Remote Deployment of a JADE Agent in Docker

Dennis Maecker, Henning Gösling, and Oliver Thomas

German Research Center for Artificial Intelligence (DFKI), Germany
{dennis.maecker,henning.goesling,oliver.thomas}@dfki.de

Abstract. This work presents a technical introduction into the implementation process of a containerized multi-agent system. More specifically, the JADE framework is used as a middleware for the development of software agents. With the goal of achieving high modularity and enhancing usability, the system will be containerized in Docker. To model an application-oriented scenario, the containerized agent-system is deployed on a headless remote server. The goal of this paper is to provide a comprehensive solution to help overcome the technical difficulties encountered in accessing the graphical user interface of agents from an end device. A procedural guide to the implementation process is provided, including the preparation of the JADE-based multi-agent system, creation of Docker containers, and deployment of the containerized multi-agent system on a remote server.

Keywords: Multi-agent system · JADE · Docker · Containerization

1 Introduction

Recently, the novel concept of a *Smart Managed Freight Fleet* was introduced [5], focusing in particular on multi-agent systems applied to supply chains in the transport market. Representing a promising framework for multi-agent systems, the Java Agent Development Framework (JADE) [3, 4], already applied in real-world telecommunication applications [8], was chosen as a candidate for the implementation process. As a Java-based middleware, JADE is used to incorporate functionalities of multi-agent systems and follows the standards of the Foundation for Intelligent Physical Agents (FIPA). In the scenario of a large-scale freight-fleet management system, it is crucial to ensure not only the interoperability of the agents, but also compatibility with different platforms, for instance remote servers or software enabled fleet assets, and facilitate the deployment of the software. For this objective, the container virtualization technology of Docker [1] was chosen. In the progress of adapting the multi-agent system for the use in modular Docker containers, several technical issues emerged. So far, deploying JADE agents in Docker containers was only sparsely investigated or documented at the time of writing. Thus, this paper documents the basic procedure in order to make a multi-agent system (MAS) available through Docker containers on an arbitrary machine. Furthermore, the deployment of containerized agents on a remote server is explained, with a focus on the accessibility of the graphical interface by local machines.

2 Pre-Requirements

2.1 Implementation of a JADE-based Multi-Agent System

A JADE-based multi-agent platform typically consists of one main container and additional agent containers, each including an arbitrary number of agents. Agent containers can register at the main container, which acts as an administration instance of the multi-agent platform. In the implementation process, a universal `Container` class was set up. By passing arguments (`MainContainer` or `Agent`) to the execution call of this class, it can be determined whether the instance acts as a JADE main container or an agent container. For a main container, the IP address of the hosting machine is required as an argument, while an agent container necessitates the agent's name and an existing main container's IP address. In the latter case, the `Container` class then initializes a predefined agent instance within the agent container. As the implementation of the MAS in Java depends on auxiliary libraries (e.g., the JADE framework) it is crucial to ensure dependency resolution with the aim of a portable software. This can be achieved by packing the compiled Java classes in a Java archive (i.e., a `.jar` file). A `META-INF` folder with the meta-information file `MANIFEST.MF` must be generated, as depicted in Figure 2a. This file specifies the project's main class, `Container`. The packing process of the Java archive file can be undertaken in various ways, depending on the development environment that is being used. After the creation, the `.jar` file can be executed for testing purposes in the console by the following command (assuming a locally installed Java environment) considering the local IP of the host machine: `java -jar SmartTransport.jar MainContainer 192.168.0.125`. Conversely, the command `java -jar SmartTransport.jar Agent agent1 192.168.0.125` ensures that the instance of the `Container` class acts as an agent container, instantiating an agent with name `agent1` and connecting to the main container with the corresponding IP address given in the argument.

2.2 Setting up an X Server

As the common interface of a Docker container is console-based, this section addresses the necessary steps for accessing the graphical interface of Docker containers on the host machine. This can be achieved by setting up an X server on the host machine, which manages the graphical output of the Docker container. Depending on the operating system, several solutions are possible, such as X.Org for Linux, VcXsrv[1] for Windows, or XQuartz[2], based on X.Org components, for MacOS. In either case, after the X server is started, the Docker container merely needs the local IP address of the host machine as an environment variable in order to forward the graphical interface to the X server. This is explained in further detail in the next chapter. Moreover, it is important to allow access to

[1] Available at: `https://sourceforge.net/projects/vcxsrv/`
[2] Available at: `https://www.xquartz.org/index.html`

the X server on the host machine. In the case of VcXsrv, this can be accomplished in the program's settings. Considering the X.Org server on Linux, it is sufficient to enter the following command in the console: `xhost +`. It is important to note, that this procedure opens the X server for all participants in the network. In high security environments, this approach is not recommended.

3 Agents on a Remote Server

The containerization of the agent system is split into two parts. First, a base image is set up in a `Dockerfile`. In a separate `docker-compose.yml` file, the establishment of the Docker-related network setup takes place. Also, two containers will be specified, one which runs a JADE main container and one running a JADE agent container. As the host of the Docker containers (i.e., the headless remote server) has no support for graphical output by itself, it has to be ensured that the graphical user interface can be accessed from an end device through a Secure Shell (SSH) connection to the server.

3.1 Accessing the User Interface of Remotely Deployed Agents

First, the remote server is accessed by an SSH connection. It is possible to run graphical applications on the remote server and have the graphical output displayed on the local machine utilizing the X server running on the local system. This can be achieved by passing an `-X` argument to the call of the SSH connection as follows: `ssh -X user@remote-server`. Alternatively, when using SSH-clients such as PuTTY, it is possible to configure the forwarding task to the X server in the application settings [7]. However, passing the graphical output of an agent application inside a Docker container through the remote server to the end device requires a more elaborate solution. The most promising approach for forwarding the user interface of the containers to the end device will be further described and can be summarized by the installation of a Virtual Network Computing (VNC) server in the Docker container. When connecting to the remote server via SSH it is possible to connect to the Docker container using a VNC client installed on the server. The graphical output of the VNC client is then tunneled via the SSH connection and displayed on the local end device. This concept is depicted in Figure 1. The substantial advantage of this approach is that the application continues its operation even under interruptions of either the SSH or the VNC connection. Additionally, as the graphical output is tunneled through the SSH connection to the end device, there is no need for additional ports being accessed on the remote server, hence contributing to the data security of the system.

3.2 Setting up the Docker Container

The `Dockerfile` for the Docker image used in this part is shown in Figure 2b. The base image used for this system is an `openjdk:18-jdk-slim` image (line `1`) that is a minimal, Debian-based environment containing a pre-installed

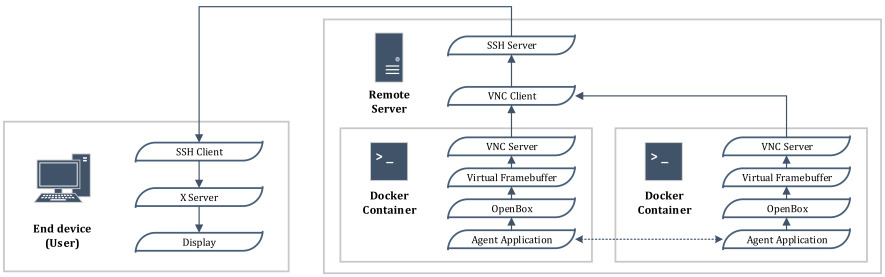

Fig. 1: Data flow of the graphical information (solid lines) as well as data exchange in between the agent applications (dashed line).

Java distribution. Subsequently, in line `2` the `.jar` Java archive for the Java agent application is copied to the `home` directory of the Docker image. As stated before, the `.jar` file can be started with different arguments, thus launching either a JADE main container or a JADE agent container. Hence, the Docker image which is created here can be used for both types of JADE containers.

In line `3`, the installation of basic packages related to the connection to the X server along with a VNC server (`x11vnc`) and a minimal desktop environment (`openbox, tint2, xterm, lxterminal`) takes place. Lines `4-6` set up the desktop environment. This process is inspired by the descriptions in a publicly available repository [6]. The last command in line `7` refers to the `docker-entrypoint.sh` script that is executed when the container is launched. The content of this `docker-entrypoint.sh` file is shown in Figure 2c and is responsible for the initialization of the desktop environment and the window manager. Finally, the Docker image can be built using the following command: `docker build -t dfki/agents`. By using the argument `-t`, a tag name for the Docker image, here `dfki/agents`, can be set, which is then further used as a base image in the `docker-compose.yml` file in Figure 2d.

3.3 Integration in a docker-compose File

The `docker-compose.yml` file describes the setup of a main container and an additional agent container, both based on the JADE framework. In lines `2-9` of the file, a network gateway is defined. Thus, it is possible for each Docker container to possess its own IP address within this separate network. This ensures portability of the Docker system, as the IP addresses do not have to be adapted to the address of the respective host machine. Beginning in line `11`, the configuration of the Docker container running the JADE main container is specified. Lines `12` states the base image as the one described by the Dockerfile in Figure 2b. In line `15`, the IP address of this container is set. Following, lines `16-19` define the ports of the Docker container which are mapped to the ports of the host

```
1: Manifest-Version: 1.0
2: Main-Class: smartTransport.Container
```

(a) Content of the `MANIFEST.MF` file.

```
1: FROM openjdk:18-jdk-slim
2: COPY ./out/artifacts/SmartTransport/
   /home/
3: RUN apt-get update && apt-get -y install
   iproute2 x11-apps libxi6 libxtst6
   libxrender1 lib32z1 xvfb openbox tint2
   x11vnc xterm lxterminal
4: ADD config /opt/config
5: RUN chmod +x
   /opt/config/docker-entrypoint.sh
   /opt/config/openbox/autostart
6: RUN rm -rf /etc/xdg/openbox && cp -R
   /opt/config/openbox /etc/xdg/openbox &&
   (rm -rf /etc/xdg/tint2 || true) && cp -R
   /opt/config/tint2 /etc/xdg/tint2
7: CMD "/opt/config/docker-entrypoint.sh"
```

(b) `Dockerfile` for containerization of the multi-agent system intended to be deployed on a remote server.

```
1: #!/bin/bash -xe
2: xvfb-run -s "$DISPLAY" -s '-screen 0
   1024x700x24 -ac' openbox-session
```

(c) `docker-entrypoint.sh` file.

```
1: version: "2.4"
2: networks:
3:  agent_network:
4:  driver: bridge
5:  ipam:
6:   driver: default
7:   config:
8:   - subnet: 192.168.56.0/24
9:   gateway: 192.168.56.1
10: services:
11:  container:
12:   image: dfki/agents
13:   networks:
14:    agent_network:
15:     ipv4_address: 192.168.56.10
16:     ports:
17:     - 1099:1099
18:     - 7778:7778
19:     - 5900:5900
20:   working_dir: /home
21:   environment:
22:    DISPLAY: :99
23:  agent1:
24:   image: dfki/agents
25:   networks:
26:    agent_network:
27:     ipv4_address: 192.168.56.11
28:     ports:
29:     - 1100:1099
30:     - 7779:7778
31:     - 5901:5900
32:   working_dir: /home
33:   environment:
34:    DISPLAY: :99
```

(d) `docker-compose.yml`-file used for launching one JADE main-container and one JADE agent container in dedicated Docker containers.

Fig. 2: Contents of relevant files used in this work.

machine. Additionally to the ports relevant for the JADE communication (`7778` and `1099`), port `5900` is opened to allow a VNC connection by the remote server, i.e., the host of the Docker container. The display environment variable in line `22` is set to `:99`, ensuring that the graphical output is sent to the VNC server. The part after line `23` represents the setup of a second Docker container, dedicated to a JADE agent container. The configuration procedure is analogous to that of the first container. Additionally, the setting of the IP address in line `27` and the port mapping in lines `28-31` differ. The ports need to be altered for this container as the initial ports on the host machine are already used by the first container. If more containers for other agents are intended to be launched than there are specified here, the respective ports have to be altered accordingly (e.g., 1101, 7780, 5902). The file can be built using the following command on the remote host: `docker-compose up`. After this, the two docker containers are running on the remote host.

3.4 Launching the JADE Agent System

The last section described how to start the two Docker containers containing the `SmartTransport.jar` that can launch either a JADE main container or an agent container. At the time of writing this paper, no solution has been found yet to run the agent application automatically when the Docker containers are starting. Rather, it is necessary to connect to the running Docker containers via a VNC client and to manually start the agent application. After setting up the SSH connection and enabling graphical forwarding to the local X server, the VNC client (e.g., vncviewer) can be launched on the remote host, displaying the VNC software user interface on the end device (see data flow in Figure 1). In the user interface of the VNC client, the IP and the port of the respective Docker container can be entered, in this case either `192.168.56.10:5900` or `192.158.55.11:5901`. In the following, the Docker container with the IP `192.168.56.10` is used to launch a JADE main container, while the other Docker container with the IP `192.158.55.11` is dedicated to the JADE agent container. A successful connection will exhibit the respective Docker container's desktop environment within the VNC client's interface. In the environment of the Docker container, it is possible to launch a terminal instance (here `lxterminal`, see installation setup in Figure 2b). A JADE main container can be started by entering the following command: `java -jar SmartTransport.jar MainContainer 192.168.56.10`. When the main container is started, the graphical user interface of the JADE framework can be accessed in the interface of the VNC client. Subsequently, a VNC connection to the second Docker container can be established from the remote server. In a terminal instance, a predefined agent named `agent1` can be started in a JADE agent container by following command: `java -jar SmartTransport.jar Agent agent1 192.168.56.10`. As described before, by passing the IP address of the JADE main container as an argument to the call of the JADE agent container, it is possible that the JADE agent container connects to the JADE main container. Hence, an arbitrary number of agents can connect to the JADE main container to form a MAS. The agent applications stay active, regardless of whether the VNC connection to the Docker containers or the SSH connection to the remote server is active.

4 Conclusion and Further Work

In this paper, we have presented an introduction to how to combine the well-known JADE middleware for MAS with Docker focusing on the deployment on a remote server. To the best of the authors' knowledge, a comprehensive introduction to this topic, as presented here, has not yet been published. Future developments in this regard can be identified for instance by configuring the JADE message protocol (MTP) to utilize HTTP [2]. Hence, JADE platforms distributed over several networks can be connected by using JADE's feature of remote platform connectivity or by implementing REST interfaces.

References

1. Anderson, C.: Docker [software engineering]. IEEE Software **32**(3), 102–c3 (2015)
2. Bellifemine, F., Caire, G., Trucco, T., Rimassa, G., Mungenast, R.: Jade administrator's guide. TILab (2003)
3. Bellifemine, F., Poggi, A., Rimassa, G.: JADE - A FIPA-compliant agent framework, pp. 97–108. The Practical Application Company Ltd. (1999)
4. Greenwood, D., Bellifemine, F.L., Caire, G.: Developing Multi-Agent Systems with JADE. WILEY (2007)
5. Heinbach, C., Gösling, H., Meier, P., Thomas, O.: Smart managed freight fleet: Ein automatisiertes und vernetztes flottenmanagement in einem föderierten datenökosystem. HMD Praxis der Wirtschaftsinformatik (2022)
6. murer: Virtual x and vnc server docker image with openbox. `https://github.com/murer/docker-xvfb-x11vnc-openbox` (2020)
7. Tatham, S.: PuTTY User Manual (2022), `https://upload.wikimedia.org/wikipedia/commons/b/b7/PuTTY_User_Manual.pdf`
8. Ughetti, M., Trucco, T., Gotta, D.: Development of agent-based, peer-to-peer mobile applications on ANDROID with JADE. In: The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies. IEEE (2008)