The Service Type of this URL consists of the first three components of this URL: **service:printer:lpr**. The first two components (**service:printer**) are called the abstract *Service Type*. This allows clients to query the SLS for all services which match this Service Type - i.e. find all printer services, regardless of the protocol used to connect to them.

The Service Attributes are simply name value pairs. For example, the attributes of the printer service could be:

```
(printer-name=Hugo),
(printer-natural-language-configured=en-us),
(printer-location=In my home office),
(printer-document-format-supported=application/postscript),
(printer-color-supported=false),
(printer-compression-supported=deflate, gzip)
```

The SLS will provide clients several different types of queries to discover services. A client can search for all services with the same service type or abstract service type - e.g. search for all service:foo services within the system. Queries on a service type can also be combined with a query on the attributes of the service, using LDAP's query language - e.g search for all **service:foo** services where **group=A** and **load <= .25**.

OSGi Service Model

The standard API to the SLP is archaic and a bit hard to understand. What we wish to provide is a simple, yet powerful model which meets the requirements of both the clients and providers of a service. Rather than reinvent such an API, the proposed service adapts the existing OSGi service API and combines it with SLP to provide a simple, clear API. This API is composed of several different actors:

- **ServiceURL**
  The URL representing the service, following SLP definitions
- **ServiceReference**
  The structure representing a service registration
- **ServiceRegistration**
  A handle representing a particular instance of a service
- **ServiceScope**
  The context in which services are registered and queried
- **ServiceEvent**
  Lifecycle events that clients receive regarding the services they have subscribed to

API

The ServiceScope is the entry point into the service from both the client and service provider's point of view. The ServiceScope provides the API to:

- Register services
- Remove services
- Modify services
- Query Services
- Register ServiceListeners

A service provider registers the description of the service with the ServiceScope. To register a service, the Service URL and the service properties - in the form of a Map - are provided to the scope. The scope registers the service and returns a service registration handle in the form of a **UUID**. Using this service registration handle, the service provider may then use the ServiceScope to remove the service, or update the properties of the registered service.

UUID register(ServiceURL url, Map<String,Object> properties)
void setProperties(UUID registration, Map<String, Object> properties)
void unregister(UUID registration)

Clients can issue queries against the ServiceScope to retrieve a list of ServiceReferences which match the supplied query. Two query methods are supplied, one which simply takes a string describing the service type, and one which takes a service type and a string representing the LDAP query for matching against the properties advertised by a service instance. The returned reference provide the ServiceURL that these clients can then use to connect to the services.

ServiceReference getServiceReference(String serviceType)
List<ServiceReference> getServiceReferences(String serviceType, String query)

Clients can also register ServiceListeners with the ServiceScope so that they can be notified of the lifecycle events which match a query.

void addServiceListener(ServiceListener listener, String query)
void removeServiceListener(ServiceListener)

When registering a ServiceListener, the query is provided which allows the listener to constrain the set of service lifecycle events to only those that match the given query. For example, the query can be "**service:foo** where **group=A**", in which case the service listener will only receive lifecycle events for FOO services that advertise that they are in Group A.

Service Lifecycle

Services have three lifecycle states that listening clients can be informed about:

- Registered
- Modified
- Unregistered

When a service is first registered with the ServiceScope, listening clients with matching queries are notified of of the registration of the service. Services which are registered are considered by clients to be ready to provide their service. After the initial registration of the service, the attributes of a service may then change. Note that the ServiceURL of a service never changes throughout the lifetime of the service. Rather, the attributes of the published service can be modified by the service as needed. Clients that are subscribed to these services will then be notified of these modifications. Finally, a service may be removed from the scope. When a service is removed, subscribing clients are notified of the unregistering of the service.

Semantics

The SLS is robust. By robust, we mean that all events for a given service instance are reliably delivered to all clients in the order they occur. These service events are delivered in the presence of failures of other process within the system, and are delivered irrespective of partitioning events within the communication topology of the system. This allows clients to be confident that they have a correct view of the global service state of the system and make local decisions regarding this state. Advertising services can likewise be confident that their advertisements and lifecycle events are seen by all clients, and seen in the correct order.

Supporting Technology

SLS is a layer built upon a system called Anubis. Anubis is part of HP's SmartFrog distributed configuration and management system. Anubis is a partition-aware tuple-space that can be used to share information reliably between hosts on a single site. Each host is notified of detected partition change events, as well as tuple insertion and removal. Add a tuple into the space, wait a heartbeat and everyone else knows it - unless a partition event has occurred, implying that at least one machine is isolated or has failed.

While similar to group systems such as JGroups and Totem, Anubis is a timed-asynchronous distributed system. This means that Anubis uses a time aware failure model as the basis of a group membership system, rather than vector clocks and a virtual synchrony model. As such, Anubis is itself a rather simple system to understand and reason about. Anubis can operate in two modes: a) timed mode b) ping mode. In the timed mode, Anubis leverages an NTP (Network Time Protocol) synchronization of the clocks within the system for a solid notion of timeliness. When used with NTP in this mode, Anubis can also provide a (near) total ordering of messages within the group by providing a partition-wide time reference. What this means is that a causal ordering of messages provides clients with the ability to order events reliably in time - e.g. Server A failed at Time T1, Servers B, C, D joined at Time T2, etc. Further, because this time reference is partition wide, these events are totally ordered - i.e. every member of the partition sees the same events in the same order. These powerful distributed semantics are leveraged by the SLS to provide a simple, reliable and robust service.

Configuration

ServiceScopes are configured with a state name that represents the state tuple of the scope within the Anubis partition. This state name uniquely identifies the scope. Anubis is an easy system to configure, needing only to define the unicast or multicast address, port and time to live. Additional protocol parameters defining, for example, the "magic number" which identifies the partition allows multiple Anubis services within the same system.

Clients of the SLS obtain the ServiceScope via a Spring configuration context. All the parameters for defining the Anubis service - e.g. multicast/unicast address, port, ttl as well as magic number, protocol type, protocol parameters - are configured through the Spring context.