4 SIMPLIFY OPENHAB UTILIZATION IN PUBLIC INSTITUTIONSUSING CLOUD TECHNOLOGIES

A. Dobler [a,*], D. Uckelmann [a], G. Lückemeyer [a]

[a] Department of Geomatics, Computer Science and Mathematics, University of Applied Sciences Stuttgart
Schellingstraße 24, D-70174 Stuttgart (Germany)
(alexander.dobler, dieter.uckelmann, gero.lueckemeyer)@hft-stuttgart.de

KEYWORDS: openHAB, Public Buildings, Docker, Swarm, Cloud

ABSTRACT:
Smart Home solutions like openHAB help to automate the control of connected devices and grant an overview of monitored appliances, as well as reduce energy consumption. Public institutions like universities or schools could also benefit from such solutions, especially considering they are often located in old buildings. But while openHAB is a feasible solution for home users, public institutions usually have other requirements towards their software, e.g. regarding security, maintainability and fault tolerance. Beside this, public institutions usually span a bigger area, often with multiple buildings which have to be connected with each other. This work analyses how the infrastructure of a common openHAB stack looks like, by examining which software components are usually used and how they are interrelated. It tries to point out possible shortcomings concerning the needs of a public building. Furthermore it showcases how these shortcomings can be tackled by using cloud technologies locally. To achieve this, an example setup was created using Docker and it's orchestrator Docker Swarm. It is explained how the single applications of an openHAB stack are integrated into the swarm, how to manage multiple possible instances and what to consider to simplify it's usage. The outcome of this work shows that such a setup helps to improve the security, by running in a separate network with strictly explicit entry points. It also demonstrates that maintenance can be kept low by allowing for an easier and central configuration as well as simpler backups of user data. This in return helps to react quicker in case of failures or changes to the system and therefore to achieve better availability.

4.1 Introduction

Big buildings or complexes need a way of monitoring, managing and automating their infrastructure to ease the work of facility managers and help them to keep an overview, as well as to control building appliances. An intelligent control of climate, heating and light systems in a building can also help to reduce energy consumption and save money (DIN EN 15232-1:2017-12, 2017). For these use cases many commercial options are available (Cisco Systems, 2018; Ultimo Software Solutions, 2018; SoftGuide GmbH, 2018), but they tend to be expensive, closed and tied to proprietary hardware.

In recent years the interest and demand is on the rise to control own private homes too, pushed by smart speakers like Amazon's Alexa and Google Home (Martin, 2018). But beside these commercial offerings, the smart home sector also offers a small range of open and extensible platforms like openHAB, FHEM or Home Assistant. They act as a hub and allow to interconnect many different platforms with each other. Out of these openHAB offers a matured and solid solution with a core that is used by commercial consumer grade solutions like QIVICON by Deutsche Telekom (Deutsche Telekom AG, 2016).

The usage of such an open system in public buildings, especially in educational facilities would be an interesting choice. Beside the already mentioned advantages for monitoring and energy savings, it would also be a platform which is extendible and open for improvements. These extensions may even be developed in projects by students, providing them with interesting educational tasks, as well as improving the infrastructure of their facility. OpenHAB would also be an economical choice as it's bare usage is free (except for maintenance and operation) and not tied to (expensive) proprietary hardware. To make this possible, openHAB as a software needs first be checked for the demands of public facilities or larger environments in general and adapted if necessary.

*Corresponding author

D. Uckelmann (Ed.): Proceedings of the 1st Conference on Smart Public Buildings, pp. 31–41,
19-20 October 2018, University of Applied Sciences Stuttgart, Germany.
ISBN: 978-3-940670-66-3

This work is derived as a part from an ongoing master thesis with the goal to adapt openHAB to be used in public institutions. This paper focuses on making it easier to deploy and operate openHAB as a solution for multiple buildings. To achieve this openHAB is considered as part of a bigger stack combined with other components. The resulting solution uses *Docker* and *Docker Swarm* to enable a simpler setup and operation.

4.2 Analysis and Concept

4.2.1 Requirements in Public Buildings

The requirements for smart home software and software which is meant to be used in bigger buildings usually differ a lot. First there is the general purpose of smart home and smart public building software. A big focus for smart home users is entertainment and comfort, for example being able to control lights without getting up, while also controlling the music playing in the living room (techuk, 2017; GfK, 2016). The market for smart buildings on the other hand is mainly driven by possible energy cost savings, they also should allow for visualization and monitoring (trend:research, 2015).

The scope in which public and home systems operate differs too. This starts with possibly different devices they need to control (e.g. entertainment systems or not), but mainly concerns with the size of the area to manage. While a smart home solution is in control of a single flat or house, a public institution can easily span a much bigger area. The HFT Stuttgart for example has eight buildings spread across an area of roughly 35,000 square meters. For a smart public building this means, it not only has to cope with much more devices (like sensors or lights), but it also needs to ensure to cover a bigger area with possibly independent sections (like buildings). This may require to represent these sections in another UI layer, not needed in a single home. Also important to consider is that public institutions themselves can vary a lot. Not only by size, but also considering equipment and infrastructure. While a university may sport a well grown server landscape, this may be different for small schools. A possible solution would ideally be able to operate within both.

Public and home systems also differ concerning the installation and operation of theses systems. The biggest challenge towards the adoption of a smart building system is actually the same as for smart home products, the cost (trend:research, 2015; techuk, 2017). In addition, public buildings like schools often do not even receive money for urgently needed refurbishments (van Laak and Schröder, 2017). This means in return, the initial setup cost and necessary resources for a smart building solution have to be kept low to enable a successful adoption. Beside the initial setup, another cost driver of software in general is the ongoing maintenance and operation, like ensuring availability, managing failures of soft- and hardware, as well as running updates and supporting users (David et al., 2002).

Two key aspects while operating any distributed software system, which are helpful in this regard, are replication and fault tolerance. Replication allows to gain a higher reliability by keeping data available even in case of system errors (Tanenbaum and van Steen, 2008, p. 303ff). As data may be changed in multiple locations, it still needs to be consistent through out the system to ensure correctness. Working with many distributed components means, that also partial failures are possible. These shall have a minimal influence on other parts of the system, which again leads to higher availability and less effort during recovery (Tanenbaum and van Steen, 2008, p. 354ff).

Another important aspect is security. While security can be provided by several mechanisms such as encryption, this also means that it is a broad field and difficult to do pr operly. Therefore it is important to define a set of security policies first, which are used to find fitting mechanisms to reach compliance (Coulouris et al., 2002, p. 298)(Tanenbaum and van Steen, 2008, p. 413ff).

4.2.2 A Common openHAB Stack

Although openHAB by itself provides a lot of functionality and its architecture is very modular, most openHAB setups come with at least one or two additional software components. These components usually provide functionality outside of the core focus of openHAB, like persisting data or managing telemetry data transport. Even the developers of openHAB seem aware of this. In their official openHAB distribution for the Raspberry Pi single board computer, *openHABian*, they directly included an installer menu for some of the most used external components like *Node-RED* and *Mosquitto* (openHAB Community, n.d.b). Further on, a set of commonly used components is described and how they also benefit in bigger setups.

Data Persistence

A common example for an additional component is item state persistence. While openHAB is able to store data in a local database, like it does for item configurations and settings, it does not rely on it to keep track of item state changes over time. To achieve this openHAB uses extensions called persistence services that allow it to store state changes in a range of external databases like MySQL or InfluxDB and can be installed like bindings directly through its web interface PaperUI. It is an important addition as it allows us to work with and analyse historical item data.

Rule Engine (Node-RED)

While openHAB provides its own textual rule engine, many users additionally rely on the external application Node-RED. As mentioned before it is even included as an install option in the official openHAB Raspberry Pi distribution (openHAB Community, n.d.b). Instead of using text files, Node-RED allows to define rules using a visual representation, displaying rules as a flow of connected nodes. It is also able to work with multiple instances of openHAB and therefore can be an advantage in a multi building scenario.

MQTT

Another often used external service or in this case rather protocol is MQTT (even in advanced setups (Pham et al., 2015; Fu et al., 2017; Wlotzka, 2016)). MQTT is a protocol tailored towards machine to machine (M2M) use cases (mqtt.org, n.d.). It allows to easily connect additional sensors and actuators not natively supported by openHAB, as well as to connect with other smart home systems. It can also be used to share events between multiple openHAB instances and thanks to the gained flexibility it can be an helpful addition.

Web server / Reverse proxy

A web server is something not strictly needed for an openHAB setup, but it provides a big addition, namely authentication. By default an openHAB instance is quite exposed to the local network. To reduce risks arising from this, the openHAB documentation recommends the usage of a web server as a reverse proxy(openHAB Community, n.d.a) and enable the basic authentication method of the HTTP standard. The web server is essentially receiving requests from a client and forwards them to the fitting application and handles additional tasks such as encryption, data caching and authentication.

Sensors and actuators

The last external part we consider here is not strictly an external component. It describes how devices controlled by openHAB are integrated. Sensor and actuator devices, like switches and lights, can be connected to openHAB with several different methods. One option is to connect devices directly to openHAB, in these cases openHAB

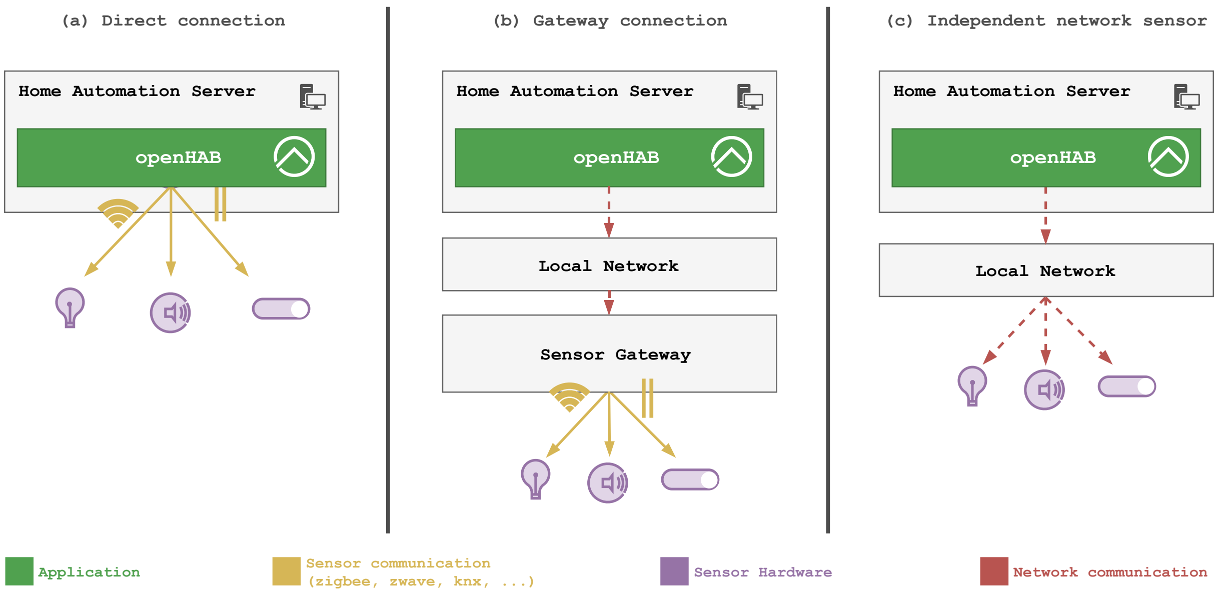

Figure 1. Connection Types of openHAB Sensors

uses available interfaces (either wired or with the help of radio hardware) and manages connected devices on its own (see Figure 1 (a)). An example here is the *Z-Wave* binding of openHAB, it uses a connected Z-Wave serial adapter to directly communicate with Z-Wave devices.

In most cases however, the devices are not directly attached to openHAB, but rely on a third-party hub or gateway to communicate with it (see Figure 1 (b)). These vary in appearance, being either other servers or more often small embedded devices. These gateways are connected to the real physical actuators and sensors, either by wires or by using a wireless connection based on a radio standard. For both wireless and wired connections a range of different protocols are available which the gateways may use to interact with the sensors and actuators, e.g. Z-Wave, ZigBee or KNX. OpenHAB then connects to these gateways, usually accessing them trough an IP connection, e.g. by using an exposed REST API, to get access to all connected devices of the gateway.

The last option are single devices which directly reside in the same local network and can be accessed through an IP connection by openHAB (see Figure 1 (c)). Towards openHAB they behave similar to sensors using a gateway but operate independently.

4.2.3 Current State

The previous section outlined, a complete openHAB setup usually consists of way more parts then openHAB itself and that public buildings can also take advantage of them. It is therefore necessary to consider openHAB together with these (and possibly further) components as a complete stack when looking for shortcomings and possible solutions, as well as to keep in mind that we additionally have to deal with multiple possible instances within separate buildings.

4.2.3.1 Security

To begin with, we will look at the state of security of our components. The previously described basic stack, when run at home, could look similar to what is pictured in figure 2. It shows the single components running as services on a single machine. It also shows how a network setup could look like using the defaults that adhere when setting up the single components. In particular, the many open accessible ports may attract attention. When applications are exposed to the network like this, they may be misused. Especially considering networks of public institutions with a mixture of staff, students and guests having access to the network.

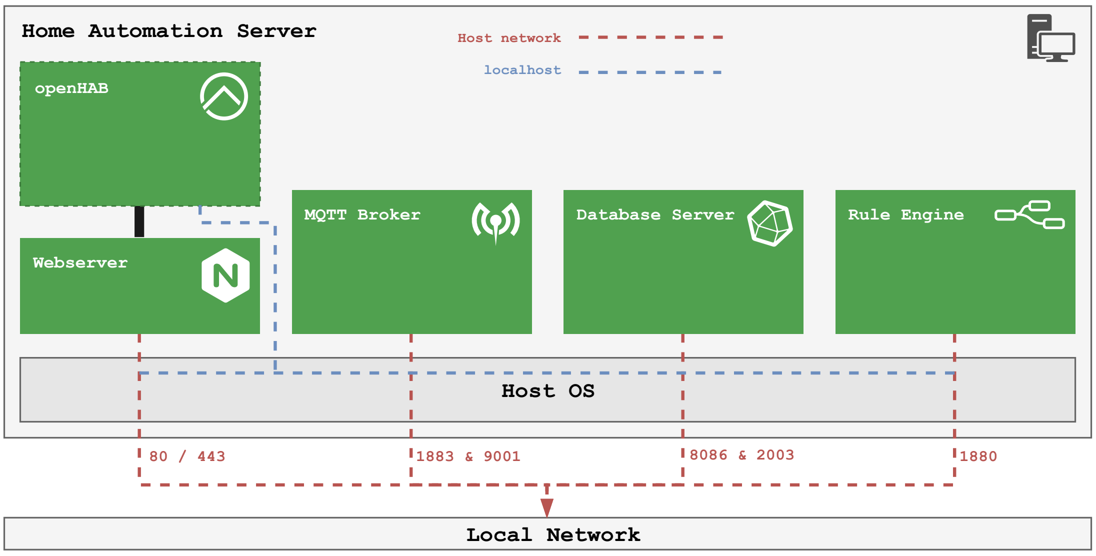

Figure 2. A typical full openHAB setup

The main problem illustrated here is not the security of the applications themselves. Some of the pictured examples (Mosquitto, InfluxDB, Node-RED) are capable of encrypting their communication and have some means for access control (openHAB only with the help of a web server). The issue here is, that all applications are easily accessible through the network, although not always necessary. This is caused by the defaults used during their setup. Parts of these may be feasible in a private network, but they do not fit the needs of public environments. A reason for this is, that defaults usually don't have to follow a security policy which would define possible security mechanisms and rules, something that would be necessary in an public environment.

4.2.3.2 Operate and Maintain

Handling multiple different software parts can not only create security risks, it also increases the complexity during operation. Installations and managing configuration data can easily grow to be a maintenance intensive task in a distributed environment. Our openHAB solution does not only consist of many software components, it also has to be distributed across multiple buildings. In a classical on premise environment, it would be necessary to setup all needed applications on every machine, each with the correct configuration for both the OS and the applications. Every change of a configuration, executed on one of the machines, needs to be mirrored at a central location to keep track of the current settings. When one of the machines has an system error and won't start any more, these steps need to be repeated for all applications which ran on that machine. Most applications we are working with like openHAB and Node-RED are stateful applications, they keep track of state data using files on disk (e.g. in case of openHAB, all changes made from PaperUI). For this user data, backups are necessary to ensure a successful recovery when a system fails or to revert erroneous changes. All needed work grows easily with the amount of machines and applications that are used. Especially when executing setup and configuration steps manually, it can not only get tedious, but is also error prone as these steps would not be easily reproducible.

Installing multiple components on a single machine can lead to influences between the applications and cause unexpected problems. Possible is for example that a faulty application, which leads to erroneous system behaviour, prevents the other components from operating correctly. This can get especially problematic when adding new components to the system. In return this can lead to lower availability of our applications.

Summarized this means, to enable proper installations, configuration changes and backups, it is necessary to keep track of different sets of files, in different directories and in addition for many different buildings. This can increase the demand for additional educated personnel (which is another big challenge for smart buildings (trend:research, 2015)) and therefore the cost needed to run such a system.

A solution to this needs to simplify the handling of configuration files and application data as well as the management of backups. The goal here is to reduce the needed effort for these tasks and in return keep possible additional maintenance cost low.

4.2.4 Creating a Better Solution

A part of the mentioned shortcomings can be solved with solutions like openHABian. This ready to use image comes with most parts needed by a home user and allows to easily install additional components from an additional UI. For a public building a similar image could be build, but this would fast render as ineffective as we have to deal with different possible target environments (e.g. bare metal, VMs) and different needs towards the components. Another solution could be the use of a managed and automated infrastructure with the help of tools like *puppet* or *chef*, but they come with a high complexity and initial effort.

A different approach to tackle these issues, is the usage of containers. A technology with a growing popularity in recent years(Forrester Consulting, 2017). Container engines like *rkt*, *LXC* and Docker allow developers to package and execute applications in a sandboxed environment with all necessary dependencies. They can be combined with orchestrating tools like *kubernetes* and Docker Swarm to gain a solution with an easier and central management of applications and their configuration.

The following pages showcase how containers, together with an orchestration engine, can help to create a better environment for running an openHAB stack in a public building. As an example the described solution is based on Docker with the accompanying orchestrator Docker Swarm.

4.2.4.1 Moving to Docker

As the title of this paper already suggested, we will use cloud technologies to tackle some of the mentioned shortcomings. The term cloud can have several meanings and can be used on different levels (like IaaS, PaaS, SaaS) (Hentschel and Leyh, 2016) and therefore needs some explanation in this context. We do not rely on specific cloud providers like Amazon Web Services (AWS) or Google Cloud Platform, but we will use tools that allow us to execute applications in sandboxed environments and help us manage them independently from the used local infrastructure. In our example we use Docker and its orchestrator Docker Swarm as the main tool to show case the advantages of a cloud like environment. An advantage of Docker is, that it is easy to get started with. It is quick and simple to install on provisioned VMs (when available, e.g. in universities with their own PaaS infrastructure), as well as on bare metal computers (e.g. in a school without proper server infrastructure). A special infrastructure is therefore not necessary and there are no licence fees for its usage.

4.2.4.2 Deploying with Docker

Deploying and running our applications in containers helps us to gain multiple advantages. The sandboxed environments make installations easier, the applications don't have to rely on packages installed on the host OS, but manage them in their own separated environment. Applications are also less likely to influence other components on the system. When they fail it only concerns the container they are running inside. What really simplifies the deployment itself though, is the easy definition of whole sets of applications working together. In case of Docker this is done with *Docker Compose*. A `docker-compose.yml` is a configuration file using the *YAML* format. It allows us to define a set of services with the application containers we need (see Listing 1 for a simplified example). When running a compose file, images containing the defined applications are automatically downloaded and started, manual installation routines are not required any more.

```yaml
services:
  openhab:
    image: "openhab/openhab"
  nodered:
    image: "nodered/node-red-data"
  mqtt:
    image: "eclipse-mosquitto"
```

Listing 1. Simplified example with three services

This approach is extended by the container orchestrator Docker Swarm, considering the use case of multiple buildings we intend to use. Docker Swarm allows to connect machines (nodes) running the Docker daemon with a single command. The connected machines then operate as a cluster. When the stack defined in the compose file is executed, the connected nodes will now download and run the needed containers. As this happens by default, depending on the available resources, it is possible to assign labels to the nodes and depending on them distribute containers. Listing 2 pictures an example where two openHAB services are placed on different nodes depending on the assigned `building` label.

```yaml
services:
  openhab-building1:
    image: "openhab/openhab"
    deploy:
      placement:
        constraints:
          - node.labels.building == b1
  openhab-building2:
    image: "openhab/openhab"
    deploy:
      placement:
        constraints:
          - node.labels.building == b2
```

Listing 2. Placement of two services in a swarm

Using containers and an orchestrator simplifies deployment a lot and allows us to get an easy overview of our applications. In addition it also handles most networking tasks by connecting all containers within their own network and makes them reachable by their service name across all nodes.

4.2.4.3 A More Secure Network

As indicated previously, a security policy is necessary to be able to achieve a defined level of security. Derived from the shortcoming described earlier considering the defaults of our stack and the surroundings of public buildings, we can define a sample set of policies that we will try to comply with:

1. By default access to none of the applications (or their ports) shall be allowed.
2. When access is needed, the applications (or their ports) are explicitly exposed to be usable. The goal is to offer as little open ports as possible.
3. Only users which are able to authenticate themselves, shall be able to use any of the applications.

These rules will help to minimize the exposed area of our system and restrict access to selected users. Especially when dealing with networks with possible uncertain participants these simple rules can help to gain a basic security level by lowering the target area.

The first step towards this was already done by moving our applications into Docker containers in the previous step. Docker takes charge of handling the networking between the containers. In case of a swarm stack, Docker uses a so called overlay network (Church, n.d.). This uses Virtual Extensible LANs (VXLAN) to create virtual networks between the connected Docker hosts and their containers. These virtual networks run separated on top of the underlying host. Therefore by default the containers in the overlay network are not reachable from the host nor the network the host is connected to. The applications inside the containers can still communicate easily with each other, even in between the hosts (see Figure 3).

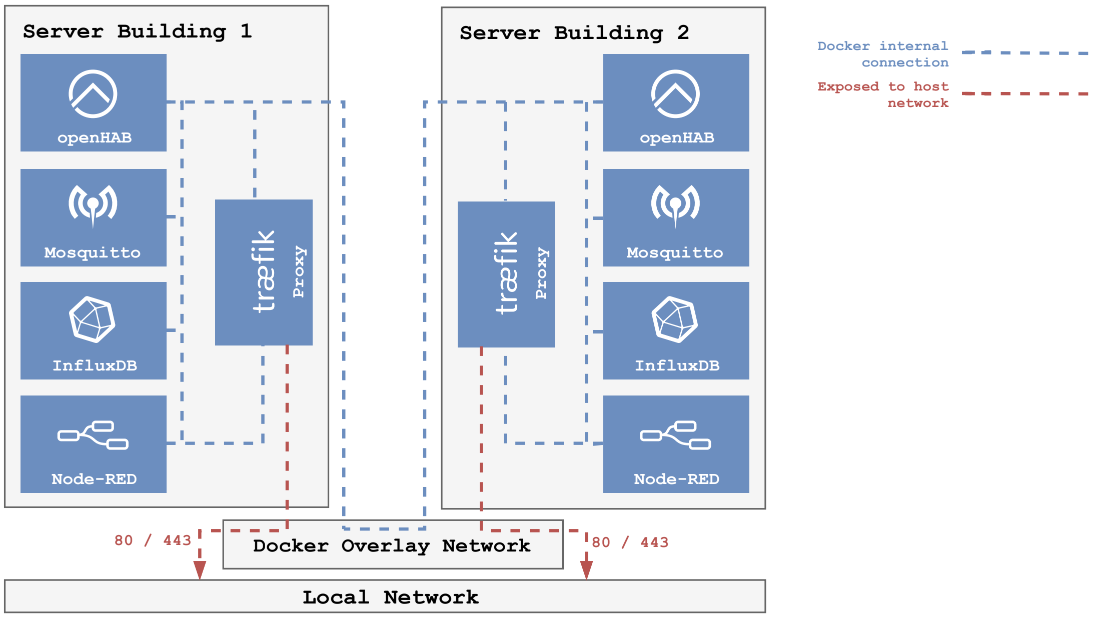

Figure 3. A stack using Docker networking and traefik

To be able to reach an application when it is necessary (e.g. for WebUIs) Docker allows to define accessible ports. For these, Docker creates bridges from the host port to the port of the container as if the application would run directly on the host (e.g. access from the local host network is also possible). Instead of simply exposing all WebUIs to the network, additional security (as well as simplicity) can be achieved by adding a web server or reverse proxy to the stack. It adds the mentioned ability of authenticating users and can reduce the number of our exposed ports to just one (either 80 or 443 with SSL). The reverse proxy then passes all requests to the correct applications and allows the usage of easy to remember subdomains instead of ports. In the pictured example we use *traefik* for this task, a reverse proxy that is tailored for cloud environments and can be configured using the deployment labels directly in our compose file. Other solutions like *nginx* could also be used, but don't offer the same level of integration by default.

With Docker networking and traefik we can cover a basic security policy. By default our applications are not reachable from the local network they are running on, in return only a single port is accessible which also handles user authentication.

4.2.4.4 Manage Configurations

Manually managing the configuration data of openHAB and it's accompanying components on multiple machines, turned out to be insufficient regarding maintainability and availability. While Docker Swarm helped us to define an easier installation and deployment of our applications on a set of different machines, it is also important to run them configured correctly. Cloud native applications like traefik are mostly configurable directly from the compose file using either labels or environment variables. However, this is not necessarily possible in case of classical server applications like openHAB, Node-RED or Mosquitto. For these cases Docker Swarm has the possibility to define so called *Docker Config* entries. These are configuration files typically placed along the compose file and named in the `configs:` section. When the swarm stack is started (or updated) these files will be loaded into the *raft log*. This is an encrypted storage that Docker uses to ensure consensus between all nodes and therefore is available on each. From this raft log the file is then mounted into the containers by the local Docker daemon. Because the raft log is always distributed to connected machines, it is ensured that even new machines (e.g. after replacing a broken one) have the needed configuration files available and are able to re-spawn missing services seamlessly. Another advantage of this solution is, that all configuration files are collected at a single location together with the compose file. This enables us to use additional services to properly manage configuration changes (see Figure 4 *(a)*). A good solution for this is provided by version control systems (VCS) like Git. This allows us to keep track of all configuration changes and additionally keeps them at a different location. A VCS also allows to revert faulty changes and in case of a recovery it is easy to simply restore all settings from the repository.

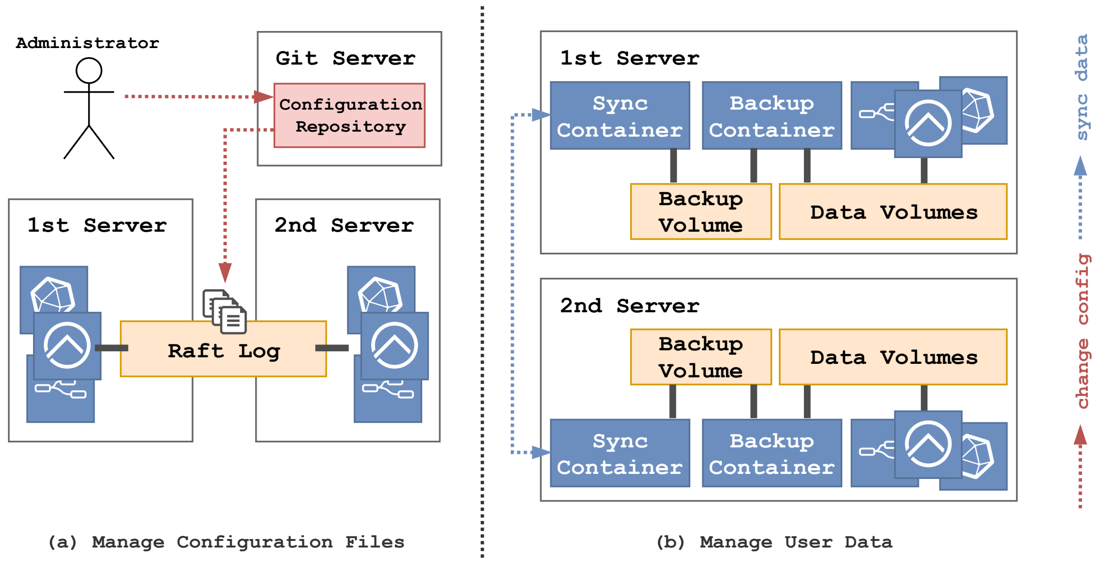

Figure 4. Manage user data and configurations within an orchestrator

4.2.4.5 Manage User Data

While Docker Config is a perfect solution for static configuration data and settings, it does not work for user or application data. This data sports regular changes also applied from within the corresponding application itself and requires write access, which is not possible with Docker Config.

User data persists the ongoing state of an application (e.g. channel links in openHAB) but can also represent temporary files or cached data. While the latter is usually not needed to be able to restore an application, state data is important. Docker makes it easier to separate these two. Persistent data is added to Docker using *Docker Volumes*. They represent a dedicated area on the local disk (e.g. in `/var/lib/docker/volumes`) and are managed by Docker. They enable us to dynamically assign it to application containers with the `-v` flag or directly in the compose file with the `volume:` entry.

Volumes sport a good solution on a single machine locally, but they are not designed for a distributed use case. While Docker Swarm synchronizes configuration data through all nodes, volumes are created locally on each node. This means if a machine with an `openhab_data` volume fails, it can be restarted on another node, but all user data stored previously in the volume is lost.

One solution to overcome this issue is to use a network storage solution like NFS. This central network storage can then be used by Docker with the help of a volume plugin as the backend for it's Docker Volumes. This allows to store the data in a separate location in the network. In case a machine stops working, a replacement can simply access the same data from the network storage. While this provides us with a nice solution in case of a system failure, it relies heavily on a proper network connection with high transfer rates. All data needs constantly be moved across the network for each machine, generating a high load on the network. When the network (or the connection to the storage server in general) fails, the applications will not be able to access their data.

An alternative to network storage for this use case are software defined storage (SDS) solutions like *Ceph* or *GlusterFS*. Similar to network storage they are attached to Docker using a volume plugin. SDS solutions allow to define clusters of connected machines as a single storage target, but the data is replicated between all machines. This allows to have a replica of all data locally. In case of a failure, a new machine would also be added to the cluster and receives a local copy of all needed data to mount the volumes again. While this solution is better in terms of network requirements (changes are written locally and then mirrored), it still depends on a solid network. Own tests have shown that e.g. GlusterFS relies on a consent of replicas, this means when a single machine in a 3-machine cluster is isolated (by loosing connection to the other two) it will stop working even locally as it cannot replicate it's data any more.

Another way to deal with possible failures is to accept and define a certain tolerance of errors, e.g. data has to be available from at least the day before (Coulouris et al., 2002, p. 643). In that case the usage of classical backups can be a feasible solution. As Docker keeps its data in volumes, it is easy to locate all important files. *Volumerize* is a backup utility made for Docker Volumes and is deployed as a Docker container itself. It uses the robust tool *duplicity* as its backend and allows to define backups for Docker Volumes which are stored either locally or in a cloud storage like Amazons S3. It also is able to directly restore the data to a volume again. To ensure this data is available when a machine gets replaced, this can be combined with a synchronization solution like *unison* which keeps the backup data in sync between all machines (see Figure 4 *(b)*). This is only necessary when using a completely local backup though, when using cloud storage this would not be necessary. The advantage compared to the other solutions above is, that network access is only needed when synchronizing backups, not during normal operation, having a minimal impact and little needs towards the network. Another advantage of a backup solution is, that it also allows to move back to older revisions of the data, e.g. in case a change caused an error and is not reversible. This solution (volumerize and unison) is also completely defined in the Docker environment and as such does not rely on any external services.

To sum it up user data is essential to be able to continue operation after a system failure. A local backup and sync setup can be a feasible solution, especially in case of smaller setups and less reliable infrastructure. Network Attached Storage, SDS or a combination of both are a great solution when working in an environment where such solutions are already available (e.g. universities with a proper virtual infrastructure). They add the advantage of an easier and more straightforward failure recovery and should be used when possible.

4.3 Conclusion

This work pointed out that public buildings sport additional needs, not covered by a usual home system. Security and maintainability are key drivers towards a simple and cost effective solution. A complete openHAB stack consists of more parts then openHAB itself. This makes the setup and maintenance extensive, especially when considering multiple buildings. Security needs also special care as additional network communication is needed.

Using cloud technologies like Docker (swarm) can help to meet these needs. Container networking can help to create a more secure separated networking environment, while swarm definitions can reduce maintainability by collecting needed information at a central position. Using Docker Config together with a backup solution like volumerize (synced with unison) helps to maintain a higher availability and easier recovery.

While a local backup is a feasible solution in a simple environment, if better infrastructure is available it should be used. Network storage and cluster file systems can help to gain an even higher availability by removing the need of reinstalls and recovery in many cases.

The described solution of this work can be accessed on GitHub (Dobler, 2018) with further details on how to use it. While offering a first basic starting point, it needs to be extended. Additional tooling to ease the usage, as well as further components like a central log output and failure notifications are missing and should be added.

Further development of the used applications can also be helpful and enable a proper distributed setup. The available UIs for now are only aware of their own openHAB instance. Adding possibilities to switch between them or combine them in a single UI could be added. The backend could also be made aware of other instances, e.g. synchronizing configuration data between instances can make failover recovery even easier. Adding compatibility with cloud native configuration stores could also be an option.

4.4 References

Church, M., n.d. Docker Reference Architecture: Designing Scalable, Portable Docker Container Networks. URL `https://success.docker.com/article/networking`. [Accessed 08th August 2018].

Cisco Systems, 2018. Cisco Energy Management Suite - Product page. URL `https://www.cisco.com/c/de_de/products/switches/energy-management-technology/index.html`. [Accessed 20th February 2018].

Coulouris, G., Dollimore, J., and Kindberg, T., 2002. *Verteilte Systeme - Konzepte und Design*. Pearson Studium, 3., überarbeitete Auflage edition.

David, J. S., Schuff, D., and St. Louis, R., Jan. 2002. In: . *Commun. ACM*, 45(1):101–106. ISSN 0001-0782. doi: 10.1145/502269.502273.

Deutsche Telekom AG, 2016. QIVICON Medieninformation - Gemeinsam wachsen: Kommunikationskonzern baut mit Partnern Europas führende Smart Home-Plattform aus. URL `https://www.qivicon.com/de/meta/presse/gemeinsam-wachsen-kommunikationskonzern-baut-mit-partnern-europas-fuehrende-smart-home-plattform-aus/`. [Accessed 7th May 2018].

DIN EN 15232-1:2017-12, 2017. *Energieeffizienz von Gebäuden - Teil 1: Einfluss von Gebäudeautomation und Gebäudemanagement - Module M10-4, 5, 6, 7, 8, 9, 10*. Beuth Verlag.

Dobler, A., 2018. openHAB Public Building Stack. URL `https://github.com/Dobli/openhab-pb-stack`. [Accessed 30th September 2018].

Forrester Consulting, 2017. Containers: Real Adoption And Use Cases In 2017. URL `https://i.dell.com/sites/doccontent/business/solutions/whitepapers/en/Documents/Containers_Real_Adoption_2017_Dell_EMC_Forrester_Paper.pdf`.

Fu, S., Shih, C.-Y., Jiang, Y., Ceriotti, M., Huan, X., and Marrón, P. J., 2017. In: . *CoRR*.

GfK, 2016. GfK Future of Smart Home Study. URL `https://www.gfk.com/fileadmin/user_upload/dyna_content/GB/documents/Innovation_event/GfK_Future_of_Smart_Home__Global_.pdf`.

Hentschel, R. and Leyh, C., 2016. In: . *HMD Praxis der Wirtschaftsinformatik*, 53(5):563–579.

Martin, C., February 2018. Speakers Pushing Smart Home Market To 800 Million Devices. URL `https://www.mediapost.com/publications/article/313842/speakers-pushing-smart-home-market-to-800-million.html`. [Accessed 20th February 2018].

mqtt.org, n.d. Frequently Asked Questions - MQTT Homepage. URL `http://mqtt.org/faq`. [Accessed 1st July 2018].

openHAB Community, n.d.a. Securing access to openHAB - openHAB User Manual. URL `https://docs.openhab.org/installation/security.html`. [Accessed 21st February 2018].

openHAB Community, n.d.b. openHABian - openHAB User Manual. URL `https://docs.openhab.org/installation/openhabian.html`. [Accessed 17th February 2018].

Pham, L. M., Rheddane, A. E., Donsez, D., and Palma, N. D., 2015. In: . *Annals of Telecommunications - annales des télécommunications*.

SoftGuide GmbH, 2018. Aktuelle Marktübersicht - CAFM (computer aided facility management) und Gebäudemanagement. URL `https://www.softguide.de/software/facility-management-cafm`. [Accessed 20th February 2018].

Tanenbaum, A. S. and van Steen, M., 2008. *Verteilte Systeme - Prinzipien und Paradigmen*. Pearson Studium, 2., aktualisierte Auflage edition.

techuk, 2017. State of the Connected Home 2017. URL `https://www.techuk.org/component/techuksecurity/security/download/11743?file=The_Connected_Home_-_FINAL_PUBLISHED.pdf`.

trend:research, 2015. Smart Building – Intelligente Gewerbe- und Industriegebäudeautomation in Deutschland bis 2025.

Ultimo Software Solutions, 2018. Facility Management Software - Product page. URL https://www.ultimo.com/de/software/facility-management-software. [Accessed 20th February 2018].

van Laak, C. and Schröder, A., 2017. Sanierungsstau an Schulen - Wenn der Putz von den Wänden bröckelt. URL http://www.deutschlandfunk.de/sanierungsstau-an-schulen-wenn-der-putz-von-den-waenden.724.de.html?dram:article_id=406399. [Accessed 17th June 2018].

Wlotzka, V., 2016. Implementierung von MQTT Services in openHAB. Hochschule Düsseldorf, Fachbereich Elektro- & Informationstechnik.