DIY Zoning: v.2 Notes

Table of contents

1. Introduction

Warning:

This document represents a tentative architecture guideline for the next version of DZ implementation. This is work in progress, at no moment in time it is to be considered stable or complete, unless otherwise noted.

The main goal of the redesign is to reduce a number of dependencies between the components and provide a unified distributed interconnection architecture that will allow individual components to interact with each other with no regard to physical location.

Basic underlying idea of the new architecture would have been JavaSpaces, if not for its sad demise. Instead, the arhitecture will be loosely based on ComponentBus paradigm.

2. Transition Strategy

Arguably, code rewrites are bad. I tend to share this viewpoint, and whereas refactoring is necessary, continuity is paramount.

Hence, the transition strategy is to keep developing new code, not necessarily compatible with old, but in every case provide transition adapters so the old code can use the new.

3. Location independence

Originally, all the components were implemented inside a monolithic kernel. Then, they were split into DAC, CORE and VIEW modules. Then, it became obvious that the components were in fact smaller than the modules, and that some components were better off located in modules other than they were originally located in. Then, it was realized that an old "one vs. many" fallacy was forgotten again, and the architecture was often providing a place for one entity where it should have provided for many (examples: one logical device per one physical device, one logger in the system). Then, it all clicked together, and it's become obvious that the following requirement should be satisfied:

A component must be able to be placed anywhere inside the system, with the framework providing seamless connectivity to other components providing this components with data or requiring this component to supply data for them.

This necessitates a requirement for the distributed interaction framework.

4. Capabilities discovery

Somewhere down the road, it was realized that the system may be distributed across multiple hosts, and the layout of individual components across the hosts may change. This brought the PnP (Plug-n-Play) extension to communication protocols to life. The requirement to this is as follows:

Sets of components deployed at different hosts must be provided with a generic way of advertising their presense and capabilities to other sets of components across the network.

At some point of time, UPnP was considered, but later rejected because of poor security, high implementation overhead and known architectural problems. In particular, denial of service attacks based on input buffer overruns, scalability problems, route flapping and so on. To be fair, it should be noted that the protocol currently used by DZ is vulnerable to route flapping - bug #731199 (race condition triggered by multiple control connection requests) - this may only be fixed by invoking strong cryptography.

Currently, Apple's Rendezvous (an implementation of ZeroConf specification) autoconfiguration protocol support is in development.

5. Multiple representation

Late in the development it was realized that one physical component may be represented as more than one logical component. Example: DS2438 as both temperature and humidity sensor. This will translate into one software component (DS2438 driver) represented by two logical components (`/sensor/temperature/${address}` and `/sensor/humidity/${address}`).

Likewise, one logical component may be represented by more than one exposed interface: for example, a thermostat is actually represented by two interfaces: `Thermostat`, which is read-only, and `ThermostatController`, which is write-only (according to Principle Of Least Authority).

6. Component Interaction

Note:
We're talking about logical interaction, not implementation.

Since most of the system's components are event driven, the Producer/Consumer (alternative: Publish & Subscribe) interaction model seems to be most appropriate. JMS would be a natural choice, but alas, it is too heavy. An important requirement:

Any component in the system must be able to interact with any other[s].

However, read literally, the above requirement will cause the same kind of mess that exists today. Therefore, an amendment:

No components must interact directly. All the interactions must happen by passing messages from an individual component to the components bus, which in turn will deliver the messages to the intended recipient.

In existing implementation, most (if not all) notifications to the listeners are delivered synchronously. If a consumer fails to process the message (throws an exception), the producer is in danger breaking, if the notification delivery mechanism hasn't caught the exception. Since Java doesn't support multiple inheritance, either the notification delivery code has to be duplicated, or a helper class used. Both solutions are error prone and resource consuming. Therefore,

Component bus must provide asynchronous and error-free notification delivery. Failure of a consumer to process the message must not affect neither the producer, nor the component bus. Additionally, failure of a consumer to process the message should (?) be reported to consumer's consumers, if any, as a failure.

7. Infrastructure Management

The components are many, and their interdependencies are complicated. All of them *produce* data and/or *require* data. In order to provide a proper instantiation order, a *lifecycle manager* is required. In order to allow the lifecycle manager to do its job, all the components must explicitly define what they *require* and what they *provide*.

In order to connect components across host boundaries, the infrastructure manager must invoke the capabilities discovery functionality, and make components aware of each other. It must also pass the data across the host boundaries.

Basically, the infrastructure manager *is* the Component Bus.

Note:

Whereas it is critical to have a producer for every "requires" clause, it is not so critical to have a consumer for every "provides" clause. Unused data may simply be discarded, or queued until better times - the latter is actually much better because of fault tolerance requirements, the consumers may not be present at the time when the message is produced.

Note:

Non-obvious corollary: all the messages must be timestamped, for the nature of the system is time-critical.

8. Fault Tolerance

Hardware and software components may fail. Network connections may fail. Therefore, *a failure* is introduced as a first class object. Therefore, the requirement is:

- *All components in the system are considered transient.*
- *All the listeners of a component will get a notification in four distinct cases: a) when a component arrives; b) when a component departs; c) when a component produces an event; d) when a component produces a failure.*
- *Component bus is the entity that performs dynamic linking of transient components into a working system.*

9. To be continued...