

# Connecting Producers and Consumers

Niclas Nilsson  
Activa Sweden AB  
[niclas.nilsson@activa.se](mailto:niclas.nilsson@activa.se)

This is a position paper for the OOPSLA 2003 workshop "Pervasive Computing; going beyond Internet for small screens".

## **Introduction**

Since the call for papers mentioned that *"The goal is to identify recurring architecture themes and patterns typically used to build such [pervasive] systems"*, I'll try to compare a few ways to connect Producers (code that creates/publishes data) and Consumers (code that read data). Connecting producers and consumers is a frequent activity in many types of applications, but in most smaller, embedded systems I have worked with, there seems to exist a nice tendency to really divide the large application into several smaller, reusable services that either produces data or consumes data (even though there of course exists some services that both produce and consume).

The recurring question in such systems is how to connect the consumers and the producers. In order to have a flexible system, you would like them to know as little as possible about each other so that you can connect your consumer to another producer when the application/environment changes with as little code change and other hassles as possible. This is even more true if your application shall exist in many different environments.

## **The example**

Let's say I want create a navigation application targeted for the car industry. The navigation application will need to get position information to be able to give the driver directions of how to drive. This application will be the consumer in this example. But how does the application get the position information? Can we be sure that there will be a GPS in all cars? No, the cars may very well have different kinds of positioning equipment. Some cars may have GPS, some may use a mobile phone to get positions (MPS), and some may only use the information from each wheel to do dead reckoning. And some cars may have more than one way of determining the position. Of course our navigation application should be able to perform with any of these cars, but we rather not have different versions for each brand.

So, we do not want our application to know anything about the position producing hardware. It makes sense to wrap these producers in one or more services. To make it easy for the consumer, all position producing services needs to use the same interface so that our application does not have to change. But in some cases there may be several position producing services (since each may represent a physical device). How do we know which producer to connect the consumer to? This is something that we may be better to choosing at deployment time than during development time. If so, we have to have a mechanism to make these choices in runtime.

We basically have three choices of how to connect consumers and producers:

1. To let the consumer choose its producer.
2. To let the producer find its consumers.
3. To have some other way to connect consumers and producers without themselves making the decision.

## ***The environment***

I chose the OSGi service platform for this example. I have seen similar solutions in several other systems, but since this is an open specification (and I for several reasons like these solutions) I thought this would be a good choice for investigating a very common problem.

As you may know, the OSGi service platform specification is a specification targeted mainly to the pervasive computing market. The specification is Java-based and makes heavy use of things like classloaders, Java 2 Security, reflection and such to create an environment suitable for create applications built up from several smaller services.

The OSGi service platform (<http://www.osgi.org>) is a specification supported by a large number of companies (see <http://www.osgi.org/about/members.asp>). Some of these companies are explicitly in the vehicle industry (i.e. BMW and Bombardier) which is a pervasive market expected to grow significantly in the future.

The OSGi service platform specification came out in its third release earlier this year (R3, which can be found on the OSGi website) and one of the additions is the Wire Admin Service which combined with the combined with the “Measurement and State specification” and the “Position specification” makes it very easy to write code using consumers and producers that are easily connected very late in the game (during the configuration/deployment of the system, and not during the development of the system).

I will in this paper compare three ways of doing connections between consumers and producers in an OSGi environment.

## **OSGi Service Registry**

The OSGi service platform specification has a central feature called a service registry. The service registry is the place where you register your own services when they are started and where you fetch handles to other services to be able to interact with them. The nice thing with this service registry is that if your service is using another service and that service goes away for whatever reason (currently being updated, not useful in current context, ...) your service/application will be notified that the service you are using is gone.

When you use some kind of service in a “normal” Java application, the common way is to create or get an object representing the service and register yourself as a listener to some API to get calls whenever something happened. This listener is usually added early and the listener is seldom removed. These types of applications tend to be more monolithic than applications built upon separated services.

Most services that can be listened to have a service API:s that contain a register-method, an unregister-method and then you have to implement the methods in the callback/listener API, which will be called when some even happens. The problem is

that you will seldom be noticed if this service needs to be unloaded, restarted or something else that will affect its availability. The service is often regarded as a part of the application instead of something separate that could serve several applications at the same time. In an OSGi environment (as well as in many other pervasive systems), there is a set of system services that perform the basic stuff, and upon these you build your user applications. When these underlying services are developed, you do not necessarily know which applications will exist in the system. You only know that if you have a positioning device, you are likely to produce positions to applications that need position datums.

Let us look at how these three approaches can be implemented

### ***Solution 1 – Let consumers find its producers***

First, let's introduce two interfaces.

```
public interface PositionListener {
    public void positionUpdated( Position p );
}
```

And:

```
public interface PositionService {
    public void addPositionListener( PositionListener l);
    public void removePositionListener( PositionListener l);
    public Position poll();
}
```

To do the most traditional Java-trick to connect consumers to producers in an OSGi environment, the consumer needs to ask the service registry for a PositionService. If the consumer is lucky there is one PositionService and it will get a handle to the service. If it is really unlucky, there may be zero PositionServices, but also the result of two or more PositionServices makes it tricky since we have to choose one of them somehow. In OSGi there is metadata connected to each service that can describe things that can make the choice easier, especially something called Persistent Identity (PID) which is used as a unique way of identifying one particular instance of a service type.

So, the consumer asks the service registry for a PositionService and filters out which one to use if it gets several services. The consumer then calls `addPositionListener()` to add itself to the producer. Whenever a new position is available, the producer will call all its consumers (listeners) `positionUpdated()`-method with the new data. If the consumer prefers to poll for data, it can use the `poll()`-method instead of adding itself as a listener.

So far, so good. This has been proved to work in many (most?) Java applications and should also work here, right? Yes it does, but it has drawbacks. When a PositionService needs to be updated and is taken down (which services can be during runtime in an OSGi system without restarting the entire system), all consumers must listen to the event from the service registry that the service it is using will go away and the consumer needs to release all resources used from that producer. If it doesn't the update will still take place, but then we may have resource problems since we have consumers hindering garbage collection of the old objects. This means that all consumers must behave correctly for the producer to be able to

be updated in a nice way. This may be the case, but most likely some consumer will fail to do this from time to time. Since one single consumer should not be able to destroy for other consumers, this approach is generally avoided within the OSGi community. It can be found in the earliest API:s like the Log Specification, but is now avoided.

### ***Solution 2 – Let producers find its consumers***

We basically just said that we trust producers (often more low level system services) than consumers (often third party applications) when we aim for a stable 24/7 system. To solve this and at the same time move some complexity from the third party applications to the lower level system services, the OSGi community turned the concepts upside down and introduced what is called “the whiteboard approach”.

The whiteboard approach means that when a consumer wants to consume, it registers itself in the service registry. For example, our navigation application can implement the same PositionListener interface, and register itself in the service registry. The producer (the PositionService) will listen to the registry and be notified whenever a PositionListener is added or removed. When a new position is available, the PositionService will call all PositionListeners `updatedPosition()`-method with the new data. The producers knows all its consumers, but the consumers have no connection to the producer.

How do we solve the problem with several producers then? Well, it is possible for the consumer to specify some metadata when registering (much in the same way that the producers registered extra data in the previous solution) which each producer can look at when it decides if it should send data or not to a particular consumer.

Drawbacks? Structurally, certainly less than in the former solution, but one drawback is that we just lost the poll method. If the consumer can't find the producer, it is definitely hard to poll for data. Sure, it can listen and throw away all uninteresting data, but that is a waste of precious resources.

### ***Solution 3 – the Wire Admin Service***

OSGi service platform specification is now up to release 3, and this release introduces a very flexible way to connect producers and consumers to each other. The new service is called the Wire Admin Service. This is a way to make the coupling between consumers and producers lesser than in both of the other solutions. The optimal low coupling scenario is when the consumer only knows what type of data it wants to consume, the producer knows what type of data it produces, but neither of them know anything about the other.

The OSGi Wire Admin Service solves this problem in a nice way. The Wire Admin Service is the “switchboard” with wires, which determines which consumer will be connected to which producer. The consumer implements a Consumer interface which looks like this:

```
public interface Consumer {
    public void producersConnected( Wire[] wires );
    public void updated( Wire wire, Object value );
}
```

And the producer implements another interface that looks like this:

```
public interface Producer {
    public void consumersConnected( Wire[] wires );
    public Object polled( Wire wire );
}
```

The WireAdmin interface is used to create all these connections between consumers and producers. OSGi suppliers are likely to create nice (graphical) configuration tools to define these connections, which will give the input needed by the Wire Admin service. The Wire Admin service has an interface that looks like this:

```
public interface WireAdmin {
    public Wire createWire( String producerPID,
                           String consumerPID,
                           Dictionary properties );
    public void deleteWire( Wire wire );
    public Wire[] getWires( String filter )
        throws InvalidSyntaxException;
    public void updateWire( Wire wire,
                           Dictionary properties );
}
```

To create a wire between the navigation application and the position producer, a call to the WireAdmins `createWire()`-method is made with the consumer and producer PIDs as arguments. This will create a Wire object, which represents the link between them, but also is what gives them possibility to communicate and still being totally decoupled from each other.

A Wire has a more extensive interface, but for this discussion, we only have to focus on the `poll()`- and `update()`-methods.

```
public interface Wire
    public Class[] getFlavors( );
    public Object getLastValue( );
    public Dictionary getProperties( );
    public String[] getScope( );

    public boolean hasScope( String name );
    public boolean isConnected( );
    public boolean isValid( );
    public Object poll( );
    public void update( Object value );
}
```

The `consumersConnected()`- and `producersConnected()`-methods on the Consumer and Producer interfaces are called when either service is registered and whenever a new connection is created or lost (there may be several wires between the same producers/consumers if there is a reason). Both the consumer and the producer will always know its current set of wires. When the Producer has a new value, it calls the update method on each Wire. Each Wire object calls the updated method on its connected Consumer.

If a Consumer needs to poll for data, it merely calls the poll-method on the Wire, the Wire object calls the poll-method on the Producer and returns the data to the Consumer.

What are the pros and cons of this solution then? Well, obviously we have now decoupled the consumers and the producers completely and created a very flexible mechanism where we can easily change the connections even in runtime if needed. Also the consumers can poll for values if it likes. The Wire Admin also have a mechanism to create filters to get data when you want it (each second, when the value passed a certain level, ...), which is a very useful way to handle data that changes very rapidly (which is common in vehicle systems) without suffering too much performance.

## ***Conclusion***

Pervasive systems that allows code from many different suppliers have a recurring problem: How should data producers and data consumers be connected when we at development time have problems deciding and/or knowing all about both parts? If we want to avoid taking these decisions early and we also want a good amount of flexibility, then some mechanism in the middle can solve this problem in an elegant way. The OSGi Wire Admin service is one example of such a mechanism. Still (which is important) it is very simple to write both the producer code and consumer code and nearly all the complexity has been moved inside the WireAdmin.

My conclusion is that pervasive systems more often should try to use or create a mechanism like this in able to create a truly flexible system with no (if such a mechanism can be used directly) or relatively little (if some mechanism needs to be created) effort. The complexity within the consumers and producers remains low and the decoupling is extensive.

## ***WhoAmI***

My name is Niclas Nilsson and I work in a small consultancy firm in Sweden called Activa. I have been developing code for embedded systems, always on computers not normally seen as computers, since I got my first programming job 11 years ago.

I have worked several years with the OSGi platform (and its predecessor) within Ericsson and later together with the OSGi Director of Technology for the OSGi organisation. I was also the Ericsson representative in the OSGi Vehicle Expert Group.

Last year, I have been developing software for vehicle computers for Volvo Mobility Systems public transport system.

I participated in this workshop in OOPSLA 2002 and look forward to this years workshop.