Deployment of automatically generated interface translators in Arrowhead Framework
Problems and ideas for possible solutions

Erik Serrander

July 22, 2021

1 Introduction

Interoperability is a key part of Arrowhead Framework, however it is often hindered by a multitude of protocol, naming and syntactic mismatches. To solve these problems considerable engineering time is spent on finding the mismatches, rewriting systems and writing interfaces. In order to reduce the engineering time associated with the interoperability issues, the autonomous generation of interface translators (ITR) has been proposed.

When an ITR has been generated it needs to be run on a device. However running it on a central server increase delays and introduces security problems. To minimize the delay and mitigate these security problems, the ITR may run on either the consumer or the provider of the service translated. A new deployment approach have been analyzed and proposed to address the aforementioned problems without in a meaningful way lowering QoS for the rest of the system. The generation and mismatch detection is outside the scope of this report.

2 Problem formulation

To address the problem of deploying an ITR four research questions (RQs) have been formulated.

1. Which device is the best option to deploy the generated interface to?
2. How can it during run time be determined if the generated interface could be run on the device or not?
3. How can the generated interface securely be transferred to the device?
4. How can the generated interface securely be run on the device?

3 Proposed solution for each RQ

In this part the proposed solution to each question is stated. For every solution validation method, validation result and status is described. Furthermore drawbacks or problems with the proposed solution is stated and discussed and alternative solutions are mentioned.

3.1 Solution to RQ 1

The first step in deploying an ITR is to determine the target device to deploy to. The simplest solution is to always deploy to the consumer. However this is not always possible or desirable depending on the capabilities of the consumer device. Therefore a solution that chooses the most capable target device is proposed and discussed in this section.

First, the proposed solution is explained. Followed by validation of, problems and drawbacks with, alternatives to and further work on the proposed solution.

3.1.1 Proposed solution

The proposed solution on how to determine which device to deploy the interface to is as follows:

1. Check both consumer and provider device with the checker described in section 3.2.1. If only one of the devices is determined to be able to run an ITR then choose this device and stop there.

2. By looking at the ram size from the two devices metadata and choose the one with the larger amount. If this information is not available choose one at random.

Step 2 in the solution can be changed to use some other part of the metadata or several together as a weighted sum. It is to be tested which way is best.

3.1.2 Validation of the solution

To validate the proposed solution, this one is implemented in java code and integrated into the ITR generator. Next it is tested in a arrowhead local cloud, this with a multitude of devices. Under a simulated load, it is tested if it can ensure a better QoS than the simple solution of always deploying to the consumer.

3.1.3 Status

Nothing is implemented, tested or validated.

3.1.4 Problems and drawbacks

It exist one major problem with this solution and is described bellow. If more than one similar (in that they all have the same accepting interface for a service) consumers all make an orchestration request for the same service and that all of the ITRs get deployed to the same provider, then the same provider will have more then one copy of the same ITR running on it. This will if hardware resources are close to exhausted, hinder the systems operation. This problem can be solved by comparing all new generation requests to a table of active ITRs and if appropriate use one of them.

3.1.5 Alternative solutions

1. Always choosing the consumer.
2. Always choosing the provider (will have the same problem as discussed in section 3.1.4).

3.1.6 Further work

- Implement and integrate the solution.
- Validate the solution.

3.2 Solution to RQ 2

Deploying an ITR to a device can potentially exhaust all or some resources of the device. This in turn making the services running on the device unavailable or reducing their quality. To mitigate this problem a solution described in this section is proposed that reduces the risk of this problem occurring.

First, the proposed solution is explained. Followed by validation of, problems and drawbacks with, alternatives to and further work on the proposed solution.

3.2.1 Proposed solution

In Figure 1 a flowchart outlining the steps to determine if an interface is to be generated or not is shown.

In the first step, the device ID is checked against a blacklist of devices where deployment of a interface has failed in some way. The blacklist exist to avoid unnecessary deployment attempts. If the device is blacklisted an unable to generate response is sent back.

Next it is checked whether or not there is a way to deploy the ITR. How this is done depends on the deployment way. For the service approach described in section 3.3.2 it is done by making an orchestration request for the service. Then for every service in the response find one if any that run on the same device as the device the interface is to be deployed on. If there is no way to deploy nothing else is done and an unable to generate response is sent back.

Continuing it is checked if the device type of the device in question exist in the lookup table of known and tested device types. The entree for each device type answers the question: If the device belonging to this device type can run an ITR under normal circumstances? This is determined through testing and the results entered into the lookup table. The entries in the lookup table can be either `YES`, `NO` or `WITH_REQUIREMENTS`. If the entree is `YES` generation is started. If `NO` generation is not started and an error sent. If on the other hand it is `WITH_REQUIREMENTS` a list of accompanying requirements is validated. The device in question must meet all these requirements for generation to start, otherwise an error is sent.

A requirement have a name that correspond to a field in the devices metadata, a requirement type describing how the requirement is to be validated and a integer value to validate against. The requirement type can be either `EQUAL`, `MORE_THEN`, `LESS_THEN`, `EQUAL_OR_LESS` or `EQUAL_OR_MORE`. To validate a requirement the metadata of the device in question is searched through to find a field with the same name as the requirement. Then according to the requirement type the value of the requirement and the metadata field are compared. If there is no field with the same name as the requirement in the metadata then the requirement is assumed to not be met.

If the device type is not in the lookup table it is first checked whether or not it is allowed to handle a device of a unknown type as a generic device. If this is not the case an error is sent back. Otherwise the device is evaluated with a generic device type in the same way as any other device type.

The maintainer of the local cloud is responsible for maintaining the lookup table. To reduce double work of testing device types and entering appropriate values into the lookup table, a way to share the data between local clouds can be implemented. Even having a global repository of tested device types and their appropriate requirements can be of great value.

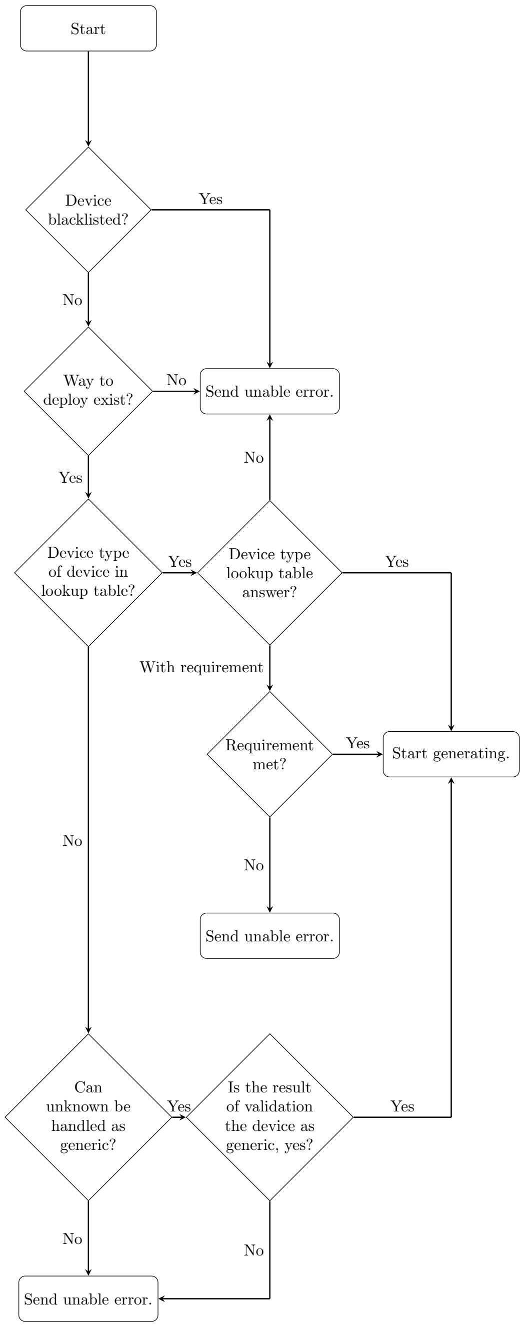

Figure 1: Flowchart describing the proposed solution to RQ 2.

3.2.2 Validation of the solution

To validate the solution, this one is implemented in java. Next the implementation is tested to properly implement the algorithm described in section 3.2.1. After this is done it is tested with real devices under a simulated load, to see that it with good values in the lookup table can hinder ITRs being deployed to devices not capable of running it.

3.2.3 Status

It is fully implemented in insecure and secure mode with dummy classes for the device and system registry endpoints. Nothing is validated.

3.2.4 Problems and drawbacks

1. All devices need to be tested and from this appropriate requirements inputted into the lookup table.
2. The system operates on device types and does not know what is already running on the device. Therefor it may make the wrong decision if the device is under unusually heavy load.

3.2.5 Alternative solutions

Here is a list in short describing alternative solutions to the problem.

1. Having a mandatory protocol for asking the device how much resources it have unused and making a decision based on this.

3.2.6 Further work

- Implement a database for the lookup table or other persistent storage.
- Replace the dummy classes with service consume calls to the real arrowhead systems.
- Integrate into the generator.
- Add support for dynamic requirements like number of systems already running on the device etc. This to alleviate problem 2.

3.3 Solution to RQ 3 and RQ 4

The solutions to RQ 3 and RQ 4 are closely connected and is therefore discussed together as a whole. To begin with this section compares two different solutions to the RQs. After this the chosen solution is discussed. For this solution it is then stated how validate it, the status of the solution, problems and drawbacks with the solution, alternative solutions mentioned and further work listed.

3.3.1 Solution comparison

There are two ways to transfer and run an ITR that has been considered, a http post request to an arrowhead service or via sftp directly to the device OS. Only the service has been implemented. Running an ITR is handled by the service for the former alternative and the later via commands sent over ssh. The main differences between the approaches is laid out in table 1.

	service	ssh(sftp)
requirements	service	ssh keys or password, a known user, a known file location and shell access
pro	using arrowhead security, less OS dependant usage, easier to detect and report errors and easier to implement on small devices	Used and mature, already on most devices and less extra software on the device
con	custom solution and adds complexity to target device	added complexity to generator system, many requirements and not using the arrowhead AA(A) system

Table 1: Assessment of the two approaches.

Looking at table 1 the service approach has many advantages over the ssh(sftp) approach. The service approaches disadvantages are connected to it being one more program to write and run. Because of the many advantages and few disadvantages with the service approach it was implemented first.

3.3.2 Implementation of the service deployment approach

To consume the service a json string is sent to it over http. The json string has 3 fields:

- ID - An id used to reference the deployed ITR later.
- port - The port number the provider part of the interface binds to.
- file - A base64 encoded string of the file to be deployed.

Before an ITR is deployed the service check for port collisions. It is the ITR generators responsibility to choose a port that does not collide with any port used by an arrowhead system on the target device. Furthermore the ITR generator is also responsible to not choose a port likely to collide with any non arrowhead system on the target device. Even if this is done there could be a collision if for example ssh is run on an non standard port. If a collision is detected an error should be sent back possibly with an suggestion for a port that is not used. Even if this is not provided the generator chooses the port with a pseudo random function from the possible unused ports. In using a pseudo random function if there is an empty port on the target device it will eventually be chosen. After the port check the ITR is deployed and when it is considered to have started a message of initial success is to be returned. If the http request times out, it is assumed that the device is unable to deploy the ITR and the device is temporarily blacklisted. At this point the deployment is done but the deployment service can send a message if for any reason the ITR stopped working. Additional monitoring and safeguards can be implemented in the service, this to mitigate issues or QoS lowering the deployed ITR might cause on other systems on the device.

3.3.3 Validation of the solution

To validate the solution, this one is implemented in java. Next the implementation is tested on a separate device. This ensures it is working. To further prove that the solution is viable in a production environment, a test were a ITR is deployed to a multitude of devices under a simulated load in an arrowhead local cloud is done.

3.3.4 Status

A service to deploy ITR in jar files is implemented and working. This without a port check.

3.3.5 Problems and drawbacks

1. An additional system needs to be written and run on the device if it is to be used in the system.

3.3.6 Alternative solutions

1. Transferring over sftp and running via sending shell commands thru ssh.
2. Sending a deploy request that does not contain the interface itself. Instead it contains a reference to it in a repository (might be a docker repository).

3.3.7 Further work

- Implement a watchdog that can end and delete a interface before the device run out of hardware resources or fails to meet time requirements.
- Implement other similar services for other runnable file types like one that can take a docker container or a binary elf file.