Dynamic Runtime Modularity

University of Colorado, Boulder
Senior Projects Fall 2015 - Spring 2016

Sponsor: Jared Stallings, Raytheon

Team: Michael Brughelli, Jonathan Huang,Neil Nistler,
Stephen Rowell, Samuel Tobey

Contents

I. Introduction
II. System Assumptions and Requirements
III. The Hard Problems
IV. Approaches
    Central Server & Client Dependency Resolver
    Client Dependency Resolver & Service Tracking
    Dependency Tree
V. Code & OSGi
    Code Overview
    Managing Dependencies and Using the Code
    Build Tools
VI. Future Solutions
    Dependency Graph & Client Dependency Resolver
VII. Conclusions

I. Introduction

The posed project topic centers around the idea of dynamically instantiating modular software using OSGi and Knopflerfish. The goal of this project is to explore solutions that manage modular software and their dependencies. Perhaps the biggest question that arises with this project is: how is a capability defined, and how can it be determined what services will fulfill that capability? Even after the dependencies of a capability are defined, how can they be provided when they may not be immediately available? Trying to solve this problem tends to lead to more complicated questions.

Throughout this research project we explored many possibilities towards making OSGi dependency resolution more dynamic. We simulated our production system with modular software by creating simple services and finding a way to manage those services dynamically using OSGi.

II. System Assumptions and Requirements

- We assumed there is one service, per bundle, per JAR file.
- We assumed that every service we utilized was within the OSGi specification and Knopflerfish runtime environment.
- We assumed our production runtime environment was Knopflerfish.
- Utilized Apache Maven for compiling and building services and Apache Felix Maven Plugin for creating bundle manifest files.

III. The Hard Problems

The first challenge of this project is adequately defining the problem. What is truly dynamic in the context of this project? How adaptable to changes does the project need to be? It is difficult to apply the idea of dynamic runtime modularity without having a set of services, files, or an existing system to work with. With that in mind, we applied solutions in the context of a problem space we defined. Our focus was to demonstrate that dependencies can be resolved at runtime.

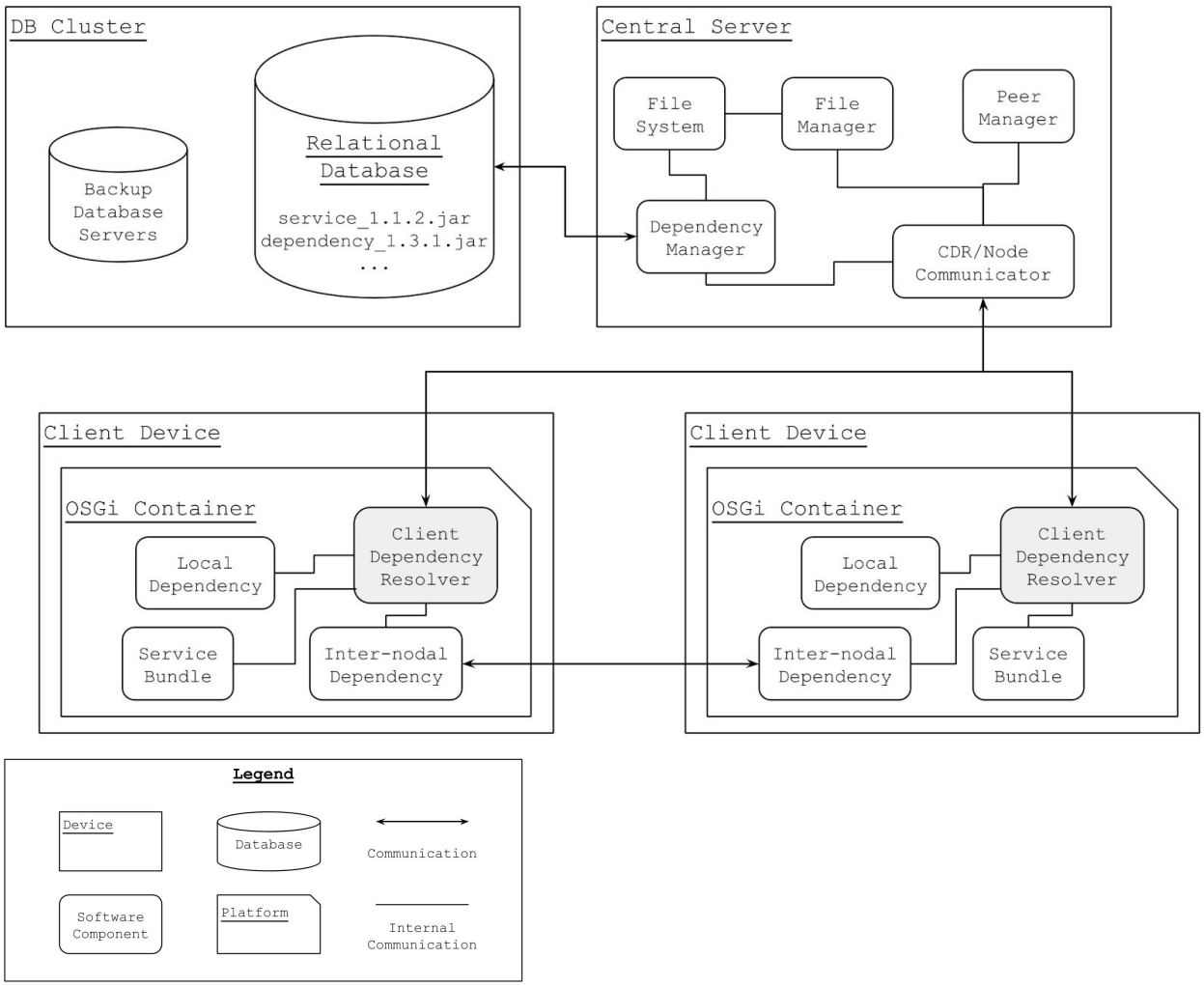
Another challenge of the project is managing a hierarchy of dependencies. We define dependencies in a few different ways: JAR files, services within bundles, or a services other devices. For the purposes of our exploration of this project, we focused on dependencies on the package level, and dependencies as JAR files.

Dynamic dependency resolution is a simple idea, but difficult to implement. We assume that the notion of dynamic dependency resolution is represented by the ability to add, remove, or replace service dependencies at runtime.

IV. Approaches

Central Server & Client Dependency Resolver

One of the first major complete ideas that we had was to divide the project into two different parts - a central server and a local dependency resolver. The central server would act as a repository of JAR files that were services that could potentially be dependencies for services running on other devices in OSGi. Within each client device, there is a bundle or service called "Client Dependency Resolver" that would manage dependency resolution locally as well as remotely with the server. Here is a figure demonstrating the general architecture of this solution:



The interaction of the central server and client dependency resolver is especially important since the central server supplies the resolver with many of the resources it needs for normal operation. The relational database is used for storing dependency information of each bundle and the file system of the central server acts as a repository for all bundles.

The purpose of the client dependency resolver is to communicate with the central server and manage dependencies within the node. What this entails is listening to and tracking services registered with

OSGi and tracking states of services and dependencies of those services. Additionally, another purpose of the CDR is to communicate with other nodes where an external dependency exists. That idea was not fully explored, we also considered a communication layer of those CDR nodes.

Client Dependency Resolver & Service Tracking

This idea was more or less an extension of the first. We felt that our first approach was too large in scope and there was too much overhead in implementing that system while addressing the problem. Instead, we decided to focus on dependency resolution itself at the local level in order to show that we can dynamically resolve dependencies. Essentially, we stripped the central server from the first idea in order to more directly address the problem. However, with further evaluation we have determined that utilizing a relational database for storing dependency information was poor design since the CDR would need to query the central server continuously for dependency information (this thought is based on our future solution). In our future solution, the central server would simply be a remote file system or cloud storage for bundles.

Assumptions:

- Instead of using central server for bundle file storage, we utilized local file storage
- Relational database stored dependency information

Dependency Tree

This idea centered on the idea of the dependencies between service nodes being represented as edges in a tree. Once that tree has been constructed, these edges may be traversed in order to define and resolve those dependencies. Unfortunately, this idea came to us quite late and we did not have enough time to explore this idea. However, a further elaboration is provided in future solution.

V. Code & OSGi:

Code Overview

1. Bundle-starter: A simple service that installs a JAR file and runs it.
2. Service IF: A bundle of interfaces to be used by some of the services within our project.
3. Date-service: A test service for the CDR, dependent on clock-service
4. Clock-service: A test service for the CDR
5. Color services: simple services that display red, green, and blue
6. Color Mixer: A project of color services and the mixer service which all inherit from Color IF in the Service IF bundle. The activator of the mixer registers colors and uses them to draw a new color. The intent of this service is to test the color tracker and how OSGi can be used to keep track of individual or multiple services from the same interface..
7. Client Dependency Resolver: The client dependency resolver contains code for fetching other services through an interface class type. This is a useful way for other services to fetch a reference quickly.
8. Color Tracker: the color tracker tracks services registered as the Color IF type. It will only track services within *one* bundle (e.g. red, green, or blue). However, when the color mixer service is registered, all colors are tracked, since they were registered in the same bundle context and activator class..
9. BundleTracker: the bundle tracker, similar to the color tracker, is intended to track the bundles associated with the color services.
10. Colors: This is a more finalized and dynamic version of Color Mixer. It functions in the scope of a single bundle with several implementations of an interface that presents a colored screen. The activator class prompts the user to select a secondary color and uses logic to register and initialize the two primary colors needed to create the secondary. The logic utilizes unique hash properties that each primary color service is given to filter through the available color services within the bundle. It also initializes the color mixer service that extracts the RGB values from the 2 primary color's services and averages them. Finally, the color mixer service displays the secondary color obtained from "mixing" the two primary services.

Managing Dependencies and Using the Code

1. Central Server: we started implementing this idea but realized it was out of the scope of the project. Ideally, the central server would be where all the binaries, JAR files, and dependency information is stored. The Client Dependency Resolver would remotely connect to the Central Server to fetch binary dependencies that are not locally installed.
2. Client Dependency Resolver (CDR): A service meant to track the status and dependencies of other services. Whenever a service needed a specific service, it would query the CDR and the CDR would automatically install and start the necessary services.
3. Color Tracker: the intent of the color tracker was to be able to track a service in order to determine if it needed to be installed, stopped, or uninstalled. Essentially, the idea was to create a tracker that can track a general service (e.g. Color IF implementations) so that a service could be replaced at runtime for a different implementation. For example, if red was being tracked and we wanted to

start green, we could stop red and register green through the tracker. Another use we had in mind is that we may want to track multiple implementations of the same service in case another service required another implementation.

We encountered some problems while trying to use this tracker when tracking services across multiple bundles (JAR files). The color tracker will track a single color (for example, blue) however, if blue is being tracked and another color is started, an issue arises which prevents that color from being started because the tracker is trying to cast the new color into the one that is already being tracked (for example, casting red as blue). This is peculiar because red and blue are registered with the Color IF rather than the raw types Blue or Red. Even more peculiar is, if color mixer is run, color tracker is able to correctly track all color services because they are all registered in the same bundle. We were hoping to be able to track services defined within different bundles because that was how we originally imagined switching out one bundle for another.

4. BundleTracker: the bundle tracker was intended to leverage this problem of tracking across bundles. Essentially, if we could track a bundle in which a certain service is registered, we could track the modification of that bundle and determine if we needed to install, stop, or uninstall another service without being restricted to a singular bundle. This is assuming that each bundle only contains one service. To use the previous example given in the service tracker section, if the red bundle was being tracked (as opposed to red service) and we wanted green, we could modify the red bundle such that the red service stops and invoke the green bundle to invoke the green service. However, operation across bundles was outside of the scope of our original assumptions. This could be a promising avenue for future work.

Build Tools

We originally were using Ant in Eclipse to build projects' JAR files, and later moved to Maven. Understanding the build.xml and manifest.mf in Ant is very important to correctly design bundles and their dependencies. In Maven, the dependencies are somewhat easier to manage, though it is still worth mentioning that the pom.xml is important for configuring how the manifest file is generated with Apache Felix.

A note on building JAR files: any time that we encountered problems with building, the source of the problem was often related to the "Import-Package" and "Export-Package" statements of the manifest that is needed for OSGi; other problems were related to dependencies in the build scripts.

Generally, building with Maven is really simple but if you do not understand how to use Maven there is a good quickstart guide on the Maven website or refer to our Maven instructional document.

VI. Future Solutions

Dependency Graph & Client Dependency Resolver:

System Assumptions:

Starting services is handled with the Client Dependency Resolver. There is one service per bundle.

Solution Description:

The crux of our solution heavily depended on the CDR. This solution would continue to further explore the CDR by adding several additional features towards dependency management. In this system, the CDR is given control for installing and starting services. For example: when the system first starts up, the first service to start is the CDR; then, other services are started through the CDR instead of being started on their own. Starting of services through the CDR can be expressed in a series of steps:

1. CDR reads the Manifest for dependency information in the JAR file of the bundle containing the service that was requested to start.
    a. Manifest dependency declaration format:
        Import-Bundle: {bundle-name};{major.minor.build};{hash}
2. The CDR checks if those services have been installed and are running. Construct a graph representation of dependencies.
    a. If services dependencies are not installed, the CDR will take the necessary steps to install those bundles.
        i. In our theoretical system, the CDR would fetch these JAR files from the central server.
    b. Service installation and dependency graph construction
        i. The CDR will continue to check if the requested bundles require dependencies (before install). This is a recursive operation, (or tree traversal operation) and will terminate when a bundle has no more dependencies.
        ii. During dependency resolution, a graph representation is constructed and stored in the local memory of the CDR where vertices represent services while edges represent a dependency relationship. This makes it much easier to find out what services are no longer needed when bundles need to be uninstalled to free up space. It is important to mention that circular dependencies may exist, so being aware of that is especially important. Unfortunately, we don't have good ideas for solving the circular dependency problem.
        iii. Differentiating between standard services and mission-specific services is especially important and that is where the hash portion of the CDR comes in handy. In our system, the hash is a SHA-1 result from running the compiled (binary) code through the digest. When a service is started, the hash is included in the service properties so service trackers, listeners, and filters can find a specific service by its hash. The hash is also treated as a "binding-id" that can differentiate between standard services mission-specific services; so, mission-level services that depend on other mission-level services can specify the hash to ensure that it uses a

mission-level dependency rather than a "standard" dependency. When a hash isn't supplied, we make the assumption that the request is for a standard service.

c. If those services are installed, the CDR checks if the services are active/running
    i. Assumption**:** all installed services are running.
    ii. Once we have started the dependencies, we will start the original service that was requested.
    iii. We propose using service trackers and/or listeners for determining when a service has been started. Using the trackers, we can check to see that a service has all of its dependencies started. For example, we might have a service "starting chain:" A→B→C. Assume service C is requested to start and the arrows represent a unidirectional dependency relationship (i.e. C depends on B, B depends on A). The "starting chain" would first start service A and attach a listener to it that notifies B when it was installed. Once B receive a notification, we start service B and attach a listener to B that would notify A. The process continues until there are no more dependencies (we reach the end of the chain). **N.B.:** Be aware of potential race-conditions especially when starting services so keeping the bundle activators Thread-safe and the resolver Thread-safe are very important to the stability of the overall system.

3. Once all the dependencies have been installed and are active, the original requested service is able to start.

In summary, the CDR is a method of abstracting dependency management and making it easier for the end user to focus on starting services rather than dependency management. As long as the user can adhere to several standards of supplying the necessary dependency information to the CDR, the CDR can dynamically start and resolve dependencies on the fly.

VII. Conclusions

Throughout the course of this project, we have arrived at a number of conclusions. Firstly, it is difficult to properly isolate the problem. Although the concept underlying the problem may be easily understood, encapsulating the theory of that concept within an actual system is very challenging. Our exploration of solutions to the problem are constrained to the tools such as OSGi and Knopflerfish, especially since the team had no prior knowledge of them. The difficulty of this project is evident from the numerous changes in scope and direction, as well as our shifting implementation focus throughout the course of the semester.

We have come up with a number of solutions, and explored them to various degrees. We feel that OSGi has many capabilities that our team could not explore due to our lack of experience with OSGi. However, we believe that our ideas can help manage modular software in a dynamic environment. We would like to have been able to test our solutions in a more realistic setting. More work needs to be done to define the context of dynamic runtime modularity, and the applications it has to making software installation more manageable.