OSGI-BASED GATEWAY REPLICATION

Jonas Thomsen

University of Aarhus
Department of Computer Science
Aabogade 34, DK-8200 Aarhus N
Denmark
jonas.thomsen@daimi.au.dk

ABSTRACT

Modern home automation systems are using OSGi-based residential gateways as the base component of the system. Having one central component being responsible for controlling the entire residence also adds a single point of failure for the residence. In this paper we propose means of eliminating this single point of failure by adding distributed redundancy to the system and being able to replicate the entire system to sub-gateways which will be responsible for operating the residence in islands, in case of a failure in the main residential gateway.

KEYWORDS

Fault tolerance, reliability, replication, residential gateways, OSGi.

1. INTRODUCTION

In a modern communication system, several heterogeneous communication protocols and physical media are utilized. In order to have the different communication subsystems interoperate, gateways are needed.

Classically, a gateway is a device which is able to route data from one communication subsystem to another. Most consumers know gateways as their router for their Internet connection at home, in order to have more than one computer attached to the Internet. Routers are typically just able to route Internet traffic to and from the local home network.

Recently, another kind of gateway has been getting some attention: The Residential Gateway. A residential gateway is, besides being ‘just a gateway’, also a platform for running applications for e.g. home automation. From a hardware point of view, a residential gateway is a box with several physical communication interfaces and an embedded computer for running the applications.

The idea of combining a computer with a gateway is, however, not limited to residences. Automotive gateways (for cars), mobile gateways (e.g. modern cell phones), and building gateways (for controlling e.g. office buildings) also exist.

Throughout the rest of the paper, we will refer to the term ‘gateway’ as a device with communication abilities and an embedded computer.

The structure of this paper is as follows: Section 2 introduces the software architecture for gateways, section 3 introduces the problem we try to solve, i.e. how to add redundancy to a residential system, section 4 gives a general introduction to the topic of replication, in section 5 we discuss our proposed solution for doing gateway replication, section 6 brings a discussion of some of the difficulties we discovered while doing gateway replication, and finally section 7 concludes our work and brings an outlook of further work to be done.

2. GATEWAY SOFTWARE ARCHITECTURE

While it would be possible to implement the gateway software directly on top of the hardware, an operating system (e.g. Linux or QNX [8]) and a gateway framework is typically used in order to have a uniform way of
implementing applications and for being able to manage the gateway. A dominant such framework is the OSGi\(^1\) Service Platform [5,6].

The OSGi Service Platform is a Java-based framework, originally designed specifically for residential gateways, but now (release 3) supports a much broader range of service platforms. The OSGi Service Platform makes it easy to develop applications (known as bundles) for gateways, by giving a strong base of services, which applications can make use of. Furthermore, OSGi is responsible for life-cycle management, security, communication, and a number of other relevant services. By being Java-based, the OSGi framework runs on most hardware, making it easy to deploy gateways in a heterogeneous hardware environment.

While the OSGi Service Platform is just a specification, commercial and open source implementations of the framework exist. ProSyst mBedded Server [7] and IBM Service Management Framework [3] are two of the most used commercial implementations and Knopflerfish [4] is a good open source implementation. Figure 1 depicts the software architecture of an OSGi-based residential gateway.

![Software Architecture of an OSGi-based Residential Gateway](image)

3. REDUNDANCY

The gateway-based architecture has gradually received more and more focus in the home automation area. A residential gateway with a service platform like the OSGi framework makes application development easy and it is possible to manage a great number of gateways from a service provider. The architecture does, however, have one great limitation: It has a single point of failure. A failure in the gateway might bring the entire home in a non-operational state, which is undesirable.

A well-known solution to avoid a single point of failure situation is to add redundancy to the system. Redundancy could be applied by adding an extra gateway and keep the two gateways synchronized. This solution does remove the single point of failure, but once one gateway fails, we are back in the single point of failure scenario.

\(^{1}\)Originally, OSGi was an acronym for Open Services Gateway initiative, but recently the OSGi special interest group has decided to decouple their bounding to gateways, which is why OSGi now is a name and not an acronym.
Another approach to add redundancy is making the redundancy distributed. The residential gateway is typically able to communicate through several physical interfaces, e.g. power lines, Ethernet, ZigBee [10], and Wi-Fi. Adding a redundant sub-gateway to each physical network makes it possible for the home to remain functional in islands, in case of a failure in the main gateway. This could be accomplished by having a special dedicated sub-gateway device for each network or by having an existing device within each network with capabilities to become a sub-gateway.

Regardless of the chosen approach to add redundancy to the residential system, replication is needed.

4. REPLICATION

Replication is a well known means to keep computers synchronized, i.e. that two or more computers share the same state of the system they are running. Replication can be used for providing high availability and fault tolerance in distributed systems and for increasing performance in a system by adding more servers. Two well-known approaches for providing fault tolerance are passive replication [1], pg. 568–560, and active replication [1], pg. 570-572.

In passive replication, one server, the primary server is responsible for serving frontends, i.e. clients to the replicated system, but not necessarily user’s clients, and for replicating its state to backup servers. In case of a failure in a backup server, that server is discarded by the primary server. If the primary server fails, one of the backup servers will be promoted to be the primary server. This can be done by an election [1], pg. 431–436.

With active replication, every frontend communicates with all the servers through multicasts. If a server fails, the frontends will know and discard that server. Active replication can tolerate byzantine failures by having frontends deliver the response that most servers deliver. This approach is among other places used in the space shuttles.

Within high availability replication, the gossip [1], pg. 572–582, architecture is a very common approach. Frontends communicate with just one of the servers (not the same for all frontends). Replication is done by the servers by issuing gossip messages to the rest of the servers, thereby keeping them updated. Updates in the gossip architecture are, however, more complex than within passive replication, as ordering of updates is a problem (there are no synchronization of frontends). This problem can though be avoided by using e.g. vector timestamps, which allows the servers to order the requests to a certain degree (causal ordering, not strict ordering).

Our needs for replication are best compared to the passive replication approach, where our residential gateway is the primary server, and sub-gateways automatically promote themselves into primary servers in their respective areas, when the main gateway fails. Implementing real passive replication is though hard, because of the need for view-synchronous group communication [1], pg. 568, which ensures that the passive replication is correct, i.e. primary server get replaced by a unique server and that all backup server agree on the state up to the crash of the primary server. We will, however, not implement view-synchronous group communication, but rely on simpler mechanisms. We return to this issue in section 5.2.

5. GATEWAY REPLICATION

Our goal is to be able to replicate OSGi-based residential gateways in order to add redundancy to a residential system. We would like to achieve this without having to change anything in the OSGi specification, and we want to avoid changing any existing bundles in the framework. In the following sections, we discuss what, how, and when to replicate.

5.1 What to Replicate

In order to keep the residence functional, all services running in the residence must be replicated seamlessly. This requires that executable code, data, and state must be replicated. It is, however, not as simple as it might sound. In the sub-gateway solution, a sub-gateway might not have all the hardware that the main gateway
has; communication interfaces might be missing. If we just replicate all services, some might not be able to run on the sub-gateway because of the missing hardware. On the other hand, the main gateway might be so damaged, that it has to be completely replaced, in which case we would like the sub-gateways to be able to bring a new main gateway into the same state as the previous one. In this case, we need to replicate all services from the main gateway, even though some of them might fail to run on the sub-gateway.

In order to avoid failures in services that depend on other services that cannot run on the sub-gateway, we need a way to find the dependencies of the services, find out which services cannot run on the target gateway, and compensate for the missing services. At the same time, we still need the missing services, their data, and their state, in order to replicate them onto a gateway that can run them.

Compensating for missing services can be done in two ways: not having the services at all or using stubs (i.e. empty code that implements the needed interfaces) as a substitute for the missing services. Both approaches do add some higher requirements to the services that will be using the missing services; they must be able to handle missing services or missing responses from calling the services they expect.

Given the interface of the missing service, it is possible to auto-generate its stubs in Java by using reflection. A better approach would, however, be that services themselves implements their stubs, so they might give some static responses in case of the missing hardware. If e.g. a power line sub-gateway needs some temperature readings which are communicated through ZigBee, the temperature delivery service might store the last known or the average temperature. In case of a missing ZigBee interface it could just return the stored temperature instead of the real temperature. This approach does, however, require services to be designed for the situation, which we would like to avoid.

Replicating everything from the main gateway might require more storage capacity than available on the sub-gateway. We do, however, not see any solution to this problem and assume that the sub-gateways will have sufficient storage capacity.

5.2 How to Replicate

In this section we look at how we can replicate gateways in OSGi. In section 4 we concluded that viewsynchronous group communication is needed in order to implement real passive replication. In our current work, we have ignored this problem and are just transferring our replicated data between gateways by direct communication. The risk of this approach is, that the main gateway will fail during a replication operation and that sub-gateways do not agree on the state of the system. There are, however, not that many changes that are really important and have a significant influence on the state of the home; if the latest temperature measurement did not get through to all sub-gateways, the islanded home will still work the way it is supposed to.

An OSGi framework is basically a collection of bundles. A bundle is the term for an application or a library, and it is just a jar-file containing the compiled Java-classes for the application or library plus an XML-file describing the bundle. The description contains information about what services the bundle uses (imports) and publishes (exports). This information is stored in the framework when a bundle is installed. It is though possible to query the framework about installed bundles and their dependencies. Furthermore, we can get a complete list of all installed bundles in the framework. Given the list of bundles and the dependency query mechanism, we can build a complete dependency graph of the framework with all its installed bundles.

5.2.1 Replicating Executable Code

From the list of installed bundles, we are able to get the bundle itself, i.e. the jar-file that contains the code. This file can be transferred to the target gateway and installed.

5.2.2 Replicating Data

In the following we define storing data in the gateway as putting data in a persistent storage or as giving the data to another service within the gateway, for it to be able to retrieve the given data at a later time. Data that is kept in the memory of the service is not stored; this is part of the state of the service. Data given as parameters to other services is also not considered for storing, unless that service promises so and thereby stores the data following the definition above.

Furthermore, we assume that a bundle only uses OSGi handles to store data in the gateway, i.e. bundles are not allowed to access the file system without going through the OSGi framework. Data stored externally
from the gateway is not considered as being part of replicating the gateway and is ignored. If a bundle stores data externally, it must handle the case where the data is unavailable due to communication failures itself.

In OSGi, data can be stored in the gateway in several ways. The framework offers a persistent storage for bundles, through the `getDataFile(String)`-method in the `BundleContext` object. This method creates or opens a file for the bundle in a protected storage for this specific bundle. The only specified method for accessing the bundle’s storage is the `getDataFile(String)`, which means there is no standardized way of getting access to the storage of other bundles. In order to replicate the storage, one has to go outside the OSGi framework, if possible. The OSGi specification does not specify how the persistent store should be implemented, so an implementation with very high security concerns might keep the persistent storage encrypted with unique keys for each bundle. In our reference implementation of the OSGi framework, Knopflerfish, the persistent storage is just a folder on the hard disk, which makes it quite easy to replicate by opening the folder directly outside the framework.

Data can also be stored persistently as preferences. Preferences are available through the `Preferences Service`. Preferences are hierarchical tree structures with named nodes, where each node in the tree can store properties, i.e. key/value-pairs. Preferences are designed to store different users’ data within a bundle. Each user has its own root node in the tree and users’ root nodes are children to the bundle’s `system root`, i.e. the `real` root of the tree. A bundle can only have one preferences tree. As preferences are managed by a service in the framework, replication of this service automatically replicates all preferences. There is, however, one catch here: Preferences and bundles are linked through the `BundleID`, i.e. a unique number that each bundle has `within` a framework. If a bundle is replicated to another gateway, it might not get the same BundleID as it has on the main gateway, which would link the stored preferences to another bundle.

The last way of storing data in the OSGi framework is through `Configurations`. Configurations are a means of decoupling a service’s runtime configuration from its implementation. This can be used when there is a need for multiple instances of the same service with different configurations. A configuration is a `Dictionary` consisting of key/value-pairs. A configuration is given to a bundle when it registers a `Managed Service` or a `Managed Service Factory`. Configurations are administered by the `Configuration Admin Service`, which would be replicated by it self.

### 5.2.3 Replicating State

The state of a running application is the position at which the CPU is processing the code, the content of the CPU registers and the memory. Moving running code is a well known issue within `Mobile Agents`. In [2], Fuggetta et al. describes two mobility models: weak and strong mobility. The state of an agent with weak mobility is basically just its program-defined data structures, where the state of an agent with strong mobility is the underlying thread’s or process’ state. With weak mobility, the agent can only be migrated at certain points in its code, where an agent with strong mobility can be migrated at any point of its execution.

The concepts of weak and strong mobility can easily be used in our gateway replication: A bundle which stores its state in a persistent storage as discussed in section 5.2.2 could be considered being a bundle with weak mobility, where a bundle that does not store its state persistently could be considered a bundle with strong mobility. Replication of a weak mobile bundle is easy — it is just a matter of replicating the code and the data as previously discussed. It is, however, hard to replicate a strong mobile bundle.

We do, however, not see the need for handling strong mobile bundles, as most usage scenarios for residential gateways are event driven, i.e. bundles perform an action when an event occurs. This would be the case for reacting on changes in sensor readings, which is the primary part of doing home control and automation. So, we support replication of state for weak mobile bundles, but not for strong mobile bundles.

### 5.3 When to Replicate

In order to maintain the highest level of reliability, replication should be performed whenever something has changed. Ideally, this information would come from the bundles themselves, but that would require them to be rewritten, which we do not want to.

We have previously seen that we only have to concern ourselves with changes in code and data — state is just data. We discuss the two cases individually below.
5.3.1 Code

The OSGi framework monitors all installed bundles. If a bundle is updated, the update goes through the framework. It is possible for a bundle (e.g. our replication manager bundle) to register itself as a Bundle-Listener, where it will get all events regarding life-cycle changes for bundles. Whenever a new bundle is installed or an existing bundle is updated, we are able to replicate it immediately because of the BundleEvents we get.

5.3.2 Data

As discussed in section 5.2.2, there are three ways of storing data in the OSGi framework: The file system, Preferences, and Configurations. We discuss these three ways here:

File System: As seen in section 5.2.2, there are no handles in the OSGi framework for getting access to other bundles’ file systems. The only way is to get access through native Java calls outside the framework. Likewise, we cannot get any notifications from the framework when other bundle’s private data is changed. Native Java does, however, not have any means of monitoring a folder and all its sub-folders for changes in files, so the way to monitor changes in the file system is by scanning it periodically.

Preferences: The Preferences Service is not specified to have any means of getting events when a change is made. As a result, we do not have any idea of when to replicate this service, so the best we can do is replicate it periodically.

Configurations: The Configuration Admin Service allows Configuration Plugin Services to see and even change configurations before they are passed to the Configuration Target, i.e. the Managed Service or the Managed Service Factory. By registering a Configuration Plugin that listens for all configurations, it is possible to detect all changes in configurations and replicate them immediately.

6. DIFFICULTIES

Throughout the last chapters, we have discussed how OSGi-based gateways can be replicated. We did, however, held back some of the problems we discovered, which we present here.

Permissions: If an OSGi framework supports Permissions, it will not be possible to perform gateway replication unless the replication manager bundle has Admin Permissions, which are required in order to query other bundles’ internal properties, e.g. the location of the bundle jar-file. In Knopflerfish, we could access all methods requiring admin permission without actually requesting the system to give us that permission, so we assume that Knopflerfish either not support permissions or has given all bundles admin permissions by default.

Ensuring consistent BundleIDs: Within the OSGi framework, all bundles are identified by their unique ID, the BundleID. This ID is, however, only unique within the framework. If we just replicate bundles in a random order, BundleIDs might not be consistent on all gateways. This will cause services like the Preferences Service to bind preferences from one bundle to another when replicated, as the binding is based on the Bundle ID.

BundleIDs cannot be controlled by bundles — this is managed by the framework itself. We can, however, avoid the problem, because our scenario is based on one main gateway and several sub-gateways. If we just replicate bundles in the order of their IDs on the main gateway, they will get the right IDs on the sub-gateways. This does, however, not allow bundles to exist on sub-gateways alone, as these bundles will take IDs which might be used by new bundles on the main gateway.

No interoperability between different framework vendors: The OSGi Service Platform specification does not contain any information about implementations of the required services. As a result of this, we cannot replicate between gateways running the OSGi framework from different vendors, as we cannot replicate persistent storages. Bundles can of course be replicated, if they do not store any data and if they are implemented without using any vendor-specific interfaces.
Standard bundles are not OSGi-compliant internally: The implementation of the required standard services in the OSGi frameworks on the market cannot be guaranteed to be 100% OSGi compliant. An example of this is the Preferences Service implementation in Knopflerfish: Storing the preferences requires access to the file system, which is done without using the built-in `getDataFile(String)` on the framework. Because of this, we have no chance of guessing where the service stores its data and we are not able to replicate it.

Boot-Strapping the Replication Manager: Currently, we have to install our replication manager manually on the gateways. It would, however, be user friendlier if we could install our replication manager automatically from either the main gateway or from a service center. OSGi does have a means of installing a Management Agent from a remote location. After installing the Management Agent, the gateway can be completely managed from a remote location and bundles can be installed automatically. This is handled by the Initial Provisioning part of the specification. We do, however, still have a boot-strapping issue, as the Initial Provisioning requires an URL to fetch the Management Agent from. We have not implemented this part.

7. CONCLUSION AND FUTURE WORK

Throughout this paper, we have presented a way of doing replication of OSGi-based residential gateways. We have succeeded do so to a great extent, but we have not been able to handle all possible code, data, and state. We have tested our replication mechanism on the Knopflerfish reference implementation with good results. There are, however, still cases we are unable to handle — some because the Knopflerfish implementation of services access storages outside the framework, e.g. the Preferences, and some because there are missing handles in the specification, e.g. getting other bundles’ persistent storages and getting events when other bundles’ preferences are changed.

Given our goal of not requiring the existing bundles to be rewritten, we have managed to come quite far, but in order to go further, some changes are needed. The bundle developer is the one with most insight to the specifics of the given bundle, and he should be able to specify when and what should be replicated, through e.g. a call to a replication manager service with a standardized interface. Furthermore, by adding more handles to the framework itself, we could get more detailed knowledge of when to replicate and thereby do a better job.

REFERENCES