UNIWERSYTET GDAŃSKI
WYDZIAŁ MATEMATYKI, FIZYKI I INFORMATYKI

Piotr Lewandowski
numer albumu: 274306

Kierunek studiów: informatyka
Specjalność: ogólnoakademicka

Praktyczna samokonfiguracja w heterogenicznych wdrożeniach IoT

Praca magisterska
napisana
pod kierunkiem
dra Wiesława Pawłowskiego

Gdańsk, 2022

UNIVERSITY OF GDAŃSK
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

Piotr Lewandowski
Record book number: 274306

Field of study: Informatics
Focus: general academic

Practical Self-Configuration for Heterogeneous IoT Deployments

Master's thesis
written
under guidance of
Wiesław Pawłowski, Ph.D.

Gdańsk, 2022

Streszczenie

Internet rzeczy (IoT) jest dynamicznie rozwijającym się obszarem badań i postępu technologicznego. W niniejszej pracy magisterskiej skupiamy się na jednym z jego głównych wyzwań, jakim jest konfiguracja heterogenicznych zasobów i proponujemy autonomiczną konfigurację (ang. *self-configuration*) jako rozwiązanie. Zaczynając od sformułowania wymagań wobec oprogramowania samokonfigurującego, w zasadniczej części pracy, przedstawiamy rozwiązanie o nazwie *Self-Configurator*. Jego opis zaczynamy od charakterystyki elementarnych składników – zasobów i łączników, modeli wymagań, modeli reakcji oraz *Self-Configuration Finite State Machine*. Następnie, przechodzimy do przedstawienia dynamiki pomiędzy tymi składnikami, kończąc na krótkim przykładzie prezentującym zachowanie proponowanego rozwiązania. Aby zilustrować wykorzystanie *Self-Configuratora*, prezentujemy trzy różne scenariusze, wykorzystujące środowisko portowe do wieloetapowej konfiguracji zasobów, utrzymywania ruchu krytycznego oraz przełączania urządzeń w „tryb ratunkowy”.

Abstract

The Internet of Things (IoT) is a rapidly growing field of research and development. In this thesis, we focus on one of its main challenges, the configuration of heterogeneous resources, and propose self-configuration as a solution. We start by describing the requirements for self-configuration software, then in the core part of this work, we present a software named Self-Configurator, starting from its elementary components – resources and connectors, requirements models, reaction models, and Self-Configuration Finite State Machine – then dynamics between those components, and ending with a short example that presents its behavior. Finally, we apply the Self-Configurator in a port environment in three different scenarios: multistep resource configuration, maintaining critical traffic, and switching devices to a "rescue mode".

Contents

Chapter 1

Introduction

Internet of Things – landscape and challenges

Internet of Things can be defined as an infrastructure that builds services by interconnecting – physical and virtual – *things* based on interoperable information and communication technologies [1]. Billions of vastly heterogeneous devices are connected to a global network, varying in applications, capabilities, physical size, weight, computing power, quality, network access speed, and many more. The total value of the IoT market was estimated at 384.70 billion dollars in 2021, and forecasts predict a market size of more than 2 trillion dollars by 2029 [2]. With ubiquitous smart devices, such as smartphones, smartwatches, fitness trackers, and others *Things* of IoT are omnipresent. The relatively cheap and accessible network connection provides the foundation for the *Internet* part of the IoT term.

According to Ericsson, in 2021 around 28 billion devices were connected globally. Of these, 15 billion are involved in machine-to-machine communication [3]. This already means that more devices communicate with each other than there are humans on the planet. Currently, IoT solutions can be found in virtually all domains. Starting with governance (smart cities, smart grids) through various sectors of industry (logistics, factories, healthcare, and retail) and ending with smart strollers. It seems that all elements for a broad set of IoT applications are already available.

IoT-based systems are difficult because of the limited number of assumptions we can make about them. *Things* are heterogeneous, they do not provide a uniform API; scaling physical devices is more complicated than spinning up another instance of a containerized virtual service; in many scenarios, energy consumption is a concern. *Networks* in various IoT applications are not standardized; critical and sensitive data can be communicated, which require security considerations; connection may not always be reliable or of high enough quality. Although the challenges are numerous, we can group them “by concern” as follows: *security and privacy*, *reliability*, *scala-*

bility, *energy consumption*, and *heterogeneity*. The latter is the focus of this work.

Self-configuration idea

We can state that an autonomous system is a system that can manage (change) itself automatically with limited human interaction. The role of an operator is then limited to supervising and managing the system at a high level rather than manually deciding when and what instructions should be executed. For a system to be autonomous, it needs to be able to inspect itself and operate on its own. IBM did the foundational work in that area [4], by defining four capabilities that make a system autonomous: self-configuration, self-healing, self-optimization, and self-protection. It should be noted that this is not the only classification of self-* capabilities; others are available in [5] and [6]. In short, self-healing – refers to the automated discovery and correction of faults; self-optimization – to monitoring and control of resources to ensure optimal functioning; self-protection – to proactive identification and protection from attacks on the system; the system is defined as capable of self-configuration if it "can dynamically adapt to changing environments. self-configuring components adapt dynamically to changes in the environment, using policies provided by the IT professional".

Software support for self-configuration

As of late 2022, we are in a situation where there are billions of devices, and growing awareness, both in academia and industry, of the required tools, but there are simply no widely adopted solutions. Given the inherent complexity of IoT deployments, self-configuration appears promising. The authors – including the author of this thesis – of [7], conclude that of the 14 frameworks and tools described, *none* is capable of self-configuration. Properly implementing software with that capability could abstract out problems of heterogeneity and constrain the unstable nature of a system, which can change over time and consists of multiple physical and virtual components. This thesis tries to fill these gaps by developing software that would be able not only to show self-configuration capabilities itself but especially endow IoT deployments with it.

Chapter 2

Implementing self-configuration

2.1 Clarifying implementation objective

To reliably implement nontrivial software, *requirements specification* needs to be defined. There are numerous approaches (with varying levels of detail and complexity involved) to tackling this problem. In the scope of this work, we will limit ourselves to a high-level overview of the desired capabilities of an application that would realize self-configuration as defined in [4]. Let us sketch a simple scenario for a smart home, to help us with extracting the specific requirements.

Consider homeowners who have a thermostat that controls an air conditioner and an underfloor heating system. To keep themselves safe, they introduced a security system that includes cameras and thermal scanners. They also have a local computer that handles all network traffic and has access to all devices. In addition, it has special software to call the police. In case a burglar cuts the power cables, they connected a set of backup batteries. Unfortunately, the batteries are not powerful enough to keep all devices up. Batteries can keep the computer and network up, but only one of the following: thermostat (with A/C and heating system), cameras, or thermal scanners. With this in mind, homeowners configured their system so that in case of a power outage *only* computers and cameras should be on, to record potential burglars and send a notification to a local police station.

From this story, we can extract the desired capabilities of self-configuring software.

- *Administrator* can define *configuration* of a system. The *configuration* describes what *resources* are required by a specific *functionality*
- *Administrator* can define *reactions* – rules on how the system should behave when a specific *action* has happened. *Reactions* can modify existing *configuration* and emit notifications.

- The system should have the ability to autonomically decide which *functionalities* will be kept in the event of limited available *resources*.
- *Resources* can notify the system about changes to its state; at least about going live and going down.
- The system needs the ability to send messages to the resources.

Before moving on, we need to focus on the vocabulary introduced above, that is, *administrator*, *configuration*, *functionality*, *resource*, *action*, and *reaction*. *Administrator* is a user that can define functionalities and reactions. *Configuration* is a set of all *functionalities. Functionality* can be understood as a capability of a system. In the scope of self-configuration, we will limit our view to the *requirements* of a particular functionality. We can do this because the self-configuration system will not execute or act on those functionalities directly. It will simply ensure that all the components required for the functionality to be available are available.

Hence, in this work, functionality can be defined as a set of required resources and other (sub)functionalities. In the context of the example, a smart home (functionality) requires a thermostat (functionality) and security (functionality). The latter requires cameras (resources) and thermal scanners (resources). The functionalities and resources form a directed acyclic graph. The direction of an edge might mean "is required for". For example, a camera is required for security functionality. Acyclic nature of a graph excludes dependencies referencing each other. This implies that resources are vertices without outgoing edges, whereas functionalities are vertices that may or may not have outgoing edges. As the functionality may depend on other resources and functionalities, we can observe that it forms a graph in which resources would be vertices without outgoing edges. Functionalities may or may not have outgoing edges.

The distinction between resource and functionality may seem arbitrary, but, practically speaking, changes to a resource will notify the system via action and potentially trigger a reaction, whereas changes to functionality will not do that. *Resource* is a Thing – physical or virtual – in the context of the Internet of Things. It can range from a virtual representation of a complex system through an external software service, to a simple sensor. *Action* is an external event about which the system that realizes self-configuration is notified. Actions always describe some change to a resource. At a minimum, the self-configuration system needs to handle resources going live (registering to the self-configuration system) or down (deregistering from the self-configuration system). *Reaction* is an operation that the system performs after an event has occurred. The reaction can modify the system configuration or communicate messages to resources.

The goal of this work is to design and develop software that will realize the aforementioned self-configuration capabilities. We will call this system

Self-Configurator.

2.2 Selecting technologies

2.2.1 Guiding principles

There are multiple ways to implement software according to Section 2.1. The selection of technology for any software project must consider multiple factors, including, but not limited to, the size and complexity of the project, performance, timeline, available tooling, and team expertise. To select the technology stack for this particular project, we now explicitly state the guiding principles.

- Innovation – this is a research project, therefore we are interested in solutions that are novel and push the boundaries (as of late 2022).
- Strong typing – we prefer solutions that can catch errors during the compile time and give tools to limit errors (by typing) that the user and the developer can make.
- Ease of integration – it is important to move fast; hence, we want solutions that can easily integrate with existing solutions and/or have rich ecosystems themselves.

The selection presented below is influenced by our personal preferences and experiences, making it biased. However, the technologies were chosen to comply with the principles mentioned above.

2.2.2 Technologies used in Self-Configurator

Scala 3. Scala is a programming language that supports both object-oriented and functional programming. It can be run on a Java virtual machine or compiled to JavaScript and run in a browser. The latest iteration of Scala language – Scala 3 – was released in May 2021 and it is just being adopted by the industry. The language has strong static typing, which can support exhaustive pattern matching, opaque data types, and type classes. Lastly, access to the JVM ecosystem, as well as Scala's own ecosystem, makes it easy to integrate with various tools.

Akka and Akka Persistence. Akka is a set of libraries for Scala and Java that implements an actor-based concurrency model. It provides abstractions for both message-driven (via communicating actors) and stream-based applications. It is one of the most popular Scala frameworks. It is widely used in both academia and industry [8].
Akka Persistence enables event sourcing for Akka's actors. It realizes it by

providing Finite State Machine inspired abstraction for actors. By default, it stores only events that change the internal state of the actors, not the state itself [9]. Akka is being actively developed, and it already provides support for Scala 3. Akka Persistence FSM-based abstraction together with an exhaustive pattern matching (Scala feature) can ensure that all defined scenarios are handled. With native support for Scala 3, it is easy to integrate with two exceptions. First, Akka Persistence JDBC, which is required for "out-of-the-box" integration with JDBC-compliant databases, does not have support for Scala 3 yet, which effectively limits datastore selection. Second, Akka HTTP – it is simply not compatible with Scala 3, because it uses Scala 2 macros in its test code. That makes Akka HTTP unusable in Scala 3 (as long as one wants to write test code). Because of that, we decided to use http4s as a http library.

Cats and Cats Effect. From Cats homepage: "Cats is a library which provides abstractions for functional programming in the Scala programming language" [10]. It provides a broad scope of type classes and data types, making it easier to express code with functional idioms. Cats libraries take advantage of the Scala compiler and allow more type-safe code to be written. It is written in Scala 3 and for Scala 3, so integration causes no friction.

Circe. Circe is a JSON library for Scala. Circe provides tools to create type-safe codecs for JSON serialization and deserialization. Codecs are generated during the compile time [11]. Circe has Scala 3 support.

EventStoreDB. EventStoreDB is a NoSQL database optimized for Event Sourcing. It stores data as an immutable series of events [12]. As mentioned previously, JDBC-compliant databases were out of the picture, but EventStoreDB provides a plugin that integrates with Akka Persistence and Scala 3 automatically.

Apache Kafka. Kafka is a distributed event streaming platform that provides publisher-subscriber mechanics [13]. It is the most popular solution of its kind in manufacturing, banking, insurance, and telecom companies. Apart from the publisher-subscriber model, Kafka can store streams of events durably and process streams of events either online or offline. Being a popular solution, various Scala 3 libraries provide integration with Kafka.

2.3 Self-Configurator

In this section, we will present how the Self-Configurator is built, how the components interact with each other, and then dive deeper into the code.

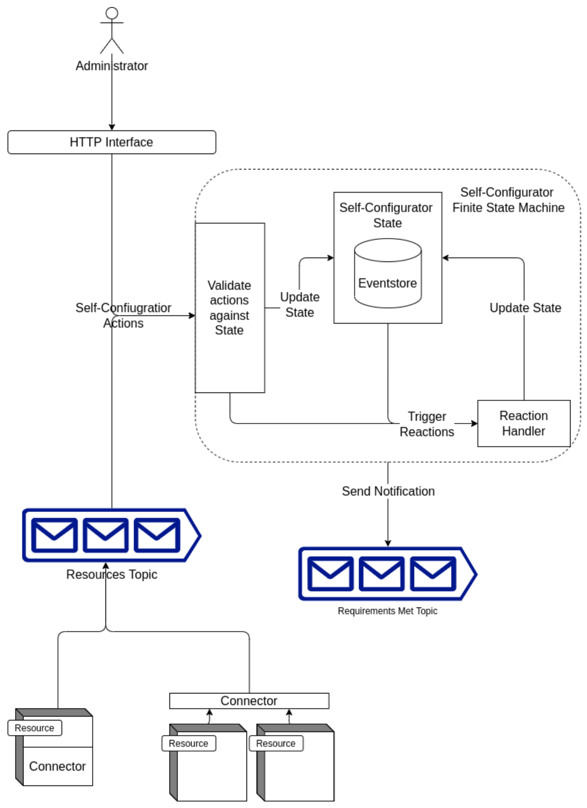

Figure 2.1: Simplified Self-Configurator component graph

Now, let us walk through the graph of Self-Configurator components presented in Figure 2.1. We can see two different groups of resources. One has a built-in connector and the other has an external connector. This reflects that there are resources of various computing capabilities. Some devices will be able to publish messages to the "resource" Kafka topic directly; some will need an external solution that will endow them with that capability. For the latter, for instance, we can think of a local gateway that communicates with low-resolution, simple cameras. Cameras are not computationally powerful enough to send and read messages from a Kafka topic, but they can delegate this responsibility to a gateway.

"Resource" Kafka topic messages are mapped to Self-Configurator actions and passed to Self-Configurator Finite State Machine. Messages coming through this channel describe changes in status of resources (that is, resources are available or resources are not available). There is an additional message type that can be sent on this topic, *custom message*, which is described later in the text.

The administrator can manipulate two special entities via the HTTP interface – they can create or delete requirement models and reaction models. Both *requirement model* and *reaction model* refer simply to the code-level representation of the concepts introduced in Section 2.1. These HTTP calls are mapped to particular Self-Configurator actions. Every action is validated against the current FSM state. Only those that would not result in an erring state will be allowed to be processed further. After the FSM is successfully updated, the reaction may be triggered if the predefined conditions in the reaction model are met. Reactions themselves can update the state or send a custom Kafka message on a predefined topic. After every state update, a notification is published to the "requirements met" Kafka topic with information on whether the registered resources are enough to meet the demands of all defined requirements models.

In the following sections, we will dive deeper into particular components and entities; hence some Scala knowledge is assumed. Note that the code listings below are abbreviated/simplified to increase readability.

2.3.1 Basic building blocks

Before diving into particular components, let us take a look at the basic techniques and data structures used in the Self-Configurator. Along the way, we will build an understanding of the selected tools. We will go through opaque types, `cats.data.NonEmptySet`, and `cats.data.Validated`.

Opaque types. When working with statically strongly typed languages, we often define new types as wrappers of existing types, rather than inventing completely new types. In many cases, this mechanism involves overhead in terms of creating these new types, accessing their fields and methods. The benefit of this approach can be illustrated in the following example taken from the banking industry. The example is selected to highlight the potential danger.

Let us define a simplified class representing accounts along with a simple stateless service to transfer money from one account to another. Before we look at the code, we need to introduce two concepts, debit and credit. Oversimplifying, we can say that debit records the sum of the money flowing into the account, and credit records the money that is flowing out of the account.

```scala
case class Account(id: String, debit: Double, credit: Double):
  lazy val balance: Double = debit - credit

object MoneyTransferingService:
  def transferMoney(
    from: Account,
    to: Account,
    amount: Double
  ): (Account, Account) = {
    (Account(from.id, from.debit, from.credit + amount),
      Account(to.id, to.debit + amount, to.credit))
  }
```

We can now imagine a devastating effect of a simple typo mistake, when we put the debit argument in credit's place or the other way around. In any reasonable application, this would be caught by a test, preferably early in the test pipeline. With type aliases, we could refactor this code so that we could catch such errors during the compilation time, making it harder to write faulty code. Please note that the code below is a simplification used to present opaque types and would not compile yet. Writing the code in this way ensures that the logic for `transferMoney` is valid at the compile time.

```
case class Account(
  id: AccountId,            // alias for String,
  debit: AccountDebit,      // alias for Double
  credit: AccountCredit,    // alias for Double
):
  lazy val balance: AccountBalance = debit - credit
  // AccountBalance is an alias for Double as well

object MoneyTransferingService:
  def transferMoney(
    from: Account,
    to: Account,
    amount: TransferAmount // alias for Double
  ): (Account, Account) = {
    (Account(from.id, from.debit, from.credit + amount),
      Account(to.id, to.debit + amount, to.credit))
  }
```

Now we can explain the concept of overhead from two different perspectives, developer's and runtime's. In the case of `opaque type`, by stating that they have no overhead, Scala means that there is no runtime overhead. During a program's runtime, values of type `AccountDebit` will be simply translated into `Double`s. For the developer, there is an overhead involved in making `opaque type` usable. The translation between the opaque type and its value is done interchangeably only within the scope of the opaque type definition. Everywhere else, they are completely unrelated from the compiler's perspective. Let us now create type aliases that would allow for definition in the code example above.

```
opaque type AccountId = String

opaque type AccountDebit = Double
```

```
object AccountDebit:
  def apply(d: Double): Option[AccountDebit] =
    if (d < 0.0d) None else Some(d)
  extension (ad: AccountDebit)
    def -(ac: AccountCredit): AccountBalance =
      AccountBalance(ad - ac)
    def +(ta: TransferAmount): AccountDebit =
      AccountDebit(ad + ta).get

opaque type AccountCredit = Double
object AccountCredit:
  def apply(d: Double): Option[AccountCredit] =
    if (d < 0.0d) None else Some(d)
  extension (ac: AccountCredit)
    def +(ta: TransferAmount): AccountCredit =
      AccountCredit(ac + ta).get

opaque type TransferAmount = Double
object TransferAmount:
  def apply(d: Double): Option[TransferAmount] =
    if (d < 0.0d) None else Some(d)

opaque type AccountBalance = Double
object AccountBalance:
  def apply(d: Double): AccountBalance = d
```

As one can see, there is a clear overhead for defining `opaque type`, but even in this toy example we can see that we have almost type-level safety for basic business logic. It is also worth noting that, then with constructors defined as above, we are effectively limiting the type values, i.e. none of the `AccountDebit`, `AccountCredit` and `TransferAmount` can be a negative number. This is clearly a more *realistic* representation of the domain.

After this short detour, which motivated usage of opaque types, we can present it in the case of Self-Configurator. We decided to implement the following type aliases.

```
opaque type ConfigElementId = String
opaque type LabelKey = String
opaque type LabelValue = String
opaque type LabelMap = Map[LabelKey, LabelValue]
opaque type FunctionalityWeight = Double
opaque type ReactionId = String
```

All of those aliases will be used in the next sections of this chapter. Regarding the limitations of those types, no `String`-based type can be blank and `FunctionalityWeight` must be a positive number.

`cats.data.NonEmptySet[T]` represents a set that is not empty; all instances of that class will have at least one element of type `T`. Cats' `No⌋ nEmptySet` is not built on Scala's `Set` as `Set` relies on *universal equality* (`hashcode` and `equals` methods) to verify the uniqueness of its elements. The underlying data structure of `NonEmptySet` is `SortedSet` This is because it requires a `scala.math.Ordering` instance, which makes it more type-safe and aligned with the design principles followed in Cats.

Self-Configurator leverages `NonEmptySet` especially in Finite State Machine, which will be described later in Subsection 2.3.7.

`cats.data.Validated` datatype represents a computation that may fail. It is different from Scala's `Try` or `Either`, because it *accumulates* errors that might occur. Let us go through a simple example to better understand this datatype. Consider a simple web form with login and password fields. At the very least we expect both fields to be non-blank. *Without* accumulating structure if user would send both fields blank, the validation would stop at the first blank field, *with* accumulation both fields are validated, and the user receives full information about the errors they have made.

Within Self-Configurator, we ensure that `opaque types` mentioned before are properly initialized, for example, that `String` types are not blank and `FunctionalityWeight` is a positive number. See below the validation code.

```
type ValidationResult[A] =
  ValidatedNel[SelfConfigDomainValidation, A]

object SelfConfigDomainValidators:
  def nonBlankString[T <: String](
    parameterName: String
  )(value: T): ValidationResult[T] =
    if (value.isBlank)
      StringParameterCannotBeEmpty(parameterName).invalidNel
    else value.validNel

  def positiveDouble[T <: Double](
    parameterName: String
  )(value: T): ValidationResult[T] =
    if (value <= 0)
      DoubleParameterHasToBePositive(
        parameterName
      ).invalidNel
    else value.validNel
```

Now, let us go through the code above. `ValidatedNel` is a non-opaque type alias for `cats.data.Validated[cats.data.NonEmptyList[SelfConfigDo⌋`

mainValidation], A] and is provided by Cats library. SelfConfigDomainValidation is a defined trait that represents any potential domain error. The .invalidNel and .validNel are convenience extension methods provided by Cats, which allow easy mapping to the ValidationResult[A] type. Now, let us see an example usage of the validator for ConfigElementId.

```
opaque type ConfigElementId = String
object ConfigElementId:
  def apply(
    value: String
  ): ValidationResult[ConfigElementId] =
    SelfConfigDomainValidators.nonBlankString(
      "ConfigElementId"
    )(value)
```

Summary. One of the guiding principles we defined in Subsection 2.2.1 talked about strong typing and, more broadly, about leveraging compiler. One of the ways of doing that is to describe the domain in types that actually represent "reality". The tools described above are aligned with that principle. We have types for non-blank strings that are not interchangeable, hence it is, for example, difficult to confuse LabelKey with LabelValue. We assume that every RequirementsModel has a positive FunctionalityWeight associated with it. That approach ensures that domain model is type-safe very often down to the method's argument level. That makes reasoning about the problem easier, because we have precisely defined entities we work with. The obvious critique of this approach boils down to "is it worth the hassle?". In [14] the authors show that the use of a statically typed language reduces the bug count by 15% compared to a dynamically typed language. Another article [15] shows that static typing reduces the time required for software maintenance tasks. Different type of criticism may refer to the so-called "code base pollution", which especially concerns generic validation types, for example ValidationResult. This means that many methods in the code base will need to deal not directly with a particular type, but with its "wrapped" form, as in ValidationResult[FunctionalityWeight] instead of simply FunctionalityWeight. In our experience, this is usually a "code smell" or a technical debt. Properly designed and implemented code bases are validated at the edges. For example, if a service is receiving an HTTP request, it is validated within the HTTP handling layer (or as close to it as possible). Then the core business logic handling component deals with validated types.

2.3.2 Resources and connectors

Resource class directly corresponds to a resource mentioned in Section 2.1: Clarifying implementation objective. It *cannot* have any dependencies.

```
case class Resource(id: ConfigElementId, labels: LabelMap)
```

The id field is a unique identifier for a particular Resource, labels is a map (dictionary) of a String key to a String value. It allows for a more granular description of an entity. For example, we might have Map("type" -> "camera", "area" -> "living_room") to describe a camera resource that monitors the living room.

An important and fundamental observation is that there is a variety of potential Resources in the world. The question arises of how to connect them to Self-Configurator. Here, *connectors* come into play. The Self-Configurator exposes a Kafka topic for Resources to communicate by. The question of *how* particular Resource will send messages to that topic is abstracted from the Self-Configurator into a connector. It is Resource responsibility to be able to do that. The obvious drawback of this approach is that it requires more work from the user. The benefit, on the other hand, is that self-configuration becomes more independent and reusable. We can easily imagine a repository of connectors that can be shared among users.

There are three types of messages that the connector needs to be able to produce. All of them implement the ResourceListenerMessage trait.

```
sealed trait ResourceListenerMessage:
  val messageType: String

case class RegisterResource(resource: Resource)
    extends ResourceListenerMessage:
  val messageType: String = "RegisterResource"

case class DeregisterResource(resource: Resource)
    extends ResourceListenerMessage:
  val messageType: String = "DeregisterResource"

case class CustomMessage(content: String)
    extends ResourceListenerMessage:
  val messageType: String = "CustomMessage"
```

The field messageType is used in the serialization of JSON. The role of RegisterResource and DeregisterResource is self-explanatory. CustomMessage can be used to trigger reactions (see Subsection 2.3.4).

Connectors may or may not also implement handling SendSimpleKafkaMessage handling (more about this structure in Subsection 2.3.4). Handling that message allows for bidirectional communication between the Self-Configurator and Resource.

2.3.3 Requirements models

RequirementsModel is a class that represents Self-Configurator view of a requirement.

```
case class RequirementsModel(
  id: ConfigElementId,
  labels: LabelMap,
  requirements: Set[FunctionalityRequirement],
  weight: FunctionalityWeight
)
```

RequirementsModel as a bottom node – analogous to Resource – would be realized by setting the requirements field to an empty set. This implies that this functionality would automatically be available. This allows for the creation of more expressive configurations. For example, we can imagine an external cloud service that we assume to be always up. It is not a resource and does not depend on any resource.

When the requirements field is nonempty, RequirementsModel requires other RequirementsModels or Resources. There are two types of FunctionalityRequirements.

```
sealed trait FunctionalityRequirement:
  val exclusive: Boolean

case class IdBasedRequirement(
  id: ConfigElementId,
  exclusive: Boolean = false
) extends FunctionalityRequirement

case class LabelBasedRequirement(
  labelKey: LabelKey,
  labelValue: LabelValue,
  count: Int,
  exclusive: Boolean = false
) extends FunctionalityRequirement
```

IdBasedRequirement specifies that either RequirementsModel or Resource with *specific* id is necessary for a functionality (represented by RequirementsModel) to work properly. LabelBasedRequirement specifies how many elements with particular labels are required; exclusive field states whether an entity can be shared with other RequirementsModel. If it is set to true, then only one RequirementsModel will be able to use it. It will be excluded from further requirements checks. For example, we might share a camera between requirements, whereas a shuttle bus that takes people from point A to point B would probably not be shared. RequirementsModel's field weight determines how important the functionality is in the scope of the IoT deployment.

Requirements check. A rather straightforward Algorithm 1 (see page 17) is responsible for checking whether all `RequirementsModel` are met.

2.3.4 Self-Configurator reactions

One of the core parts of Self-Configurator is its ability to react to actions. Reactions are defined by the system administrator. The reaction class is defined as follows.

```
case class ReactionModel(
  reactionId: ReactionId,
  filterExpression: FilterExpression,
  action: ReactionAction
)
```

This simple structure consists of three elements – an identifier, a filter expression that dictates *when* the reaction should be triggered, and reaction action (do not confuse with Actions from previous section) that tells *what* should happen.

First, let us focus on the filter expressions.

```
sealed trait FilterExpression

case class ResourceIsAvailable(id: ConfigElementId)
    extends FilterExpression

case class ResourceIsNoLongerAvailable(id: ConfigElementId)
    extends FilterExpression

case class ResourceWithLabelIsAvailable(
  labelKey: LabelKey,
  labelValue: LabelValue
) extends FilterExpression

case class ResourceWithLabelIsNoLongerAvailable(
  labelKey: LabelKey,
  labelValue: LabelValue
) extends FilterExpression

case class CustomMessageContent(content: String)
    extends FilterExpression

case object AnyEvent extends FilterExpression
```

Most of the `FilterExpressions` are self-explanatory, so we will focus only on `CustomMessageContent`. This expression is connected to a `Custo⌋ mMessage` from Subsection 2.3.2. The expression will be evaluated to `true`

Algorithm 1 Requirement check algorithm

```
Input:
  RMs – Non empty set of RequirementsModels
  Rs – Set of Resources
Output:
  True if all RequirementsModel in RMs are met, false otherwise.

  function CheckRecursive($head$, $tail$, $avIds$, $avLabs$)
      $reqIds \leftarrow$ requirements of class IdBasedRequirement of $head$
      $reqLabels \leftarrow$ requirementss of class LabelBasedRequirement of $head$
      if $reqIds$ is subset of $avIds$ then
          $avIds \leftarrow (avIds - reqIds)$ marked as exclusive
      else
          return false
      end if
      if $reqLabs$ is subset of $avLabs$ then
          $avLabs \leftarrow (avLabs - reqLabs)$ marked as exclusive
      else
          return false
      end if
      if $tail$ is empty then
          return true
      else
          $avIds \leftarrow$ add $head$ id
          $avLabs \leftarrow$ add $head$ labels
          $nextHead \leftarrow$ first element of $tail$
          $nextTail \leftarrow$ all but first element of $tail$
          return CheckRecursive($nextHead$, $nextTail$, $avIds$, $avLabs$)
      end if
  end function

  $sortedRMs \leftarrow$ inverse topological sort of $RMs$
  if $sortedRMs$ is empty then
      return true
  else
      $avIds \leftarrow$ all ids from $Rs$
      $avLabs \leftarrow$ All add labels (with counts) from $Rs$
      $head \leftarrow$ first element of $RMs$
      $tail \leftarrow$ all but first element of $RMs$
      return CheckRecursive($head$, $tail$, $avIds$, $avLabs$)
  end if
```

if and only if the content of `CustomMessage` is equal to the `CustomMessageContent`'s `content` field. It should be noted that all these expressions only refer to actions that occur through the "resource" Kafka topic. Excluding `CustomMessageContent`, they are triggered by a resource (or its connector). `CustomMessageContent` might be either triggered by a resource or by an external actor having access to a "resource" Kafka topic.

There are six reactions that an administrator can define: `SendSimpleKafkaMessage`, `ReplaceConfiguration`, `UpsertConfiguration`, `ConditionalAction`, `KeepHighestWeightFunctionalities` and `NoAction`.

`SendSimpleKafkaMessage` sends a specific message on a specific topic. This is used to send, for example, configuration messages to a particular connector. The message is extended with information about what triggered it. For example, the message below was triggered by adding a resource with id `"resource_id_1"`.

```
{
  "trigger" : {
    "event" : "ResourceAdded",
    "content" : {
      "id" : "resource_id_1",
      "labels" : {
        "label_key" : "label_value"
      }
    }
  },
  "content" : "This is just a test"
}
```

The difference between `ReplaceConfiguration` and `UpsertConfiguration` is simple. The former replaces the entire set of `RequirementsModel`, while the latter can insert new or update existing `RequirementsModel`s.

`ConditionalAction` lets the user perform a reaction only if the current configuration and the incoming action meet certain specific criteria.

```
case class ConditionalAction(
  conditionalCheck: Condition,
  action: ReactionAction,
  fallback: ReactionAction
) extends ReactionAction
```

The construction of the class is simple; if `conditionalCheck` is evaluated to `true`, then `action` is executed; otherwise, `fallback` is run. The `Condition` class was implemented to express a variety of Boolean conditions: `AndCondition`, `OrCondition`, `NotCondition`, `ContainsRequirements`, `ContainsRequirementWithId`, `ContainsRequirementsWithLabels`, `ContainsRequire` mentsWithWeight, ContainsRequirementsWithRequirements, ContainsResources, ContainsResourceWithId, ContainsResourceWithLabels, MessageContainsResource, and MessageContainsResourceWithEvent. The broad selection of Condition with the fact that the administrator can nest ConditionalAction makes it possible for an administrator to define behavior even for more complex scenarios.

KeepHighestWeightFunctionalities retains only RequirementsModel that the current set of available Resources can support. It is a one-off action, meaning that after executing this reaction, the state of Self-Configurator will be changed. An algorithm is presented below which solves the problem of an autonomous decision on how to select which RequirementsModels Self-Configurator should keep on a limited budget. To state it more explicitly, given a set of Resources, return a set of RequirementsModel that has a maximum sum of `weight`. Algorithm 2 presents a naive solution.

Algorithm 2 Get the requirements met with highest weight

```
Input:
  RMs – Non empty set of RequirementsModels
  Rs – Set of Resources
Output:
  Set of RequirementsModels

  $powerSetRMs$ ← powerset of $RMs$                        ▷ Without empty set
  $sortedPowerSetRMs$ ← sort $powerSetRMs$ by it's weight sum
  for subset in sortedPowerSetRMs do
      if AllRequirementsMet(subset, Rs) then          ▷ Call algorithm 1
          return subset
      end if
  end for
  return $\phi$
```

2.3.5 Self-Configurator as EventSourcedBehavior

Before we can freely talk about Self-Configurator's actions, reactions, and its finite state machine, we need to understand Akka's EventSourcedBehavior [16].

First, let us focus on the *event sourced* part. In [17] events have the following key characteristics: they occur in the past; they are immutable; they are one-way messages; events have a single source, but can have one or more recipients; they include metadata; they should describe "business intent". The basic idea of event sourcing is to ensure that each change to an application's state is captured in an event object, and that these event objects themselves are stored in the sequence in which they were applied, instead of storing the state directly. Event sourcing persists the state of an entity as a sequence of events that alter the state.

Now we can switch to Akka's support for that concept. Akka persistence allows stateful actors (which are called persistent or event sourced) to preserve their state, making it possible to recover their state when they are restarted. The key concept behind Akka Persistence is that state changing events are persisted, but not the state itself. A persistent actor is recovered by replaying the stored events to the actor, effectively rebuilding its state. When a persistent actor receives a (non-persistent) command, the command is validated if it can be applied to the current state. Validation can be as complex as required, ranging from checking message type or content to communicating with an external validation service. Only after validation succeeds are events generated on the basis of the command. These events are then persisted, and then the actor's state is changed according to the event. Commands that do not generate events might be used to query the state. At the code level, this is represented by the `EventSourcedBehavior` trait.

```
// parts of code omitted
trait EventSourcedBehavior[Command, Event, State]

object EventSourcedBehavior:
  def apply[Command, Event, State](
    persistenceId: PersistenceId,
    emptyState: State,
    commandHandler: (State, Command) => Effect[Event, State],
    eventHandler: (State, Event) => State
  ): EventSourcedBehavior[Command, Event, State] = ...
```

The `persistenceId` is a unique identifier for the actor. `emptyState` describes initial (empty) state of a persistent actor. `commandHandler` inspects incoming `Command` having current `State` at disposal. It can then, via `Effect`, persist particular `Event`, run an additional code, or send a reply. `eventHa⌋ndler` describes what a resulting state is based on the incoming `Event` and current `State`. The result of `commandHandler` (if there is persist-type `Event`) is passed to `eventHandler`.

2.3.6 Self-Configurator actions

Actions are *only way* to communicate with the Self-Configurator Actor, regardless of whether the request comes from the system administrator, a resource, or if it is a custom Kafka message. The actions in this section are mapped to the `Command` type parameter in `EventSourcedBehavior` [16]. The following actions are available: `AddResource`, `RemoveResource`, `AddFuncti⌋onalityModel`, `RemoveFunctionalityModel`, `AddReactionModel`, `Remove⌋`

ReactionModel, ReplaceRequirements, UpsertRequirements, and CustomMessage. All of them implement the following trait.

```
sealed trait SystemConfigCommand:
  def replyTo: ActorRef[SystemConfigResponse]
```

Most of those actions are self-explanatory, but the last three require additional context. ReplaceRequirements simply replace all existing requirements with those passed through this action's parameter. UpsertRequirements will either replace or insert requirements based on RequirementsModel id. This action has a special removeDangling flag that will remove RequirementsModels that would be orphaned; the parent-child relationship is formed via RequirementsModel's requirements field. Let us say that we have a *smart home* requirements model that depends on *security module*, which in turn depends on *cloud security provider*. If we update the *smart home* requirement model with exclusive dependency on *cloud security provider* with removeDangling enabled, then as a "side effect" *security module* would be removed from the set of requirements model. CustomMessage is a custom message that comes through the Kafka topic. It can be used to trigger reactions.

2.3.7 Self-Configurator Finite State Machine

Core part of Self-Configurator is a Akka's EventSourcedBehavior that is effectively a Finite State Machine (FSM). The object containing this entity is called SystemConfigBehavior. The FSM of the Self-Configurator can store three different sets: of Resources, of RequirementsModels, and of ReactionModel. From this it follows that there are 8 (2^3) different states. The basic trait is named SystemConfigState and the following concrete states are available: EmptyState, StateWithRequirements, StateWithResources, StateWithReactions, StateWithResourcesAndRequirements, StateWithResourcesAndReactions, StateWithRequirementsAndReactions, StateWithResourcesRequirementsAndReactions

There is only one type of validation that this actor performs, that is, to ensure that RequirementModels always form a directed acyclic graph. This is done using the Algorithm 1.

The graph below shows all possible states. Moving between states is to perform one of the *Actions* in the previous section.

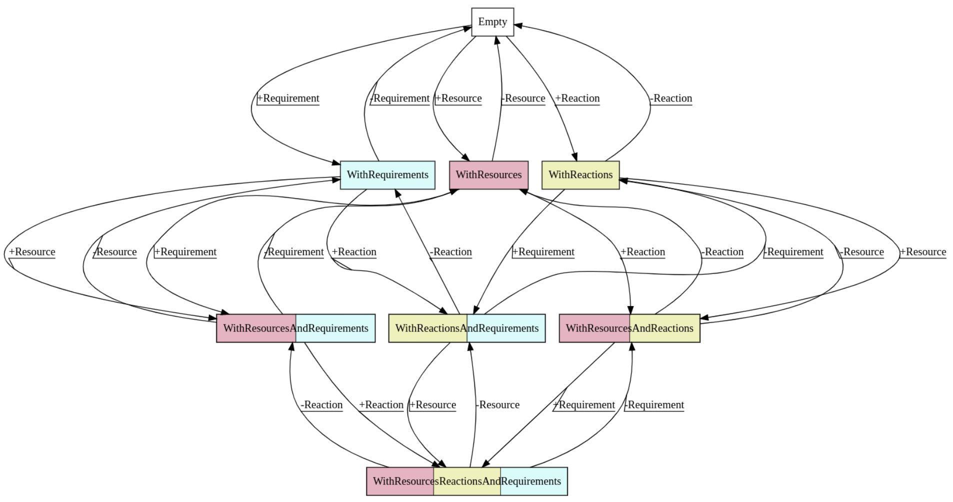

Figure 2.2: Simplified Self-Configurator state transition graph. The + and - refer to either the addition or removal action of a particular entity. Self-edges were omitted – for example, if the Self-Configurator is in the WithRequirements state and *+Requirements* is executed, then the Self-Configurator remains in the WithRequirements state. Only the set of requirements is getting updated.

2.3.8 Communication interfaces

There are two communication channels, HTTP and Kafka.

HTTP API is used to add or remove `RequirementsModel`s and `ReactionModel`s. When deploying Self-Configurator, those endpoints should be secured so that only the deployment administrator can access them. `RequirementsModel` can be created at `POST /requirements-model`, deleted at `DELETE /requirements-model/id`. `ReactionModel` have analogous paths at `POST /reaction-model` and `DELETE /reaction-model/id`.

As for Kafka, there are two necessary topics – one for resources and one for notifying whether registered resources meet *all* defined requirements. Requests coming from either channel are converted to actions, then validated and handled by Self-Configurator.

2.3.9 Combining components together

Now to bring all previously defined components together and let us go step by step through the following requests.

1. Set a simple requirements model.

2. Send another requirements model that will not be accepted because it breaks DAG invariant.

3. Define a simple reaction model
4. Register a resource and trigger reaction.

Set a simple requirements model. This will start with an HTTP request `POST /requirements-model`

```
{
  "id" : "smart_home_functionality",
  "labels" : { },
  "requirements" : [
    {
      "id" : "wide_view_cam",
      "exclusive" : false
    },
    {
      "id" : "thermostat_functionality",
      "exclusive" : false
    }
  ],
  "weight" : 2.0
}
```

This request will be deserialized as `RequirementsModel`. Then this model will be wrapped in `AddReactionModel` and sent to `SystemConfigBehavior`. Within this behavior, the following part of the command handler will be triggered.

```
Effect
  .persist(RequirementsModelAdded(requirementsModel))
  .thenRun(requirementsNotification(requirementsMetNotifier))
  .thenReply(replyTo)(_ => SystemConfigAck(msg))
```

First, the `RequirementsModelAdded` event will be persisted. Persisting of the event will result in a change in the state of `SystemConfigBehavior` from `EmptyState` to `WithRequirements`. The message will then be published on the “requirements met” Kafka topic, and from now on the message will state that the requirements are not met. Finally, the acknowledging message will be returned to the original sender.

Send another requirements model. Now let us consider another requirement model that will sent through `POST /requirements-model` endpoint.

```
{
  "id" : "thermostat_functionality",
    "labels" : { },
    "requirements" : [
      {
        "id" : "smart_home_functionality",
        "exclusive" : false
      }
    ],
    "weight" : 2.0
  }
```

This request will follow the same path as the one in the previous paragraph, but during validation in SystemConfigBehavior, it will be rejected, because there is a circular dependency between "smart_home_functionality" and "thermostat_functionality". Due to this, the state will not be updated, so it will remain in the previous WithRequirements state. Also, no notification will be published to “requirements met” Kafka topic.

Define a simple reaction model. Now, we will send a request to another endpoint `POST /reaction-model`.

```
  {
    "reactionId" : "reaction_1",
    "filterExpression" : {
      "messageType" : "ResourceIsAvailable",
      "id" : "wide_view_cam"
    },
    "action" : {
      "requirements" : [
        {
          "id" : "smart_home_functionality",
          "labels" : { },
          "requirements" : [
            {
              "id" : "wide_view_cam",
              "exclusive" : false
            }
          ],
          "weight" : 2.0
        }
      ]
    }
  }
```

This reaction will be triggered when a Resource with id "wide_view_cam" becomes available. The reaction will replace the existing set of Requireme⌋ ntsModel with those defined in the "requirements" field. Effectively, it will simplify the existing "smart_home_functionality" to require only Resource or RequirementsModel with "wide_view_cam" id. After deserializing from JSON into ReactionModel, it will be passed to SystemConfigBehavior via the AddReactionModel command.

```
Effect
  .persist(ReactionModelAdded(reactionModel))
  .thenReply(replyTo)(_ => SystemConfigAck(msg))
```

After persisting ReactionModelAdded event, the state will change from StateWithRequirements to StateWithRequirementsAndReactions.

Register a resource and trigger reaction. The last step will be to trace how the connector would communicate with Self-Configurator. A connector publishes the following message on “resource” Kafka topic.

```
{
  "messageType" : "RegisterResource",
  "resource" : {
    "id" : "wide_view_cam",
    "labels" : {
      "device_type" : "camera"
    }
  }
}
```

This will be converted into the RegisterResource message and then passed to SystemConfigBehavior via AddResource. The following things will happen. First, the ResourceAdded event will be persisted, which will effectively change the state from StateWithRequirementsAndReactions to StateWithResourcesRequirementsAndReactions. Then the reaction will be triggered, which was defined in the previous step. However, due to Akka’s nature, before that happens, the message will be sent to “requirements met” topic notifying the user that not all requirements are met. Only after that will the requirements be replaced, and then the notification will be sent that the requirements are actually met.

Chapter 3

Example: self-configuration in port environment

3.1 Motivation

The global market for port infrastructure is valued at USD 148.1 billion in 2020 and is expected to reach USD 243.1 billion by 2030. [18]. Port infrastructure is the basis for port operations used to service ships, cargo, and passengers. Ports play an important role in the international supply chain. Disruptions in port operations are often caused by technical issues with the devices used in ports, such as cranes, rubber tire gantries (RTGs), and automated guided vehicles (AGVs). The tight schedule of the global logistics chain cannot be maintained unless all container handling equipment is running efficiently and without malfunctions. Port operators now have the opportunity to use IoT-enabled monitoring systems equipped with accurate sensor technology to detect performance problems. Those systems have the ability to minimize the significant risk of unplanned downtime. IoT sensors collect data from various port systems and send them to software capable of analyzing such issues.

In this chapter, we will devise sample scenarios in the port environment for the Self-Configurator. Then, we will describe software-level constructs that will be required to realize them. Finally, we will go through each scenario step by step.

3.2 Defining port scenarios

To showcase the application of Self-Configurator, we decided to devise three port-based scenarios. Each of them is elementary, but they show the basic principles on which Self-Configurator operates. Three scenarios are: multi-step configuration, maintaining important traffic, and rescue mode.

3.2.1 Scenarios description

In a multistep configuration, a set of AGVs connect to the self-configured system, and before being available and configured, they need to: download the latest port map and be assigned to a particular port region.

The second scenario, maintaining important traffic, is defined as follows; we have a set of RTG and AGV groups that handle cargo with various priorities. Due to system malfunction, only a limited set of AVGs are available; hence we need to prioritize high-impact cargo.

The last scenario, the rescue mode, might occur in case of large-scale fire in the port. It will stop all existing actions and send all resources to a safe parking location, but only if a special parking zone dispatcher zone is available. Otherwise, we will update the configuration to be safe during fire.

3.2.2 Resources

Given the descriptions above, we can identify two types of resources – RTG and AVG. Each of them could be identified by a label. Each of them clearly requires a unique id. Let us now turn to the code-level description.

```
val sampleRTG = Resource(
  "rtg_x",
  LabelMap.of("resource_type" -> "rtg")
)
val sampleAGV = Resource(
  "agv_x",
  LabelMap.of(
    "resource_type" -> "agv",
    "configuration_step" -> "not_configured"
  )
)
```

The ids are suffixed with `"_x"`. In reality, this id should be unique in the scope of deployment. The `"configuration_step"` label will be used in the multistep configuration scenario.

Connectors

We can assume that all those resources are computationally powerful enough to handle the connector code internally. For the sake of scenarios, all connectors can handle bidirectional (as described in Subsection 2.3.2: Resources and connectors) Kafka communication. Each connector subscribes to a topic with a name equal to their `"resource_type"`. For example, AGV will subscribe to a topic named `"agv"`.

3.2.3 Requirements

Interestingly enough, the scenarios described above are not `RequirementsModel` heavy. For multistep configuration and rescue mode, the models are omitted for clarity of presentation. To maintain important traffic, the following `RequirementsModel`s can be introduced. They simply represent a requirement for a specific number of AGVs that will be *exclusively* used within that requirement.

```
val highPriorityRM = RequirementsModel(
  "high_priority_cargo_handling",
  LabelMap.empty,
  Set.of(
    IdBasedRequirement("rtg_1", exclusive = true),
    LabelBasedRequirement(
      "resource_type", "agv", 5, exclusive = true
    )
  ),
  FunctionalityWeight(10)
)

val mediumPriorityRM = RequirementsModel(
  "medium_priority_cargo_handling",
  LabelMap.empty,
  Set.of(
    IdBasedRequirement("rtg_2", exclusive = true),
    LabelBasedRequirement(
      "resource_type", "agv", 4, exclusive = true
    )
  ),
  FunctionalityWeight(7)
)

val lowPriorityRM = RequirementsModel(
  "low_priority_cargo_handling",
  LabelMap.empty,
  Set.of(
    IdBasedRequirement("rtg_3", exclusive = true),
    LabelBasedRequirement(
      "resource_type", "agv", 3, exclusive = true
    )
  ),
  FunctionalityWeight(5)
)
```

3.2.4 Reactions

Contrary to the limited number of `RequirementsModel`s, more `ReactionModel`s are required to handle the defined scenarios.

For the multistep configuration, we require a reaction for each step. To express this in the language of Self-Configurator, we need to:

1. React to a resource with label `"configuration_step" -> "not_configured"` by sending a link to a Kafka topic.
2. Wait for confirmation that the map has been downloaded and updated. Mark the resource with label `"configuration_step" -> "map_downloaded"`
3. After resource has downloaded the map, send a message asking *zone assigner service* to assign a specific zone.

```
val msConfigurationStep1 = ReactionModel(
  "multi_step_configuration_1",
  ResourceWithLabelIsAvailable(
    "configuration_step", "not_configured"
  ),
  SendSimpleKafkaMessage(
    "agv", "map_download_link: http://proper_address.com"
  )
)

val msConfigurationStep1 = ReactionModel(
  "multi_step_configuration_2",
  ResourceWithLabelIsAvailable(
    "configuration_step", "map_downloaded"
  ),
  SendSimpleKafkaMessage(
    "agv", "ask_assigner_for_zone. assigner_location: ABC"
  )
)
```

For the scenario of maintaining critical traffic, we only need one `ReactionModel`. It will be triggered by a custom message with `"critical_event"` as its content. The reaction will trigger the Algorithm 2.

```
val keepCriticalTraffic = ReactionModel(
  "keep_critical_traffic_running",
  CustomMessageContent("critical_event"),
  KeepHighestWeightFunctionalities
)
```

The fire hazard scenario will also require only one ReactionModel. It will be triggered on any incoming action. The interesting part is that the reaction is conditional – "go_to_parking" message will be sent only if there is a "parking_zone_dispatcher" resource registered with Self-Configurator. Otherwise, the configuration will be replaced.

```
val goToParking = ReactionModel(
  "go_to_parking",
  CustomMessageContent("fire"),
  ConditionalAction(
    ContainsResourceWithId("parking_zone_dispatcher"),
    SendSimpleKafkaMessage("resource", "go_to_parking"),
    ReplaceConfiguration(
      /*configuration for stopping all operations*/
    )
  )
)
```

3.3 Port scenarios walk-through

In the sections below, a step-by-step progression is presented for each scenario. Resources are presented as circles, requirements as rounded rectangles, Kafka topic as a horizontal cylinder (although it may be omitted and only a message will be presented in a "callout" element), and the Self-Configurator itself is a square aptly named "Self-Configurator".

3.3.1 Multi-step configuration

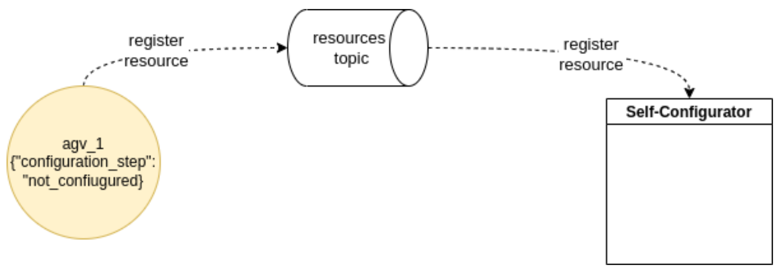

Figure 3.1: Multistep configuration: AGV registers in Self-Configurator

AGV sends a RegisterResource message to a "resources " Kafka topic. Then, the Self-Configurator consumes the message from that topic and transforms it into the AddResource command for SystemConfigBehavior. After event is persisted, the state of SystemConfigBehavior is updated. Finally, a reaction is triggered that sends a Kafka message.

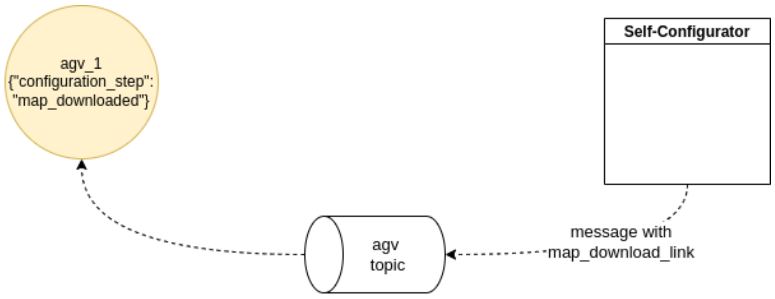

Figure 3.2: Multistep configuration: Self-Configurator sends a message to AGV

Self-Configurator sends a Kafka message to the “agv” Kafka topic, from which the AGV can consume. The message itself will contain a metadata-like field with information about what triggered the message, in this situation it would be `"agv_1"` registering in Self-Configurator. This will allow agv subscribers to filter messages that are of interest to them. The message would have the form presented below.

```
{
  "trigger" : {
    "event" : "ResourceAdded",
    "content" : {
      "id" : "agv_1",
      "labels" : {
        "configuration_step" : "not_configured"
      }
    }
  },
  "content" : "map_download_link"
}
```

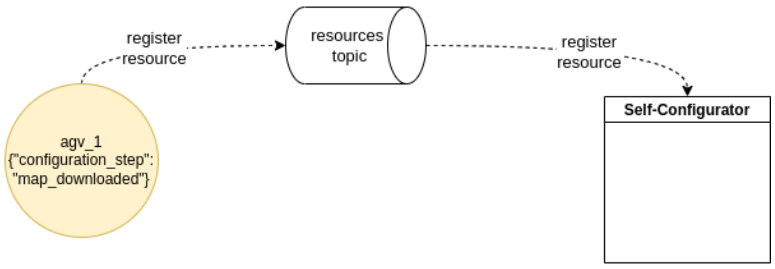

Figure 3.3: Multistep configuration: AGV registers again to update its state in Self-Configurator

After downloading the map, AGV must once again register with Self-Configurator, this time with an updated `"configuration_step"` label. In

Self-Configurator, the "agv_1" resource representation will be updated, because the same resource id was used.

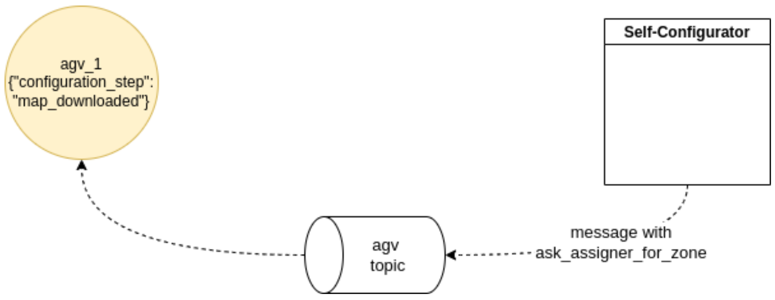

Figure 3.4: Multistep configuration: Self-Configurator sends another message

Now, the Self-Configurator sends another message with details about the zone assigner. Based on that, AGV will once again update its label and register again with Self-Configurator. Having reactions with various triggers as well as conditional reactions can create much more complex flows.

3.3.2 Maintaining important traffic

To improve readability in the graphs below, the Self-Configurator component was omitted, so that we can explicitly focus on requirements and resources.

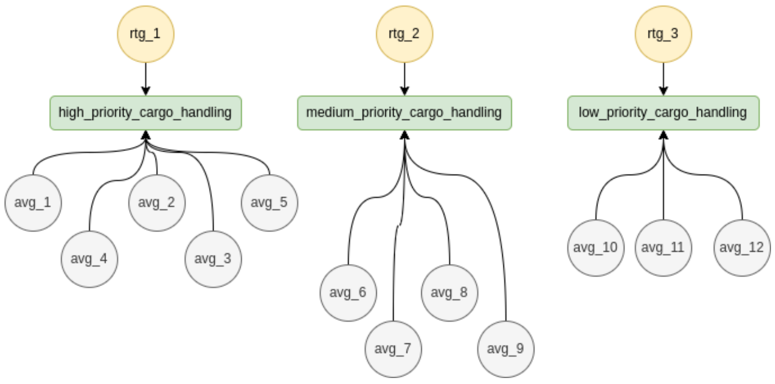
rtg_1
rtg_2
rtg_3
high_priority_cargo_handling
medium_priority_cargo_handling
low_priority_cargo_handling
avg_1
avg_2
avg_5
avg_4
avg_3
avg_6
avg_8
avg_7
avg_9
avg_10
avg_11
avg_12

Figure 3.5: Keeping important traffic: initial state

At the very beginning of this scenario, we have a situation in which all requirements are met. There are in total 12 AGVs spread over three requirements that represent the handling of cargo of varying priority.

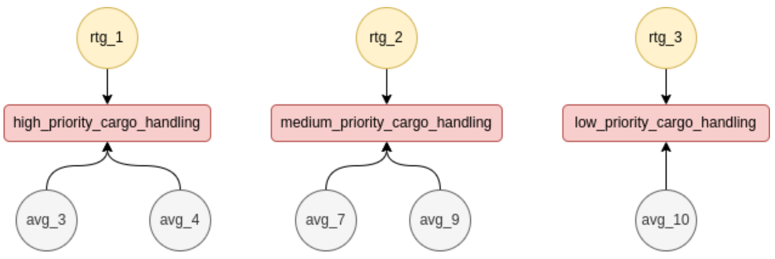
rtg_1
rtg_2
rtg_3
high_priority_cargo_handling
medium_priority_cargo_handling
low_priority_cargo_handling
avg_3
avg_4
avg_7
avg_9
avg_10

Figure 3.6: Keeping important traffic: resources went down

Due to some external circumstances (for example, faulty components), only five of the initial 12 AGVs are available. This means that in the current configuration, not a single requirement is met. Effectively, a cargo of no priority is being handled properly.

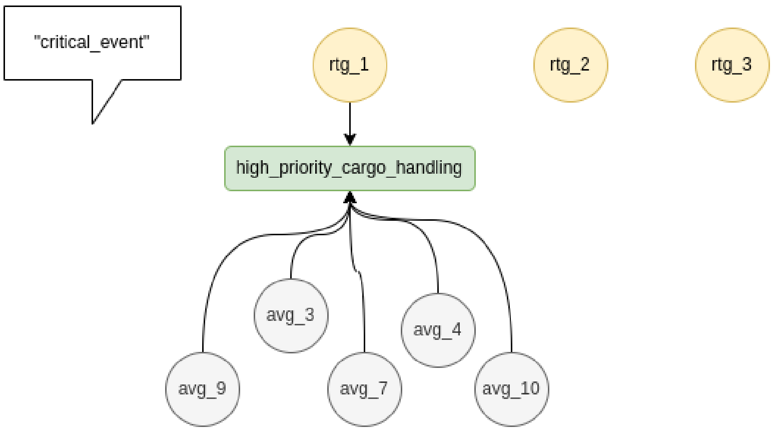

Figure 3.7: Keeping important traffic: requirements with 5 AGVs

A "critical_event" message was received on the "resource" Kafka topic. That triggered KeepHighestWeightFunctionalities reaction which used Algorithm 2. All requirements need AGVs, high-priority requires 5, medium 4, and low 3. By investigating the weights of each RequirementsModel, we notice that the highest weight we can achieve is 10 and that is achieved by assigning all AGVs to "high_priority_cargo_handling". Now let us consider what would happen if there were seven AGVs available and the same reaction would be triggered.

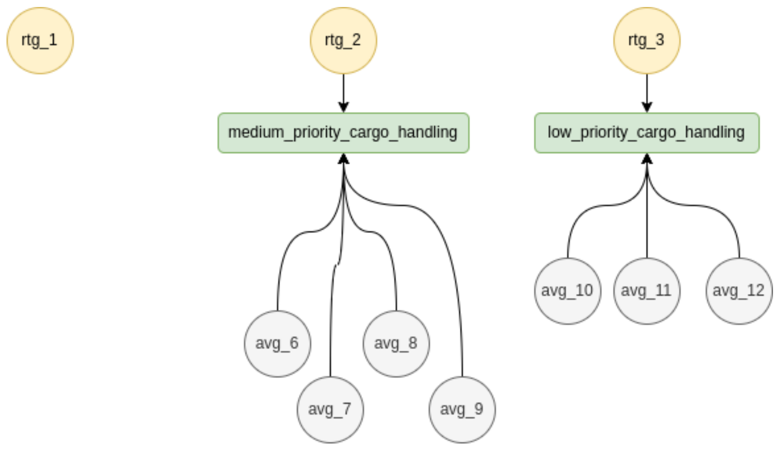

Figure 3.8: Keeping important traffic: requirements with 7 AGVs

Now the situation is different. With 7 AGVs, the highest sum is achieved by combining medium and low cargo handling requirements. When defining the requirements and then triggering `KeepHighestWeightFunctionalities` reaction, it is important to be aware that the Self-Configurator will try to achieve the maximum weight sum across all defined requirements.

3.3.3 Rescue mode

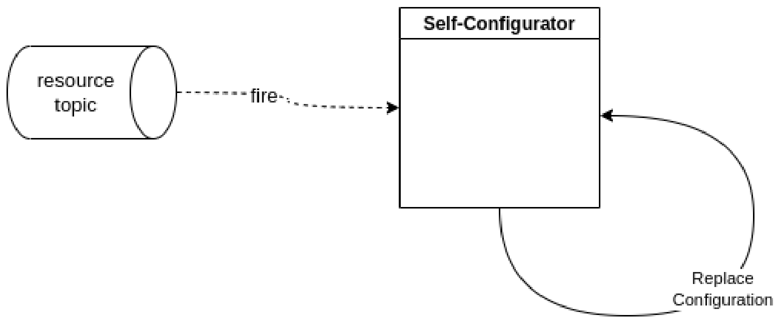

Figure 3.9: Rescue mode: without `"parking_zone_dispatcher"`

After receiving the `"fire"` message on “resource“ Kafka topic, when there is no `"parking_zone_dispatcher"` registered in the Self-Configurator, a fallback action will be executed and the Self-Configurator will replace its configuration with a predefined schema. `ReplaceRequirements` would be executed on `SystemConfigBehavior`.

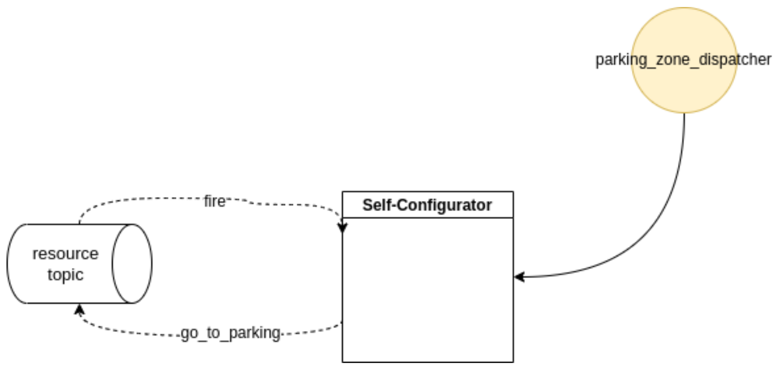

Figure 3.10: Rescue mode: with "parking_zone_dispatcher"

When there is "parking_zone_dispatcher" registered in Self-Configurator, a primary action will be executed. This means that a "go_to_parking" message will be sent on a "resource" topic.

Chapter 4

Summary

In the scope of this work, we sketched the current landscape of the Internet of Things and focused on one of its main challenges, the configuration of heterogeneous resources. We described autonomous systems based on the self-* approach proposed by IBM. In the Chapter 2, we clearly defined what it would mean for software to realize the self-configuration concept. We presented the guiding principles for selecting tools used in the thesis. Then, the core part of this work, software named Self-Configurator, was presented. Starting from the ground, we described elementary building blocks, which were then used to build more complex components. Later, we presented a simple example of how these components interact with each other. By introducing the concept of a resource and a connector, we were able to "abstract out" the problem of non-uniformity of APIs for heterogeneous resources. We provided a tool for describing the requirements of a system's functionality and ensured that adding new requirements will not leave the system in an "impossible to meet" state. The software can react to external events by executing complex, conditional actions constructed out of building blocks. Finally, its users are notified of every state change. The state itself is represented as an append-only log of events, utilizing event sourcing. Then, in Chapter 3, we used the Self-Configurator to showcase how it could be leveraged in a port environment in various scenarios – multistep configuration, maintaining important traffic, and handling dangerous situations, such as a fire breaking out. Software that we built met all the criteria we defined for self-configuration.

We believe that the presented Self-Configurator software can be already applied to both simple and complex IoT deployments. It also forms a good foundation for further work that could be leveraged both in academia and industry. In particular, the software we built in this thesis will (with some extensions and enhancements) be used in pilot deployments of the cutting-edge European research project ASSIST-IoT [19].

We built Self-Configurator using novel, type-safe tools in an extensible and easy-to-maintain way. During the course of the implementation, we deepened our experience with languages, libraries, and datastores used, collecting some observations, which we summarize below.

Scala 3 introduced a lot of improvements compared to Scala 2 – revamped syntax, redesigned implicits and macros, and an considerably improved type system. In principle, the Scala 2.13 code can be used in Scala 3, but the first challenge we faced was that Scala 3 does not support Scala 2.13 macros [20]. Combining that with the fact that Akka HTTP was not yet migrated to Scala 3, we faced the following problem. We could execute Akka HTTP code in Scala 3, but it broke ScalaTest dependencies (de-facto standard library for writing tests in Scala). This is because ScalaTest heavily leverages Scala 2.13 macros. Because of that, we were forced to switch to http4s as an HTTP library [21]. Currently, IntelliJ IDEA is the most popular IDE for working with Scala. It has great support for Scala 2.13 and greatly improves the developer's performance. However, in the authors development environment, Scala 3 (currently) had longer compile times than Scala 2.13, hence working with IntelliJ IDEA tools became frustrating. During development, we changed to using VS Code with the Metals – a language server for Scala [22]. The good part was the ease of writing a type-safe code. The opaque types and extension methods made writing code that uses the Scala 3 compiler a pleasant experience.

Cats library helped us with additional type-safety, especially by using its data structures like `"NonEmptySet"`. We used the Cats Effect to handle Kafka communication, mostly because of the lack of native Akka solutions for Kafka in Scala 3. We think that going more into the purely functional "side" of Scala 3 would give interesting results – we could completely omit Akka and try to leverage the Cats ecosystem even more. On the other hand, with Akka Persistence, we had FSM and Event Streaming support "out-of-the-box", which helped to shorten the development time. Nonetheless, in the coming months when the Akka ecosystem will gain full Scala 3 support, having all components within the umbrella of one ecosystem will further simplify the software. Considering those comments, we believe that we made the right decision about the selected tools. In conclusion, we faced a standard set of challenges when working with novel technologies. Slower compilation times, less complete tooling support, not as comprehensive documentation, and a smaller number of examples.

List of Algorithms

List of Figures

Bibliography

[1] ITU-T Study Group 20. *Overview of the Internet of things*. URL: https://handle.itu.int/11.1002/1000/11559. (accessed: 05.09.2022).

[2] Fortune Business Insights. *Internet of Things (IoT) Market Size, Share & COVID-19 Impact Analysis, By Component (Platform, Solution & Services), By End-use Industry (BFSI, Retail, Government, Healthcare, Manufacturing, Agriculture, Sustainable Energy, Transportation, IT & Telecom, and Others), and Regional Forecast, 2022-2029*. URL: https://www.fortunebusinessinsights.com/industry-reports/internet-of-things-iot-market-100307. (accessed: 05.09.2022).

[3] Ericsson. *Cellular networks for Massive IoT*. 2016.

[4] *An architectural blueprint for autonomic computing*. URL: https://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf. (accessed: 20.09.2022).

[5] Stefan Poslad. "Autonomous Systems and Artificial Life". In: *Ubiquitous Computing: Smart Devices, Environments and Interactions*. 2009, pp. 317–341. DOI: 10.1002/9780470779446.ch10.

[6] Mohammad Reza Nami and Koen Bertels. "A Survey of Autonomic Computing Systems". In: *Third International Conference on Autonomic and Autonomous Systems (ICAS'07)*. 2007, pp. 26–26. DOI: 10.1109/CONIELECOMP.2007.48.

[7] Kumar Nalinaksh, Piotr Lewandowski, Maria Ganzha, Marcin Paprzycki, Wiesław Pawłowski, and Katarzyna Wasielewska-Michniewska. "Implementing autonomic Internet of Things ecosystems – practical considerations". In: *Lecture Notes in Computer Science* (2021), pp. 420–433. DOI: 10.1007/978-3-030-86359-3_32.

[8] *Akka*. URL: https://akka.io/. (accessed: 14.09.2022).

[9] *Akka Persistence Documentation*. URL: https://doc.akka.io/docs/akka/current/typed/persistence.html. (accessed: 14.09.2022).

[10] *Cats*. URL: https://typelevel.org/cats/. (accessed: 14.09.2022).

[11] *Circe*. URL: https://circe.github.io/circe/. (accessed: 14.09.2022).

[12] *EventStoreDB V21.10 Documentation.* URL: https://developers.eventstore.com/server/v21.10/. (accessed: 14.09.2022).

[13] *Apache Kafka.* URL: https://kafka.apache.org/. (accessed: 14.09.2022).

[14] Prem Devanbu, Tom Zimmermann, and Christian Bird. "Belief & Evidence in Empirical Software Engineering". In: *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016).* Association for Computing Machinery, May 2016. URL: https://www.microsoft.com/en-us/research/publication/belief-evidence-in-empirical-software-engineering/.

[15] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. "An empirical study on the impact of static typing on software maintainability". In: *Empirical Software Engineering* 19.5 (2013), pp. 1335–1382. DOI: 10.1007/s10664-013-9289-1.

[16] *Documentation for EventSourcedBehavior.* URL: https://doc.akka.io/japi/akka/2.6/akka/persistence/typed/javadsl/EventSourcedBehavior.html. (accessed: 19.09.2022).

[17] Archiveddocs. *Reference 3: Introducing event sourcing.* URL: https://learn.microsoft.com/en-us/previous-versions/msp-n-p/jj591559%5C%28v=pandp.10%5C%29.

[18] *Port Infrastructure Market Research, 2030.* URL: https://www.alliedmarketresearch.com/port-infrastructure-market. (accessed: 27.09.2022).

[19] *ASSIST-IoT – Architecture for Scalable, Self-*, human-centric, Intelligent, Secure, and Tactile next generation IoT.* URL: https://assist-iot.eu/. (H2020-EU.2.1.1, Grant agreement ID: 957258, 2020-2023).

[20] *Dotty documentation. Dropped: Scala 2 Macros.* URL: https://dotty.epfl.ch/docs/reference/dropped-features/macros.html. (accessed: 1.10.2022).

[21] *http4s: typeful, functional, streaming HTTP for Scala.* URL: https://http4s.org/. (accessed: 1.10.2022).

[22] *Metals: Scala language server with rich IDE features.* URL: https://scalameta.org/metals/docs/. (accessed: 1.10.2022).