FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Decentralized Run-time Architecture Tracking

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Bernd Rathmanner
Matrikelnummer 0825340

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
Mitwirkung: Christoph Mayr-Dorn, Ph.D.

Wien, 21. Januar 2016

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Decentralized Run-time Architecture Tracking

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Bernd Rathmanner
Registration Number 0825340

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
Assistance: Christoph Mayr-Dorn, Ph.D.

Vienna, 21. Januar 2016 _____ _____

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Bernd Rathmanner
Deutschgasse 3-5/4, 2700 Wr. Neustadt

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

"I'm a man of few words. [long pause] Any questions?"
— Captain Tenille*, The Simpsons*

I hereby want to thank my supervisors, Christoph Mayr-Dorn and Schahram Dustdar, for their great support. Especially I want to thank Christoph for his great patience throughout the development of this thesis.

A special thanks goes to my parents, Maria-Helene and Reinhard, for their guidance, support and patience. At this point I want to dedicate this thesis to my father who is currently undergoing cancer treatment, I hope you will get better soon. I also want to thank my brother René, my girlfriend Tamra and my best friend Herbert for their encouragement and support.

Last but not least I want to thank my fellow students, Gerald, Janos, Jürgen and Michael, which have made the university life a pleasant one. I also want to thank my colleagues Barbara, Guido, Katja, Marius and Venkatesh for making the working life enjoyable.

Abstract

Distributed applications are becoming more common, especially cloud-based applications are a major topic in recent research. These application mostly contain complex application logic and should be built for dynamic adoption. Thus monitoring of an applications components is an important topic for adaption and scalability.

There currently are different methods of adaption approaches: localized and remote.

Localized approaches do not always yield optimal solutions due to their localized views. Adoption strategies might include or effect multiple parts of the application, or might need information of those parts to create an optimal adoption strategy. Due to the localized view it is impossible to assess the impact of the adaption effects in one part of the system on other system parts, thus yielding sub-optimal adaption strategies and unforeseen side effects.

Remote approaches on the other hand provide a holistic view which allows for more intelligent adoption strategies. But existing solutions have limitations with distributed applications. Pure probing might not be feasible (e.g. bandwidth, granularity, ...) and the given level of details may not be needed by the adaption control.

We propose a distributed architecture tracking framework based on *architecture-based-self-adaption*. By decoupling the application and the model generation itself we are able to generate a holistic view of the overall application architecture. Thus it is possible to perform optimal adaption decisions for the application.

Our proposed framework is evaluated using a simple distributed application, that is a cloud-based publish-subscribe system.

Kurzfassung

Forschungen auf dem Gebiet der verteilten Anwendungen belegen dass diese immer mehr Bedeutung gewinnen. Besonders Applikationen die in Cloud-basierten Umgebungen laufen

Diese Anwendungen enthalten meist komplexe Anwendungslogiken die darauf ausgelegt sein sollten sich dynamisch an eine sich verändernde Umgebung anzupassen. Um Anpassungen aufgrund solcher Umgebungen automatisiert durchführen zu können ist es wichtig Anwendungen und deren Umgebungen zu beobachten.

Es existieren verschieden Ansätze um auf verschiedene Ereignisse, die während der Laufzeit einer Anwendung aufträten, zu reagieren. Hierbei unterscheiden sich die Ansätze nur durch die Umgebung in der sie eingesetzt werden.

Lokale Adaptionsstrategien basieren auf Informationen die eine Anwendung lokal sammeln kann. Dies kann zu Adaptionsszenarien führen die nicht optimal für eine verteilte Anwendung sein können. Solche Adaptionsstrategien umfassen meist mehrere Teile einer Applikation oder benötigen Informationen von diesen um eine optimale Strategie zu erstellen. Aufgrund der Unvorhersehbarkeit, die eine solche Adaptionsstrategie auf eine verteile Applikation haben kann sind diese für verteilte Anwendungen nicht zu empfehlen.

Durch externe Adaptionsmechanismen ist es möglich einen vollständigen Überblick über eine verteile Applikation zu erhalten. Dies ermöglicht es bessere Adaptionsstrategien auszuarbeiten. Existierende Ansätze weisen jedoch Beschränkungen für optimale Strategien für verteilte Applikationen auf. Dies kann auf die verwendeten Ansätze zurückgeführt werden wie Informationen über das verteilte System gesammelt werden. Eine simple Sondierung von bestimmten Informationen könnte aufgrund von diversen Faktoren (z.B. Bandbreite, Granularität, ...) nicht praktikabel sein. Des weiteren könnten die gegebenen Informationen nicht ausreichend für Erstellung einer optimalen Adaptionsstrategie sein.

Wir stellen ein Framework vor welches speziell auf verteilte Applikationen ausgerichtet ist und auf *architecture-based-self-adaption* basiert. Unser Ansatz entkoppelt die Applikation von der Adaptionslogik. Dies erlaubt es einen Überblick über die gesamte Applikation einfach zu erstellen und aufgrund dessen eine optimale Adaptionsstrategie zu erarbeiten.

Unser vorgestelltes Framework wird mittels einer simplen verteilten Applikation, einem cloud-basierenden Publish-Subscribe System, evaluiert.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Distributed systems are more and more associated with self-* properties. Especially the importance of self-adaptive systems is increasing. Such systems are required to handle different scenarios for self-adaption [7]:

- A *system error* covers an undesirable condition from the target system itself.
- *Environmental changes* and *resource variability* describe an undesirable condition that arises outside the target system but causes problems for the target system itself.
- A *change in user needs* consists of a change to the requirements of the target system.

These scenarios share one common property of being a change that was not planned during the system's development.

To cover these scenarios self-adaptive system must be able to monitor and analyze changes as well as adapt to those. There currently exist two different approaches to the realization of self-adaptive systems:

- *Internal mechanisms* are heavily used in today's software. These mechanisms include exception handling and fault-tolerant protocols, which are easy to use and integrated in most of today's programming languages. They provide a localized view of the system and are directly integrated in the system itself.
- *External mechanisms* provide a separation between the adaption itself and the target system to be adapted. The adaptation logic is extracted from the target system and treated as a separate component. Prominent existing approaches are the IBM MAPE loop [40] illustrated in Figure 1.1 and the Closed-Loop control based on the Rainbow framework [7] illustrated in Figure 1.2.

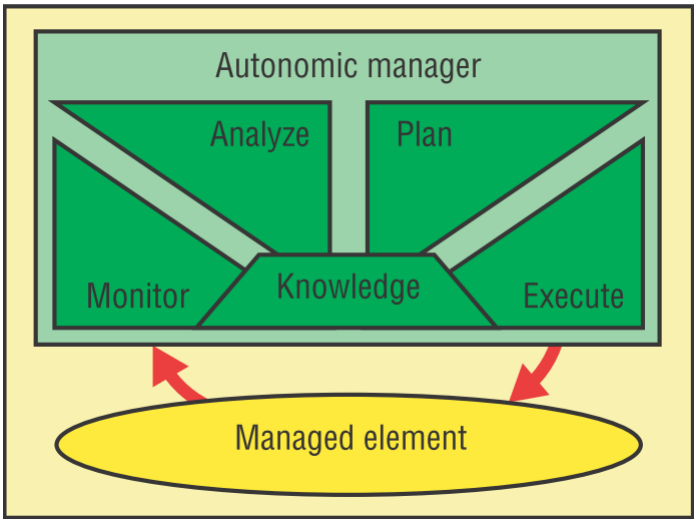


Figure 1.1: IBM MAPE loop [40]

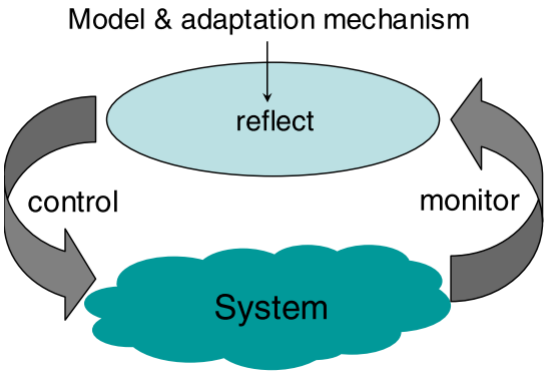


Figure 1.2: Closed-Loop Control [7]

To react to different changes self-adaptive systems need a model to reflect the system and it's properties. Some recent work in this field suggests an architectural model. Such models provide a global system perspective, expose important properties and constraints [16, 43, 44]. This type of adaption is also known as *architecture-based-self-adaption*.

1.1 Motivation

Distributed applications are becoming more common, especially with the arising of cloud-based applications distributed systems containing complex business logic have been created. Such systems have been a major topic in recent research as well as the real world. These applications

should be built for dynamic adaption. The first step to the realization of self-adaptive systems is to include monitoring for the extraction of important system properties.

There currently exist different methods of adaption approaches which do not yield optional results for distributed systems:

- *Localized* approaches only provide a localized view of a single part of the application. Adaption strategies might include or effect multiple parts of such applications, or might need information of other parts to create an optimal adaption strategy. Thus such approaches might lead to undesired system adaptions yielding non-optimal application states with unforeseen side effects.

- *Remote* approaches on the other hand provide a more holistic view of an application which allows for more intelligent adaption strategies. But existing solutions have limitations with distributed applications. Pure probing might not be feasible (e.g. limited bandwidth, granularity, etc.) and the given level of details may not be needed by the adaption control.

- Other approaches are e.g. based on hierarchical decomposition of adaption managers, hiding details within each level. These approaches include an inefficiency if there are many decomposition levels. Upwards monitoring- as well as adaption (downward) events take some time to reach it's destination. These latency issues might cause undesired adoption strategies.

We propose a distributed architecture tracking framework based on *architecture-based-self-adaption*. We are able to deploy our framework localized or remotely using both kinds approaches. Thus we are able to generate a localized architecture view for each part of an application and aggregate them to show the overall architecture of a distributed application.

Cloud-based environments create further challenges for distributed systems which aim to optimally scale such applications. Resources may be added by creating new virtual machines and thus extending a distributed application to handle increased system load and vice versa to decrease the costs of running them. Thus we chose to run our evaluation scenario in such an environment. From the previously described components of our evaluation application we run most of them on virtual machines in the cloud. Namely publishers and the message broker are each run on dedicated machines. To be as close as possible to a real-world scenario we run the subscribers in a local (non-cloud) environment. This allows us to create audio streams in the cloud and start the playback on commodity hardware, i.e. a Personal Computer (PC).

1.1.1 Motivated Scenario

We have aimed to evaluate our proposed approach using scenarios that already exist in the real world, thus we have chosen to create a distributed system for audio-based streaming. This scenario has some interesting traits that make it challenging for a distributed system, this includes,

but is not limited to, the real-time transfer of data and different quality of service levels to ensure that the provided streams may be consumed without failures.

Furthermore we have extended the simple audio-based streaming scenario by using a publish-subscribe system to handle multiple audio streams at the same time. This allows us to process multiple stream sources simultaneously and gives clients the ability to only subscribe to certain audio streams based on their preferences.

Our evaluation application consists of three components, publisher, message broker and subscriber, as shown in Figure 1.3. We have created a generic publish-subscribe system that is capable of delivering all kinds of messages. The transport of audio data creates some challenges for a publish-subscribe system. This includes the special handling of audio-based meta-data and the delivery of messages in the correct order to support a successful playback of audio streams for clients.

The scenario in general was inspired by the work presented in [18].

Figure 1.3: The components of our evaluation application each run separately.

Creating a dynamic publish-subscribe system where many clients may create difficulties involving the dynamic handling of these. Most environments provide capabilities to handle these scenarios with little to no effort, but some architecture-based software adaptation approaches do not provide such capabilities. This requires us to overcome this shortcoming by extending the approach itself to provide these capabilities.

Our approach is aimed to overcome these shortcomings by providing methods to directly interact with the runtime architecture of an application. This allows the application to directly manipulate the architecture and dynamically create or remove components or connectors. By directly exposing the architecture we are not limited to a certain design- or architectural-pattern. Thus our approach may be used in all kinds of architectures not just publish-subscribe systems.

Choosing a publish-subscribe system as our evaluation application ultimately allows us to use it directly in our approach.

1.2 Results of the Master's Thesis

The effective outcome of this thesis is a distributed architecture tracking framework providing the following capabilities:

- **Local and decentralized (sub-)architecture management**
  The architectural model will be made ware of which elements (i.e. components or connectors) are located on different hosts. Thus enabling control differentiation over those elements that are under immediate control and those that require remotely triggered adaptation. By decoupling the application and architectural model generation we are able to directly create a holistic view of the application as well as a view of the local architecture.

- **Architecture probes and sensors for cloud application, based on our evaluation scenario**
  The prototypical tools will allow the collection of architectural elements and their configuration in cloud-based environments. This includes the implementation of specific probes and sensors tailored to the evaluation scenario.

- **A distribution mechanism for aggregating the overall architecture on specific granularity levels**
  Based on the publish-subscribe paradigm higher level architecture model managers will be able to specify at what granularity level they require change events (i.e. added/removed components/connectors/links) and receive those events from various localized architecture managers for ultimately constructing a holistic system architecture view.

The created framework is evaluated using a simple, cloud-based, publish-subscribe system. Thus showing the integration of the framework, the creation of the architectural model and the advantages of using the framework.

1.3 Structure of the Master's Thesis

The topics of this thesis are organized as follows:

Chapter 2 will give an overview of related research work for runtime adoption and the handling of runtime models. Additionally related work on the field of dependency injections is presented.

Chapter 3 will introduce the tools and methods that will be used in this thesis. Here we will cover the description of the architectural style and the associated technologies.

Chapter 4 presents the methodology of our approach and will present the our evaluation scenario in more detail.

Chapter 5 presents our architecture tracking framework. We will explain the details of our approach and the resulting framework. We will also show how the created framework can be integrated into different kinds of applications.

Chapter 6 describes the evaluation of our approach using a simple publish-subscribe system running in a cloud-based environment. We shall give a detailed overview of the scenario and the

results we have extracted. Due to the cloud environment we can show how our approach works in a decentralized environment.

Chapter 7 summarizes our work and will give an outlook to possible future work.

CHAPTER 2

Related Work

This chapter gives an overview of existing research work regarding self-adaptive systems. Here we will focus on the used models during design- and runtime and how these models are created and extracted.

Additionally we will compare Myx and especially myx.fw to other existing solutions for (architectural-) dependency injection.

2.1 Runtime Architecture Modeling

The Myx architectural pattern is one of many patterns used for architecture-based software adaptation. In this section we will compare different approaches that employ runtime monitoring as our approach does. Here we will focus on the way each approach represents a running (distributed) system.

One of the first approaches of architectural software adoption was introduced by Gorlick and Razouk [32] by using automated agents, called weavers, to monitor a distributed system consisting of networks of fine grained tool fragments called weaves. These tool fragments work with simple objects by using them as inputs and producing them as outputs and transporting them via ports. Weaves themselves provide methods to observe their behavior which is used by the weaver to dynamically adapt the weave.

One of the most popular approaches to runtime adaptability is the IBM's approach to autonomic computing [35, 40] which introduces autonomic managers and the IBM MAPE loop (see Figure 1.1). Each autonomic element is managed by one or more autonomic manager(s) which monitor the behavior of the element and it's connections to other autonomic elements. The autonomic manager is integrated at an architectural level.

In the following we will group the approaches by their employed (architectural) runtime model.

The C2 architectural style [49], a predecessor of the Myx architectural style, was one of the first component and connector based architectural styles that has been used to issue architectural changes at runtime [44, 45]. The runtime model is described by the C2 architectural style and deployed with the application which allows it to evolve at runtime. This approach is extended in [16] by describing the runtime model with XADL 2.0 and adding external monitoring for the runtime model.

A widely used approach to runtime monitoring is the use of probes and gauges. One of the first approaches to use probing to extract information about an architectural model at runtime was described by Schmerl et. al. [46]. The approach uses probes to extract information about the runtime system and gauges to propagate this information to a central entity creating the architectural runtime model. This model is represented using Acme Architecture Description Language (ADL) [23]. The defined probes represent a mapping of the architectural model and the runtime observations.

Clemens et al. [8, 9] introduced multiple architectural views at runtime that follow different styles. The paper introduces three view-styles: module-, component-and-connector- and allocation view-type. Each of these styles represent another aspect of software architectures.

Another representative approach to self-adaptive software is Rainbow [22] which uses external runtime architecture monitoring [24]. The running system is represented by a graph-based architectural model [7] that is based on the different views introduced by Clemens et al. [8]. In this model the nodes represent components (which can be sub-architectures) and the edges connectors, where each of them can have different kinds of architectural properties [4]. The monitoring component is a central entity monitoring the whole application [25], which may be decentralized. The extraction of architectural properties is done via probes and gauges and translated into the runtime model [5]. The rainbow framework comes with an integrated adaptation engine which uses the Stitch language to define all kinds of adaptation strategies [6].

The Mobility- and Adaptation-Enabling Middleware (MADAM) [21] is an approach focusing on the different variants of software architectures. It uses component frameworks [48] at different stages in an applications development, that is at design- and runtime. As in other approaches the component framework consists of components which may be atomic or a composition of other components. The components interact using ports for interaction (connectors). Both components and connectors have different properties attached to them which may change at runtime. The runtime monitoring is done by observing the properties attached to components and connectors externally [33]. These observations are reflected into the runtime model, which is called *instance architecture model*.

Di Marzo Serugendo et al. use an service-oriented architectural model in their approach presented in [17]. Each defined component contains runtime-meta-data. This meta-data is extracted and stored directly by the runtime environment, which itself uses a service-oriented architecture as well.

Weyes et al. [56] describe a truly decentralized approach to runtime adaptation. The architecture of an application is composed of self-adaptive units. As such self-adaptive units have only control over the part of the application that is running on the local (sub-)system. Unlike the monitoring of centralized setting, such as Rainbow or MADAM, the self-adaptive units

may only directly monitor the local system, leading to a partial model of the complete system. Therefore self-adaptive units may need to interact with other units to share the locally collected data. The monitoring of a self-adaptive unit is composed of two parts. The first one being platform-dependent hooks which extract values from the runtime system and platform-independent interpretation logic which validates the monitored data and passes it to the runtime model. The runtime model is represented by different meta-level models: the system model that is a representation of the running system, the concern model which models objectives of a self-adaptive unit (e.g. an optimization problem), the working model representing shared information between meta-level computations and the coordination model used to describe the coordination between other self-adaptive units. Meta-level computations are the typical feedback control loop computations found in self-adaptive systems.

Amoui et al. [1] propose a low level approach to self-adaptive software. The Graph-Based Runtime Adaptation Framework (GRAF) can be integrated into Java applications. The runtime model is based on TGraphs [19], a graph based model describing the state of the adaptable software. To handle the adaptations different behavior descriptions are directly integrated into the runtime model. The state of the application is extracted using Java annotations and Aspect Oriented Programming (AOP). It is described by explicitly exposed variables, control flow points as exposed methods and actions as different behaviors for control flow points. The model is updated as soon as the value of a variable changes which triggers the integrated adaptation engine. As this is a low level approach it may only be used for adaptations of one application instance which makes it unsuitable for distributed systems.

Using model-engineering based approaches and meta-models to define the architectural models of an application is recently gaining more attention in research. Thus we will discuss some of those recent approaches.

The approach described in [57] uses an architectural model which separates non-adaptive behavior from adaptive behavior. Global invariants define properties to be satisfied by the runtime model. The verification of such properties is done using model checking.

Solomon et al. [47] introduce two separate models, a Platform Independent Model (PIM) and a Platform Specific Model (PSM). The PIM represents the general architecture of an application. The PIM is represented as a UML meta-model and is transformed into the PSM using different mappings. The PSM is further partially transformed directly into executable code. Different kinds of sensors send data about the running system to the runtime model which is filtered and the PIM is updated accordingly.

The approach introduced by Vogel et al. [53] takes the idea of multiple meta-models even further. The source model, which represents the runtime architecture of a managed element (e.g. the whole application), is defined and updated using different kinds of sensors. The source model is transformed into multiple target models, each raising the level of abstraction and representing a different view on the architecture of the application. These models are used by the feedback loop (e.g. the IBM MAPE loop [40]). This transformation is done via Triple Graph Grammars [26, 27]. The adaptation of an application is done by reflecting the adaptive changes to the different target models back to the source model and thus to the managed element itself using effectors [52].

The interaction of the different meta-models introduced by [53] is extended by [55] by introducing a megamodel to describe the relations between the different models at runtime.

The approach presenting in [54] extends [53] by providing an incremental model synchronization between the running system and the source model, again based on Triple Graph Grammars.

Luckey et al. introduce an approach in [42] where UML is used throughout the adaption process, as a specification language and a meta-model. Each adapt case consists of three parts, a *Monitor*, an *Adaptation* and an *Adaptation Context*. The *Monitor* is used to monitor the a managed element at runtime. It also defines the allowed values of these elements, named a corridor. Once the value of an element leaves the specified corridor an event is emitted which starts the adaptation of the managed element. Each adapt case is mapped to a MAPE-K model [40] thus directly providing a feedback loop.

M. Vierhauser et al. [50, 51] propose a general purpose monitoring framework for Systems of systems (SoS) named REMINDS. The approach is based on the Requirements Monitoring Model (RMM) for specifying monitored requirements. These requirements are monitored using probes that are integrated in the system to be monitored. All information extracted by a probe is packaged into so called event models that are forwarded to the framework.

All of the approaches described above do not completely satisfy our requirements. Pure probing does not allow us to quickly forward changes in an application's architecture, the change may only be observed on the next probing. The presented model-engineering based approaches are not directly suitable for our approach but rather provide methods for extensions.

Choosing the Myx architectural style and the associated tools, like myx.fw, allows us to directly hook into the architecture and extract all required information to create a runtime representation of an application's architecture.

2.2 Dependency Injection Frameworks

The Myx architectural style is a component and connector based architectural style. It comes with an architecture framework to bridge the gap between the concepts of an architectural style the capabilities of a given platform called myx.fw. Being an architecture framework it provides a kind of dependency injection by using the definitions of the architectural style and an ADL to create an instance of an application.

In this section we will compare myx.fw to frameworks and approaches that support similar dependency injection features.

The myx.fw dependency injection is purely object oriented and directly coupled with the Myx architectural style [2]. myx.fw interfaces are directly mapped to Java interfaces, thus if a component or connector provides such an interface it directly exposes the Java interface. The brick may choose to implement the Java interface itself or return a auxiliary object which does. To acquire an instance of such a required interface in another brick myx.fw provides utility methods which provide this functionality. myx.fw looks up the implementing class which is specified in the architectural description of a brick and returns a fully initialized instance. The

application is responsible to actually acquire an instance of a required interface, thus myx.fw does not inject an object automatically.

The Context and Dependency Injection (CDI) provided by the Java EE application programming environment is based on so called beans [37]. Nearly all Java classes may be used as such beans. To use CDI an application requires a file called *beans.xml* to be present, which in most cases is completely empty for it is only used to configure specific parts of CDI.

The dependency injection heavily relies on Java annotations. To inject a dependency into a field it is required to annotate the field with the *Inject*[1] annotation. This annotation defines the implementation to use at the time of development or deployment. To use a specialized implementation it is possible to add another annotation which specifies the specialized type.

It is possible to have more than one bean implementation to use for a specific purpose. To define alternative implementations of a bean the class has to be annotated with the *Alternative*[2] annotation. The definition of the alternative to use is by specifying the implementation in the *bean.xml* file.

Producer methods provide a way to inject objects which are not beans or which may vary at runtime, e.g. a random number. Such methods are annotated with the *Produces*[3] annotation and another annotation which defines where the dependency injection is used. These methods return the injected value. If the value varies between invocations the field to be injected must be of type *Instance*[4]. Producer methods combined with alternatives provide a way to choose the implementation of a bean at runtime. In this case all possible implementations have to be known by the producer method and it returns the implementation to choose. In most cases the possible implementations are directly injected into the producer method by specifying them as injected parameters using the *New*[5] annotation which constructs new instances of the implementations.

Enterprise Java Beans (EJB) are beans running in a custom container provided by the underlying Java EE server, e.g. a Glassfish server. There are two different types of EJB: session- and message-driven beans. Session beans may be directly invoked by their exposed interface. Message-driven beans do not expose an interface but rather providing a message listener which can be invoked by sending a message via e.g. Java Message Service (JMS).

Session beans may be injected using the *EJB*[6] annotation which is the simplest way of obtaining such beans. Another way is to use Java Naming and Directory Interface (JNDI) lookups by invoking the static `lookup` method on the *InitialContext*[7] class.

The dependency injection of the Spring Framework is centered around the Spring Inversion of Control (IOC) container [38]. Injectable objects are once again referred to as beans. There exist multiple forms of defining injectable beans: via eXtensible Markup Language (XML),

[1] `javax.inject.Inject`
[2] `javax.enterprise.inject.Alternative`
[3] `javax.enterprise.inject.Produces`
[4] `javax.enterprise.inject.Instance`
[5] `javax.enterprise.inject.New`
[6] `javax.ejb.EJB`
[7] `javax.naming.InitialContext`

annotations or plain Java. Spring supports all kinds of classes to be injectable beans, which is not possible with CDI. The Spring based dependency injection framework is compatible with the CDI based annotations.

By reading the bean configuration with an instance of the *ApplicationContext*[8] interface enables to use the `getBean` method to retrieve a fully configured bean including all dependencies of it.

The Spring framework supports the automatic wiring of beans, that is a required bean does not have to be referenced. According to the documentation this approach has some limitations and a required bean should explicitly be referenced.

Guice is a dependency injection framework provided by Google [31]. Unlike the other discussed frameworks the configuration of injectable objects is done solely via Java code. Dependencies of beans can be defined via so called bindings [29]. The bindings are configured by creating an implementation of the *AbstractModule*[9] classes `configure` method. In this method all dependencies of an injectable class can be bound by different types [28]. The most important kinds of bindings are:

- Linked bindings — Map a type to it's concrete implementation.
- Binding annotations — Annotations can be used to specify the concrete implementation to use if multiple implementations exist.
- Instance bindings — It is possible to bind to a concrete instance of an object.

To define an object to be injected Guice uses the *Inject*[10] annotation and supports constructor-, method- and field injections. Here it is possible to mark method- and field injections as optional allowing them to only be injected if there is a matching implementation.

As the Spring framework Guice is compatible with CDI based annotations [30].

The described approaches offer some excellent methods of dependency injections, CDI, EJB, Spring and Guice are more mature in this field than myx.fw. We have chosen myx.fw because of the direct support for the Myx architectural style.

[8] `org.springframework.context.ApplicationContext`
[9] `com.google.inject.AbstractModule`
[10] `com.google.inject.Inject`

CHAPTER 3

Background

In order to gain thorough understanding of the methods used to create this thesis, this chapter will work through the technologies that were used by our approach and why we chose them.

3.1 Myx Architectural Style

The Myx architectural style [10,36] is a set of constraints put on development to elicit beneficial properties. It is based on the C2- [49] and the Weaves architectural style [32].

The style enforces the following rules on the architecture:

- Basic entities are components and connectors, collectively called bricks.
- Components are the source for computation providing services to other bricks.
- Connectors are the source for communication by moving data between bricks.
- Bricks communicate only through well-defined interfaces with specific service-types and directions. Two different service-types exist: provided and required. Provided interfaces are used for bricks providing services for others and are invoked by those bricks. Required interfaces on the other hand are used to invoke services provided by other bricks. The direction of an interface determines the flow of control. Three directions are usable: in, out or inout.
- Bricks are connected via links. They are associations between provided and required interfaces, having exactly two endpoints. Links may only occur between the two different service-types. Links between the same service-type are not permitted.
- All bricks have two domains called top and bottom. Each interface must be assigned to one of those. Interfaces of both service-types may occur on each domain. Links are used to connect a interfaces on a brick's top domain to an interface of another brick's bottom domain. Thus inducing layering of Myx architectures.

- The creation of a cycle is not permitted. Thus a brick may never be above or below itself and a link may not connect a brick to itself.
- Applications have at least one main thread of control. Each brick may create and maintain new threads as necessary.
- Communication is based on specific patterns which are described below.

3.1.1 Communication Patterns

The Myx architectural style provides different communication patterns. In this subsection we aim to describe the ones most common.

These patterns can be categorized into synchronous- and asynchronous invocations. Synchronous invocations are only permitted upwards, meaning from a brick's top domain to another brick's bottom domain. Asynchronous invocations on the other hand are permitted in both directions. The difference between the two categories is described below by describing each pattern.

Synchronous Call

This pattern represents a synchronous procedure call, see Figure 3.1. Here no explicit connector is used between two components, but the call itself is an implicit connector. A call transfers the thread of control from the calling component to the called component.

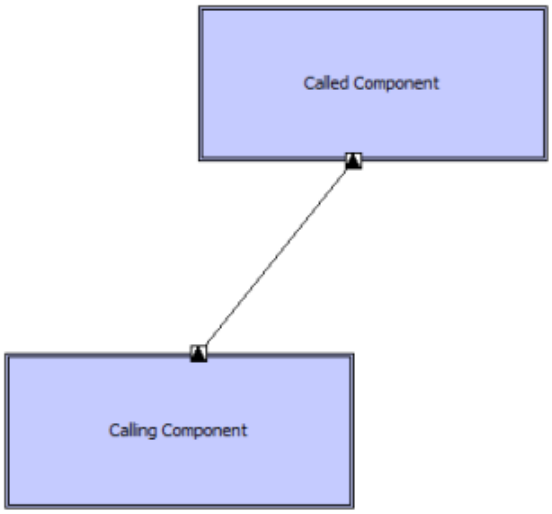


Figure 3.1: Myx Synchronous Call Pattern [36]

Synchronous Call with Proxy

This pattern is an extension to the Synchronous Call by an intervening connector called proxy. The proxy is used to pass the calls onto the called component but may provide additional features enabling dynamic linking, data format transformation, logging or debugging, see Figure 3.2.

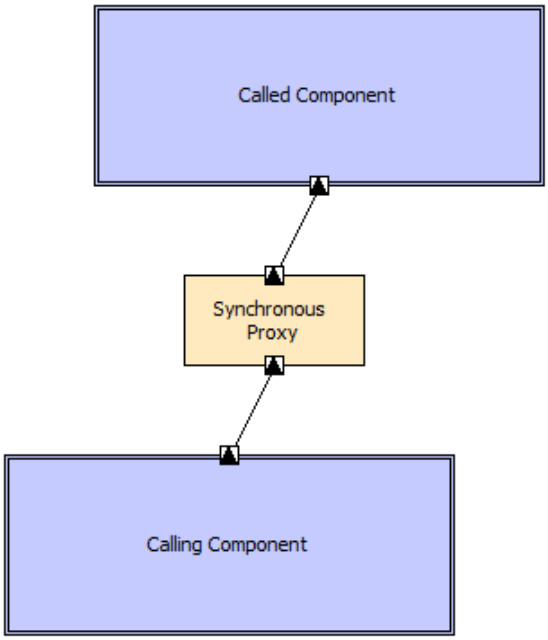


Figure 3.2: Myx Synchronous Call with Proxy Pattern [36]

Asynchronous Notification

This communication pattern allows bricks of higher layers to notify underlaying bricks of state changes, see Figure 3.3. This is done by the asynchronous event pump connector. Such a connector receives a message on its provided interface and forwards that message to all bricks connected to its provided interface. To ensure the asynchronous communication the messages are forwarded in a separate thread.

Asynchronous Request

This pattern works the same way as the asynchronous notifications pattern, but reversing the flow of control to upwards, see Figure 3.4.

3.2 xADL

xADL 2.0 is an extensible ADL based on XML [14, 15]. Using an ADL allows for the easy description of different software architectures. Thus we use xADL to describe the design- and runtime architecture of a monitored application as well as our approach itself.

xADL is defined as a set of XSDs providing it with extensibility and flexibility. It currently contains the following set of schemes:

- run- and design-time elements of a system
- support for architectural types

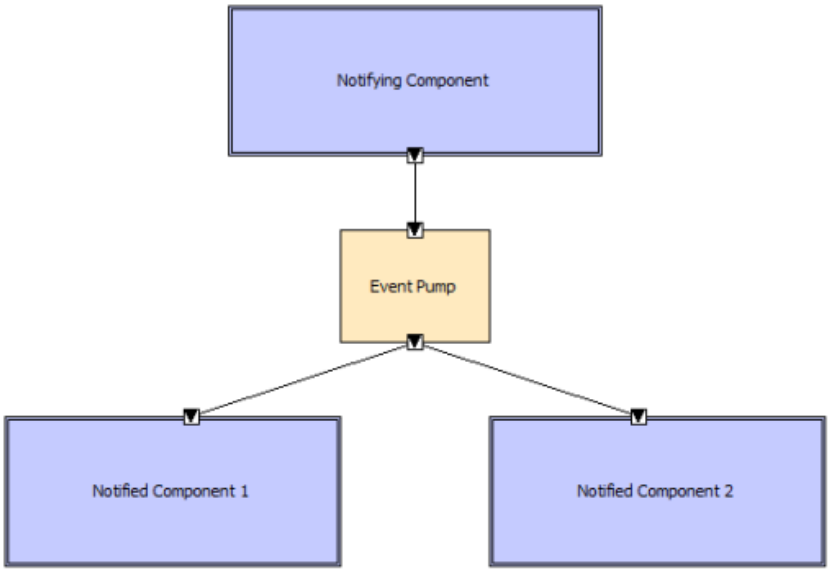


Figure 3.3: Myx Asynchronous Notification Pattern [36]

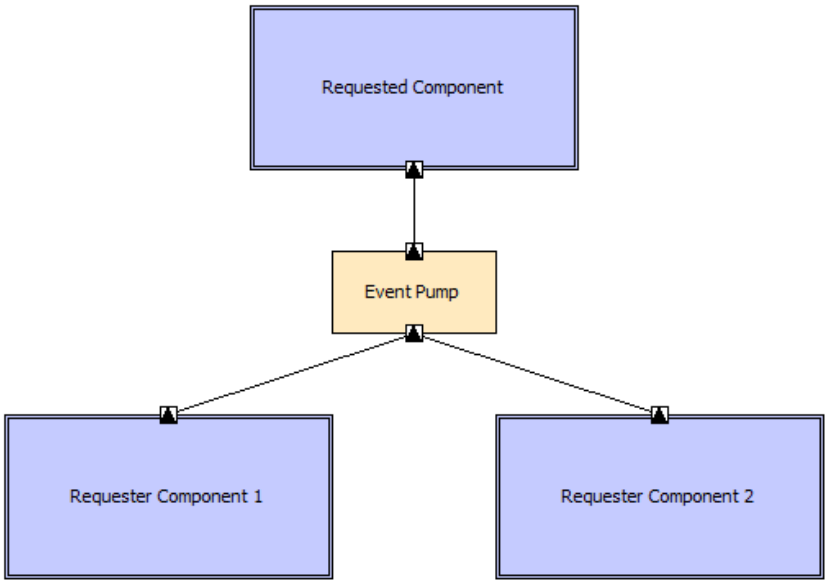


Figure 3.4: Myx Asynchronous Request Pattern [36]

- advanced configuration management concepts such as versions, options, and variants

xADL is not bound to any particular architectural style, tool or methodology thus it may be used if an architectural style changes. The hierarchical composition of the xADL schemes can be seen in Figure 3.5. The purposes and features provided by each schema are described in Table 3.1.

xADL is an application of xArch [11], a standard for XML-based representation of soft-

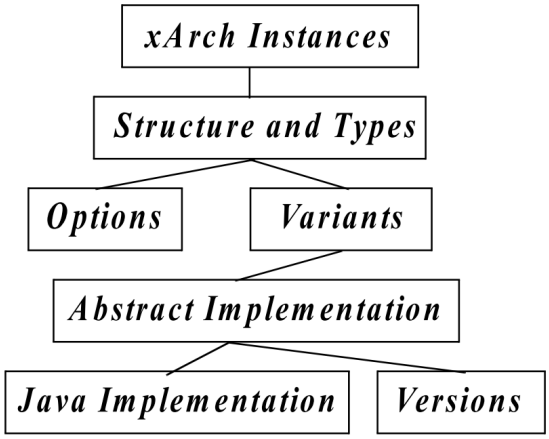


Figure 3.5: Hierarchical composition (dependencies) of the xADL 2.0 XSDs. Child nodes are dependent on their parent node [15]

ware architectures. It provides a common core XML notation as a stand-alone representation of architectures and as a starting point for more advanced XML-based ADLs. xArch consists of one single XML schema for defining instance structures for architectures, called *instances*. This XML schema is also the basis of xADL, which can be seen in Figure 3.5 as the top level element.

One of the reasons we have chosen xADL was it's close connection to the Myx architectural pattern which eases the development of our approach. But due to the fact that xADL is not bound to an architectural style enables us to switch it with little to no effort.

3.2.1 Schemes

In our approach we focused on three xADL schemes, *xArch Instance*, *Structure and Types* and *Implementation*. This subsection will be used to describe these schemes in more detail.

xArch Instance

This schema is the basis for all xADL schemes. It provides the root element for all xArch and xADL documents as well as the elements for identifiers, descriptions, directions, links to other XML elements, arbitrary groups and elements for hierarchical construction.

As the name suggests its main purpose is to describe instances, that is component-, connector-, interface- and link instances. An *InterfaceInstance* is composed of an *Identifier*, a *Description* and a *Direction*. A *ComponentInstance* or *ConnectorInstance* is composed of an *Identifier*, a *Description*, multiple *InterfaceInstanes* and a *SubArchitecture*. To define links between component and connectors this schema defines *LinkInstances*, which consists of a *Description* and two XML links to *InterfaceInstances*.

It is used to represent the runtime architecture of an application.

Purpose	Schema	Features Included
Architecture Modeling - Description and Prescription	**xArch Instance**	Component, connector, interface, and link instances; arbitrary groups; hierarchical construction.
	Structure and Types	Design-time architectural prescription; architectural structure (components, connectors, interfaces, and links), programming-language style types-and-instances model; hierarchical construction via types.
Instantiatable Architectures	**Implementation**	Abstract placeholder for implementation information for components and connectors.
	Java Implementation	Java-specific implementation information for components and connectors.
Architecture Configuration Management / Product Family Architectures	**Options**	Optional components, connectors, and links.
	Variants	Variant component and connector types.
	Versions	Version graphs for components, connectors, and interfaces.

Table 3.1: xADL 2.0 schemes and features provided [15]

Structure and Types

This schema defines the design-time architectural perception. That is it provides elements for components, connectors, interfaces and their types.

Types are definitions for their corresponding elements. An *InterfaceType* is composed of an *Identifier* and a *Description*. A *ComponentType* or *ConnectorType* is composed of an *Identifier*, a *Description*, multiple *Signatures* and a *SubArchitecture*. A *Signature* is a method to say that a component or connector should contain a specific interface.

The structures of an architecture are described as *Component*, *Connector* and *Interface*. A *Component* or *Connector* is composed of an *Identifier*, a *Description*, multiple *Interface* definitions and their types. A *Interface* is composed of an *Identifier*, a *Description*, a *Direction* and a *Signature*. To connect components and connectors this schema defines *Links*, which consists of a *Description* and two XML links to *Interfaces*.

Implementation

This schema extends components, connectors and interfaces (and their types) to contain implementation specific information. That is an element called *Implementation*. Extensions of this schema, such as the *Java Implementation* schema further add Java specific information, such as a main class by the element *JavaClassName*. This information gives us the possibility to correctly instantiate ad architectural description.

3.2.2 Choosing xADL

We have chosen xADL, especially xADL 2.0, over other approaches, like UML or Systems Modeling Language (SysML), mainly due to the tight integration with the Myx architectural pattern. But due to the fact that xADL is not bound to any architectural style and the easy extensibility makes it a logical choice for our approach.

CHAPTER 4

Achieving Decentralized Application Architecture Monitoring

To monitor the architecture of an application at runtime one must be able to access the architecture of such application at runtime or keep track of it by some other means. In Section 2.1 we have described other approaches that utilize probing for architectural properties or integrate them into the source code of an application and expose them to be monitored by external services.

We propose an approach that seamlessly integrates into an application with minimal overhead. It actively publishes events about specific architectural properties and aggregates them into the applications runtime architecture. Thus the architecture's changes are directly reflected into the runtime model. Figure 4.1 shows how different applications are monitored by our approach. The Figure shows two application instances named *App A* and *App B*, both based on the Myx architectural style, and each running on different hosts. Both application instances are connected to each other by a link that cannot be created with the current tools associated with the Myx architectural style, depicted by the dotted line. Our approach, shown as the *Monitor* extracts the following architectural properties from the application instances (depicted by the lines originating from the *Monitor*):

- components and connectors
- local and external links between bricks
- a brick's runtime status
- properties about the host where the application is running

After the architectural properties have been extracted they are forwarded to be aggregated. This task is done by the *Monitor* and it forwards each extracted architectural property to the *Aggregator*, which is shown in Figure 4.2. The *Aggregator* uses the received architectural properties to create the runtime architecture of each application instance. By extracting properties for external

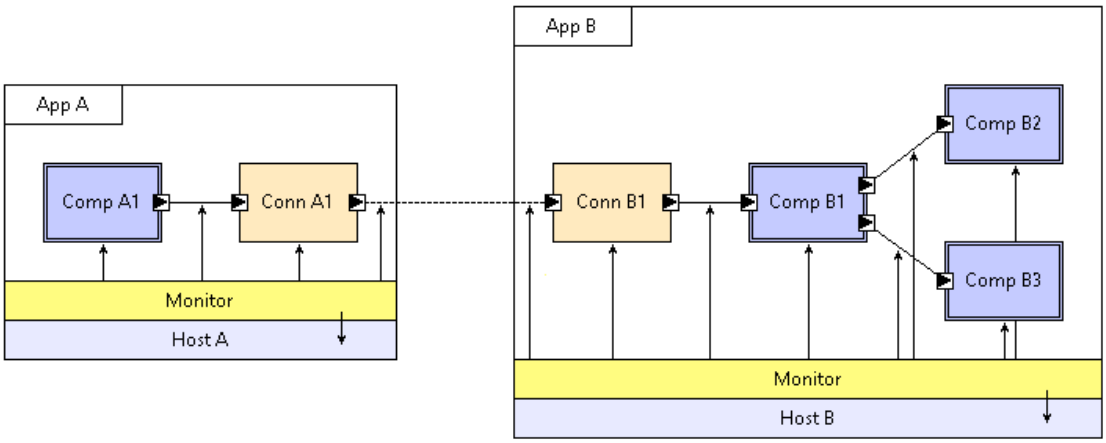


Figure 4.1: Extracting architectural properties from an application at runtime.

links we are not only able to create localized views of each application instance but can connect them to each other. To be aware of the decentralized architecture of distributed application we use the properties that are extracted from the host to integrate them directly into the runtime architecture and associate application instances with it.

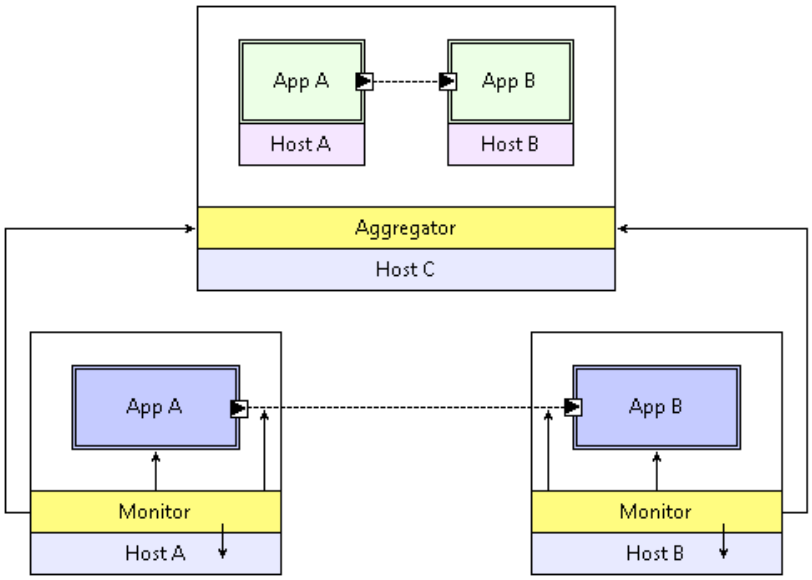


Figure 4.2: Aggregating (distributed) architectures.

In the following sections we will describe our approach in more detail. We will show how the runtime architecture of an application is created and how distributed architectures are supported as described.

4.1 From Blueprint to Instance

Both design-time and runtime architecture are utilized by our approach. That is we require the design-time architecture to be known to use it as a blueprint for the runtime architecture and use it directly to instantiate an application. This allows us to only forward certain architectural properties for the aggregation process to work instead of transferring all the information about the runtime architecture. The aggregation process uses the design-time architecture to extract all required information to create the full runtime architecture of an application.

Our approach requires the design-time architecture to be described using xADL. That is it has to be specified using the *Structure and Types* schema inside the xADL file. The resulting runtime architecture, also being described using xADL, uses the *xArch Instance* scheme as a basis. In the following we will describe how we are able to create the runtime architecture of an application using it's design-time equivalent as a basis and how it is possible to create a mapping between both of them.

To create a runtime architecture based on it's design-time architecture our approaches is composed of three steps:

1. **Extraction of architectural properties** — The extraction of architectural properties mostly takes place at the lowest possible level, that is the instantiation of an application. Here the main properties to create the runtime architecture are extracted by our approach.

2. **Propagation of extracted properties** — The extraction of architectural properties does not yet yield a runtime architecture. The extracted properties are packed into simple events containing all required information about the architectural property. All of them share the following properties:

   - A unique event identifier.
   - A time stamp that refers to the time the event was created.
   - An identifier referencing the running application.

   The properties that each different event contains is described in the following. The event is then propagated to the aggregation process.

3. **Aggregation of the runtime architecture** — Our aggregation process is based directly on the described events, which are the source for the resulting runtime architecture. Each event is handled differently by our approach which will be described in the following.

In the following we will show how the runtime architecture of an application is created using the described architectural properties and steps. We will also show what kind of information is extracted for each property and how it is used to create the resulting runtime architecture.

4.1.1 Basic Architectural Elements

The basic elements of a runtime architecture are components, connectors, the associated interfaces and the links ultimately connecting the elements.

These elements are based on the design-time architecture. xADL allows us to create instances for these runtime elements using the *xArch Instance* scheme. This scheme offers the basics to create a runtime architecture but it does not provide us with the capabilities to validate if the created runtime architecture complies to the design-time architecture. It also does not allow for a analysis if attached architectural restrictions have been violated. Using the *xArch Instance* scheme also requires us to extract all the information about these elements and propagate them to the aggregation process.

To ease this process we have created a xADL extension named *xArch Instance Mapping* which extends certain parts of the *xArch Instance* scheme. Namely it provides extended types for *ComponentInstance*, *ConnectorInstance* and *InterfaceInstance*.

The Listings 4.1, 4.2 and 4.3 show the XSDs of these extensions. Our scheme provides three new Types called *MappedComponentInstance*, *MappedConnectorInstance* and *MappedInterfaceInstance*. The two types *MappedComponentInstance* and *MappedConnectorInstance* are extensions to *ComponentInstance* and *ConnectorInstance* respectively. They both extend their base by a single property called *blueprint*. This property allows us to specify a link to the design-time component or connector. The *MappedInterfaceInstance* type extends *InterfaceInstance* with a single property called *type* which is used to provide a link to the interface's type which is specified in the design-time architecture.

The full *xArch Instance Mapping* XSD can be found in the appendix.

```
<!-- TYPE: MappedComponentInstance -->
<xsd:complexType name="MappedComponentInstance">
    <xsd:complexContent>
        <xsd:extension base="archinst:ComponentInstance">
            <xsd:sequence>
                <xsd:element name="blueprint" type="archinst:XMLLink" />
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Listing 4.1: The *xArch Instance Mapping* xADL schema extension for components.

```
<!-- TYPE: MappedConnectorInstance -->
<xsd:complexType name="MappedConnectorInstance">
    <xsd:complexContent>
        <xsd:extension base="archinst:ConnectorInstance">
            <xsd:sequence>
                <xsd:element name="blueprint" type="archinst:XMLLink" />
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Listing 4.2: The *xArch Instance Mapping* xADL schema extension for connectors.

```
<!-- TYPE: MappedInterfaceInstance -->
<xsd:complexType name="MappedInterfaceInstance">
    <xsd:complexContent>
        <xsd:extension base="archinst:InterfaceInstance">
            <xsd:sequence>
                <xsd:element name="type" type="archinst:XMLLink" minOccurs="0"
maxOccurs="1" />
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Listing 4.3: The *xArch Instance Mapping* xADL schema extension for interfaces.

Component and Connectors

The architectural properties for the creation and destruction of components and connectors are extracted directly from the underlying runtime which handles the instantiation and shutdown of the application. Our approach extracts the runtime identifier of the created instance as well as the blueprint identifier, which identifies the instance in the design-time architecture. Additionally we add the type of event, that is if the instance was added or removed, as well as the type of instance, being component or connector. All this information is packed into an event called *XADLEvent*, this kind of event represents changes to components and connectors.

Once our aggregation process encounters a *XADLEvent*, for a added component or connector, we validate if it is already contained within the runtime architecture. This is done by validating that the runtime identifier is not yet present in it. The next step is to create the instance based on it's blueprint. We either create a *MappedComponentInstance* or *MappedConnectorInstance* with the given runtime identifier and add the blueprint identifier as well. To fully create the instance we copy the other properties including the interfaces of the instance. Here we create *MappedInterfaceInstance*s which include the interface's type. The created instance is finally added to the runtime architecture. If the instance is already present in the architecture or the blueprint element cannot be found we do not add the instance.

If a component or connector is removed and we receive such an event we again validate if the instance is present in the architecture and remove it if found. This removal also includes all associated links that are still connected to it.

Local Links

The architectural properties for the creation of local links between components and connectors are extracted at the same level as information about components and connectors. Here the underlying runtime creates an event called *XADLLinkEvent* once a link is created. The event and thus the architectural property contains the information about both ends of the link, that is the runtime- and blueprint identifier of the source- and destination instance. Additionally the type of the interface is included as well as the type of event, that is if the link has been created or removed.

If a local link is created and our aggregation process encounters a *XADLLinkEvent* we extract the matching interfaces from the source- and destination instances by matching the interface's

type. This is be done by using the *MappedInterfaceInstance*'s type. Myx only allows us to establish links between two interfaces that correspond to the same interface type, which allows us to base our linking process on it. The interfaces are extracted by comparing the type of all interfaces on the source- and destination instance with the given type and use the first matching interface. The simplicity of this approach imposes some drawbacks. It is currently not possible for an instance to have multiple interfaces with the same type. If two matching interfaces could be extracted a *LinkInstance* is created between them.

Once the local link is removed we simple remove the created *LinkInstace* from the runtime architecture.

4.1.2 The Runtime Status

Just as the other architectural properties the runtime status of a component- or connector instance is extracted by the underlying runtime. Here we again extract the runtime- and blueprint identifier of the instance as well if the instance has entered a running state or has been stopped. This architectural property is represented by an event called *XADLRuntimeEvent*.

Our aggregation process currently handles these kind of events by appending the state of the instance to it's description directly in the runtime architecture.

4.2 Supporting Distributed Architectures

By using the previously described architectural properties we are able to monitor the runtime architecture of an application. Yet we are not able to fully monitor distributed applications because important information is still lacking from our runtime architecture.

In the following subsections we will discuss how our approach allows the integration of distributed architectures.

4.2.1 Recognizing Distributed Applications

We are yet not able to distinguish between local- and distributed architectures. Therefore our approaches uses a simple concept of hosts to describe the environment of an application instance.

This concept of a host is integrated directly into the runtime architecture of an application. That is multiple hosts may be added to the architecture which are described using a unique identifier. This identifier needs to be generated by the underlying runtime using different utilities. Each host allows for the definition of a description of the host itself, associated properties about the host and lists of hosted components, connectors, groups and sub-hosts. Thus our concept of hosts is used to describe the physical- or virtual environment of an application instance.

To be able to use this concept in the runtime architecture we have again extended xADL and propose an extension called *xArch HostProperty*. This extension provides a top level type called *HostedArchInstance*, which is an extension to the *ArchInstance* type introduced in the *xArch Instance* scheme. Listing 4.4 shows the XSD for the introduced type.

```
<!-- TYPE: HostedArchInstance -->
<xsd:complexType name="HostedArchInstance">
    <xsd:complexContent>
        <xsd:extension base="archinst:ArchInstance">
            <xsd:sequence>
                <xsd:element name="host" type="Host" minOccurs="0" maxOccurs="
                    unbounded" />
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Listing 4.4: The xADL schema extension for hosted architectural instances.

This type allows us to define an unlimited number of hosts for a runtime architecture. The definition of a host is depicted in Listing 4.5. By referencing components and connectors for each host we are able to get a more detailed view of the runtime architecture which allows us to distinguish distributed architectures.

```
<!-- TYPE: Host -->
<xsd:complexType name="Host">
    <xsd:sequence>
        <xsd:element name="description" type="archinst:Description" />
        <xsd:element name="hostProperty" type="Property" minOccurs="0" maxOccurs="
            unbounded" />
        <xsd:element name="subhost" type="Host" minOccurs="0" maxOccurs="unbounded"
             />
        <xsd:element name="hostsComponent" type="ElementRef" minOccurs="0"
            maxOccurs="unbounded" />
        <xsd:element name="hostsConnector" type="ElementRef" minOccurs="0"
            maxOccurs="unbounded" />
        <xsd:element name="hostsGroup" type="ElementRef" minOccurs="0" maxOccurs="
            unbounded" />
    </xsd:sequence>
    <xsd:attribute name="id" type="archinst:Identifier" />
</xsd:complexType>
```

Listing 4.5: The xADL schema extension for hosts.

The properties associated with a host are created as a simple map. Here a key may be associated with one ore more values. These properties are meant to be used to describe all kinds of information about the host itself, e.g. the network host-name or information about the current system load. Listing 4.6 shows the XSD of such properties.

```
<!-- TYPE: Property -->
<xsd:complexType name="Property">
    <xsd:sequence>
        <xsd:element name="name" type="archinst:Description" />
        <xsd:element name="value" type="archinst:Description" minOccurs="1"
            maxOccurs="unbounded" />
    </xsd:sequence>
</xsd:complexType>
```

Listing 4.6: The xADL schema extension for host properties.

Our approach does not yet support the depicted groups and sub-hosts. These elements may be used for further research and development. The full *xArch HostProperty* XSD can be found in the appendix.

The propagation of these architectural properties is once again based on events. Here we have created an extensions to our existing base event which adds the following properties:

- The unique identifier of the host.
- The type of event, i.e. is the property being added, updated or removed.

These properties are common the all architectural properties. In the following we will show how our approach integrates the different host based architectural properties.

Hosts

The architectural properties for general information about a host are extracted directly by the underlying runtime. This is done at the startup and shutdown of an application instance. Our approach extracts the unique host identifier as well as the name of the host. This information is packed into an event called *XADLHostInstanceEvent* which simply extends the base event with a description about the host, which is used to store the name of the host.

Once our aggregation process encounters such a *XADLHostInstanceEvent* we either add or remove a host from the runtime architecture. As a safety measure there exist some limitations as when a host is allowed to be added and removed. Our approach does not add a host with the same unique identifier twice and only allows the removal of a host if it is no longer associated with any components, connectors, groups or sub-hosts.

Hosted Components and Connectors

The architectural properties for hosted components and connectors are extracted form the underlying runtime as well. Here our approach extracts the required information about the host and the component or connector once it is added to or removed from the runtime architecture. All this information is propagated to the aggregation process using a *XADLHostingEvent* which encapsulates multiple components, connectors, groups and sub-hosts. Our approach does currently make no use of the fact that event is capable of holding information about many components or connectors, yet we issue one event per added or removed runtime instance.

The aggregation process for *XADLHostingEvent*s is a rather simple one. If a component or connector is added we associate it with the host specified in the event. Once it is to be removed we simply remove the created association.

The creation of these associations allows us to describe a distributed architecture and integrate this description directly into the runtime architecture.

Host Properties

Each host may be associated with multiple properties which may be used to describe detailed information about the host. This architectural property is not directly extracted by our approach, we rather provide the means to integrate different extraction mechanisms dynamically into a

monitored application. The extracted properties are encapsulated by our approach into a *XADL-HostPropertyEvent* which may hold one or more properties. The event itself does not impose any restrictions on the property's type.

Our aggregation process handles three different kinds of *XADLHostPropertyEvent*s, that is the addition, update and removal of them. If properties are added or updated they are directly integrated into the runtime architecture and associated with the given host. The only difference between these two types of events is the that an update overwrites the existing properties. Our approach imposes a limitation on the name and value(s) of a property, that is they have to be serialized into simple strings. The value of a property may also be a list of such strings. Once a property should be removed we remove all properties that are contained in the event from the runtime architecture.

Our approach comes with some predefined extraction mechanisms which provide the means to extract information about the *CPU* utilization and memory usage of the host.

4.2.2 Creating Links Between Distributed Instances

Distributed applications are usually not composed of isolated instances but rather these instances are communicating with each other. Myx and xADL do not support to define such links in the design-time architecture and the associated tools do not support the creation and thus the monitoring of such external interconnections.

Our approach provides the means to monitor these connections and to integrate them into the runtime architecture of an application. To represent such external links in the runtime architecture and distinguish them from locally created links we introduce another xADL extension called *xArch External Identified Links*. It provides a new type *ExternalIdentifiedLinkInstance* which extends the *LinkInstance* type by a single attribute called *extId*. This attribute allows us to specify an identifier that represents an external connection. Listing 4.7 shows the XSD of the extension.

```
<!-- TYPE: ExternalIdentifiedLinkInstance -->
<xsd:complexType name="ExternalIdentifiedLinkInstance">
    <xsd:complexContent>
        <xsd:extension base="archinst:LinkInstance">
            <xsd:attribute name="extId" type="archinst:Identifier" />
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Listing 4.7: The xADL schema extension for externally identified links.

We impose no restrictions on the type of connection, our approach simply requires that an interface with the same type is present on both endpoints of the connections.

Because it is not possible to simply monitor these connections it is the developer's responsibility to integrate the following behavior into the component or connector that utilizes external connections. Each endpoint, be it component or connector, of an external connection needs to create an unique identifier that identifies the connection once it is established. This identifier has to be the same for both endpoints. As soon as the establishment of the connection has been finished each endpoints needs to pack the unique connection identifier alongside the runtime- and blueprint identifier of the endpoint, as well as the interface type and information if the con-

nection has been established or destroyed into a so called *XADLExternalLinkEvent* which is then sent to our aggregation process. Once the connection has been disconnected each endpoint needs to send a *XADLExternalLinkEvent* once more signaling the destruction of the connection.

If our aggregation process consumes a *XADLExternalLinkEvent* we extract the matching interface for the connection. Here we use the same approach as for locally created links. The extracted interface is then stored and associated with the external connection identifier. Once two or more events are received containing the same identifier a link is created between their matching interfaces. Here we create a new *ExternalIdentifiedLinkInstance* and add it to the architecture. It is important to note that each instance is connected to all stored instances. This logic only imposes a single restriction on the creation and monitoring of external connections: Each instance must generate the same identifier for the connection, if the identifiers do not match the link cannot be established.

Once an external link is removed from the architecture we simply remove all matching *ExternalIdentifierdLinkInstance*s that are associated with the received interface.

4.3 The Architecture Tracking Framework

After the theoretical description of our approach we take a closer look at some general implementation details of our approach. We have created a framework that is composed of two loosely coupled applications, *Monitor* and *Aggregator*. The *Monitor* is responsible for running a Myx based application, the extraction of architectural properties as well as the propagation of those properties. The *Aggregator* handles the propagated architectural properties and aggregates the resulting runtime architecture of the monitored application.

Both applications are based on the Myx architectural style, we thus use Myx based applications to monitor Myx based applications. In this section we will give a detailed description of both applications of the framework, that is we will show and describe the design-time architecture of each application.

4.3.1 Monitor

The *Monitor* is responsible for running and monitoring a Myx based application. That is it instantiates an application instance, extracts all required architectural properties and propagates them accordingly.

The architecture of the application can be seen in Figure 4.3. It is based on the architecture of the launcher application of ArchStudio 4 [12] and was extended to monitor and propagate our defined architectural properties. The architecture itself is composed of six components which will be described in the following.

Bootstrap

The *Bootstrap* component is the main entry point for running an application. It fetches the design-time architecture of the application from the *ModelRoot* component. Afterwards the *Launcher* component is instructed to launch the application specified by the design-time architecture.

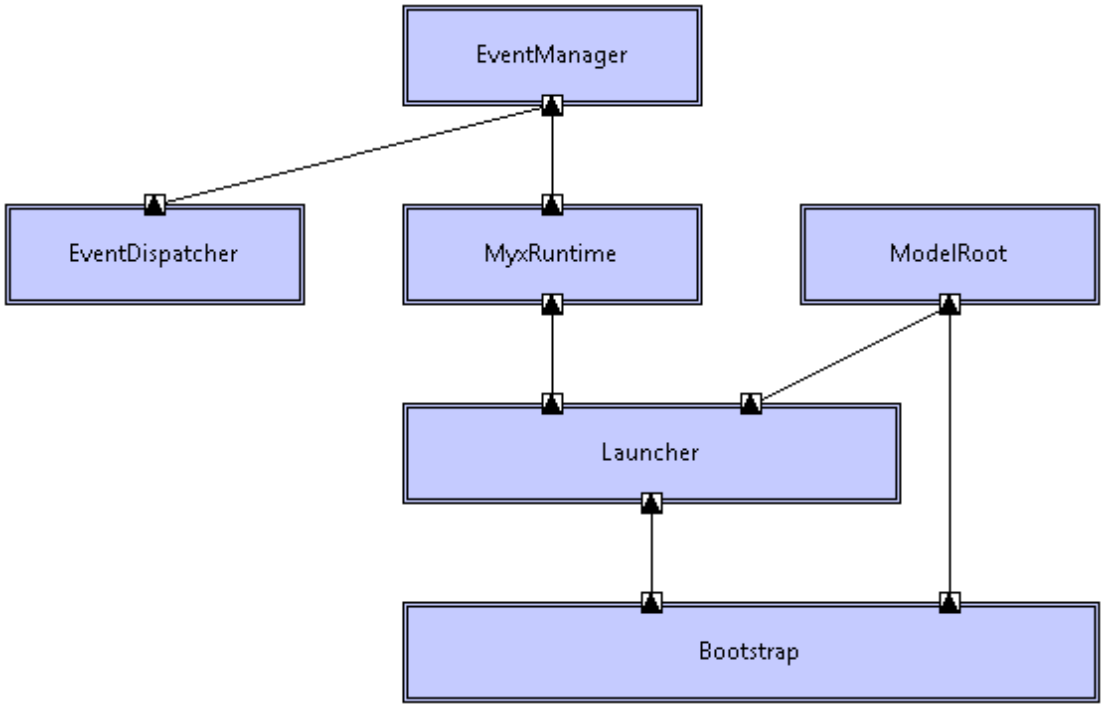


Figure 4.3: The architecture of the framework's monitor.

ModelRoot

The *ModelRoot* component represents the design-time architecture which is specified by a xADL document. To allow maximum flexibility we expose the complete architectural document by exposing the root elements of the xADL document.

Launcher

The *Launcher* component is responsible for instantiating the design-time architecture. This is done by parsing the architecture exposed by the *ModelRoot* component and finally launching the contained components and connectors using the *MyxRuntime* component.

MyxRuntime

The *MyxRuntime* component provides all capabilities to instantiate an application based on the Myx architectural style. It is also used to monitor the instantiated application by extracting some architectural properties. These properties include information about components and connectors, local links and the runtime status of components and connectors. The extracted properties are propagated using the *EventManager* component.

EventDispatcher

The *EventDispatcher* component may be used to provide extended application monitoring capabilities which mostly comprise properties about the application's environment, that is information about the host. The component allows to run multiple dispatcher instances and provides the means to propagate extracted properties via the *EventManager* component.

These capabilities allow us to monitor a host by extracting host-based architectural properties.

EventManager

The *EventManager* components handles the propagation of architectural properties to e.g. our aggregation process. The component itself is constructed in a simple way so it may be replaced easily.

4.3.2 Aggregator

The runtime architecture of an application is not directly constructed by the *Monitor* instead it propagates all extracted properties to a given endpoint. It is the *Aggregator*'s responsibility to receive these propagated architectural properties and create a runtime architecture accordingly.

We have constructed the *Aggregator* in a way that it is able to act as an endpoint for architectural properties and create the resulting runtime architecture, as well as further propagate the received architectural properties. We thus chose a publish-subscribe system [3] as the basis for our *Aggregator*. It currently supports the topic-based publish-subscribe scheme [20]. As this system is part of our motivated- and thus our evaluation scenario we are able to reuse some parts of our proof of concept implementation.

Using the publish-subscribe pattern we are able to create the runtime architecture of a local or distributed application and achieve hierarchical decomposition by allowing each connected subscriber to subscribe to only some architectural properties.

The design-time architecture of the application can be seen in Figure 4.4 and is based on architecture of our evaluation scenario shown in Figure 6.1. In the following we will describe each component and connector of the architecture to get a better understanding of the application.

PublisherDispatcher and SubscriberDispatcher

These two components are responsible for handling incoming connections. They share a common functionality with the only difference being the creation of different connectors. Each component waits for an incoming connection and instructs the underlying runtime to create a new handling instance. This instance is either a *PublisherEndpoint* or *SubscriberEndpoint*. The incoming connection is then provided to the created handling instance. This is done by an interface that allows the handling instance to poll the connection.

The creation of the handling instances is done via the *MyxRuntimeAdapter* component.

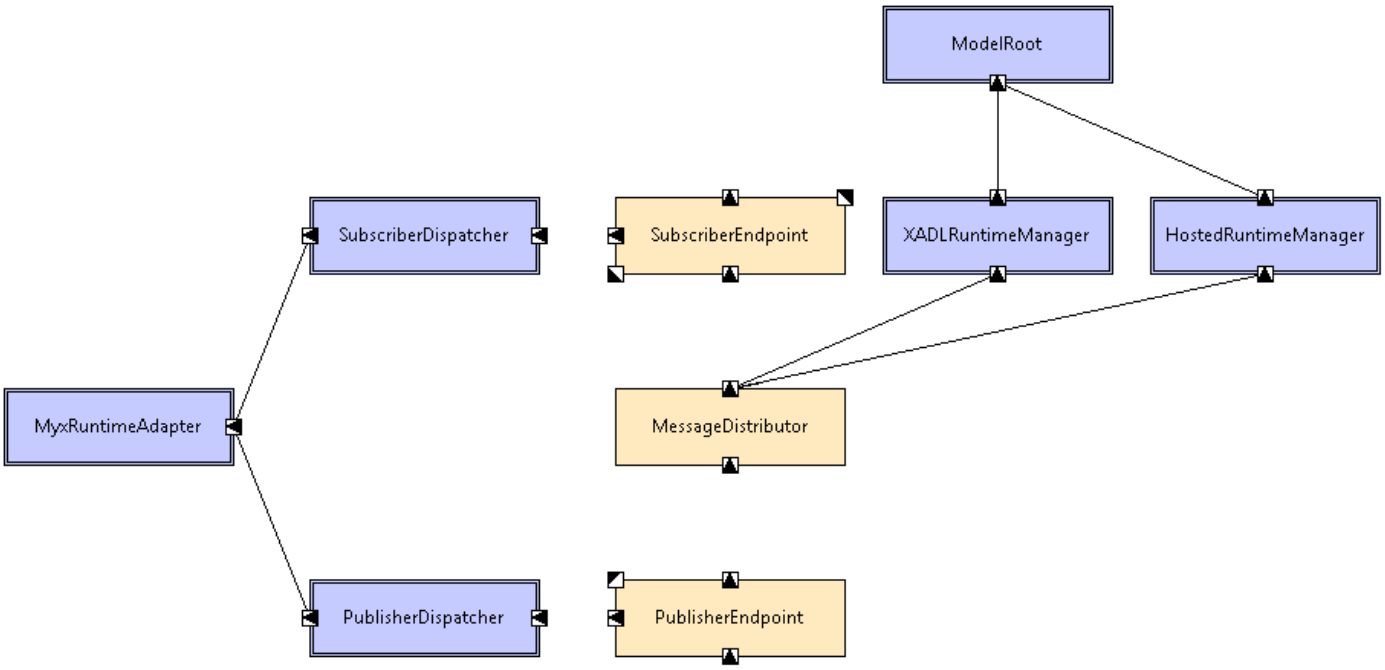


Figure 4.4: The architecture of the framework's aggegator.

MyxRuntimeAdapter

This component exposes the underlying runtime to the other components and connectors. This allows us to directly manipulate the runtime architecture of the application and dynamically create and remove specific components or connectors. This behavior is exposed via an interface that defines the allowed runtime architecture manipulation. Here it would be possible to expose the complete runtime architecture but it is encouraged to only specify the behavior that is allowed by the application.

Currently our approach uses this feature to create instances of *PublisherEndpoint* and *SubscriberEndpoint* instances.

PublisherEndpoint

This connector is one of the two dynamically created connectors in our design-time architecture. It is not instantiated while starting up the application but once a publisher connects. It fetches the incoming connection from the *PublisherDispatcher* instance and then listens for incoming events. Each event is directly forwarded to the *MessageDistributor* connector which takes care of forwarding them to the appropriate subscribers. The connector listens for events until the connection is closed, and removes itself from the runtime architecture using the *MyxRuntimeAdapter*.

ModelRoot

The *ModelRoot* component represents the design-time- and runtime architecture of the monitored application which is specified as a xADL document as well. It provides the necessary methods to read, create and update the runtime architecture as well as the means to read the design-time architecture.

SubscriberEndpoint

This connector, as the *PublisherEndpoint*, is dynamically created. It is instantiated once a subscriber connects and fetches the connection from the *SubscriberDispatcher* instance. The connector first consumes the subscribed topics from the subscriber connection. Once the topics are received it fully connects itself to the *MessageDistributor* connector using the *MyxRuntimeAdapter*, thus receiving all events from the connected publishers. As the *PublisherEndpoint* if the connection is closed it automatically removes itself from the runtime architecture.

The subscribed topics are given as a list of regular- or glob syntax expressions [39]. Based on the given topics the subscriber is able to receive all or only specific events.

The *SubscriberEndpoint* connector is not only responsible for forwarding messages to connected subscribers but is also an interface to receive information about specific components or connectors of the runtime architecture. If a subscriber receives an event about an instance that it does not yet know it can simply request information about this brick. The connector listens for such requests and queries the *ModelRoot* component for information about the requested component or connector, returning information about the instance itself and it's interfaces.

XADLRuntimeManager

The *xADLRuntimeManager* component is integrated in the architecture as a subscriber. It receives events about previously defined topics that describe all the architectural properties except ones about hosts.

Once an event with an architectural property is received we invoke the logic described in the previous two sections and atomically update the runtime architecture represented by the *ModelRoot* component.

HostedRuntimeManager

The *HostedRuntimeManager* component is also designed as a subscriber. It receives only events about host-based architectural properties and invokes the logic described in the previous section. Here we excessively use the xADL extension *xArch HostProperty* to represent the properties in the runtime architecture.

MessageDistributor

This connector is the central exchange point between the connected publishers and the subscribers. It is based on the *event pump* which is already a part of the Myx architectural style.

It implements a special kind of asynchronous notification that allow us to forward all incoming events in the order they were received whilst using features of concurrency. We are also able to store initial events that may hold meta-data and send them to each connected outgoing connection.

4.3.3 Deploying the Framework

Now that both applications of our approach have been described we are now able to create deployment strategies for different kinds of monitored applications. Here we will describe three specific deployment strategies that show the flexibility of our approach. It is important to note that our approach is not limit to these strategies due to the decoupling of both *Montitor* and *Aggregator*.

To monitor the runtime architecture of a (non-distributed) application both *Monitor* and *Aggregator* can be deployed on the same host. Figure 4.5 shows this deployment strategy.

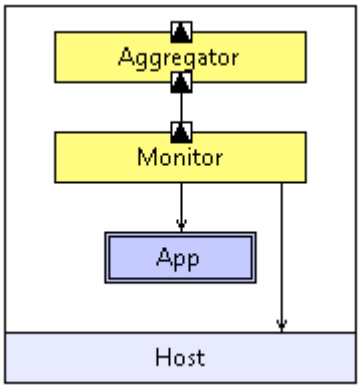


Figure 4.5: The simplest deployment scenario where both *Monitor* and *Aggregator* are running on the same host.

To monitor a distributed application and aggregate the overall runtime architecture the *Aggregator* can be decoupled from the *Monitor* and deployed as a centralized entity. Thus it is possible that multiple *Monitor* instances propagate their architectural properties to the centralized *Aggregator* which yields a holistic view of the runtime architecture of a distributed application. This deployment strategy is shown in Figure 4.6.

To reduce the amount of transmitted data to the central *Aggregator* instance it is possible to combine the two previous deployment strategies. Figure 4.7 shows an *Aggregator* instance on each host the application is running which represent a localized view of the runtime architecture. To create a holistic view of the distributed application's runtime architecture the local *Aggregator* instances are able to further propagate some (or all) received architectural properties to a central *Aggregator* instance which constructs the runtime architecture of the distributed application.

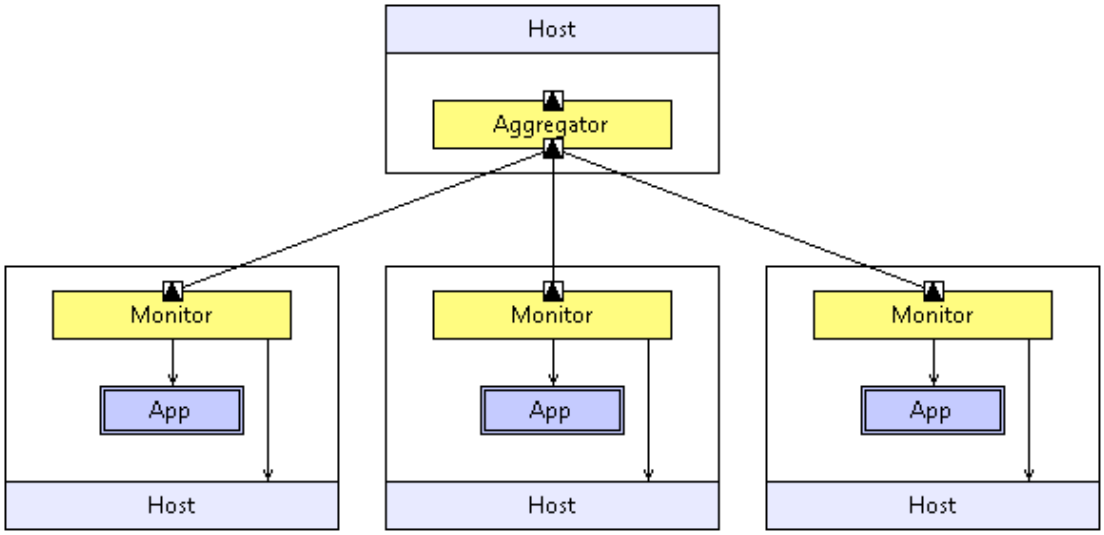


Figure 4.6: A deployment scenario for distributed applications with a centralized *Aggregator*.

Using this combined strategy we are able to only propagate certain properties to the centralized *Aggregator* instances, thus reducing the aggregation complexity and the bandwidth usage for transmitting the architectural properties.

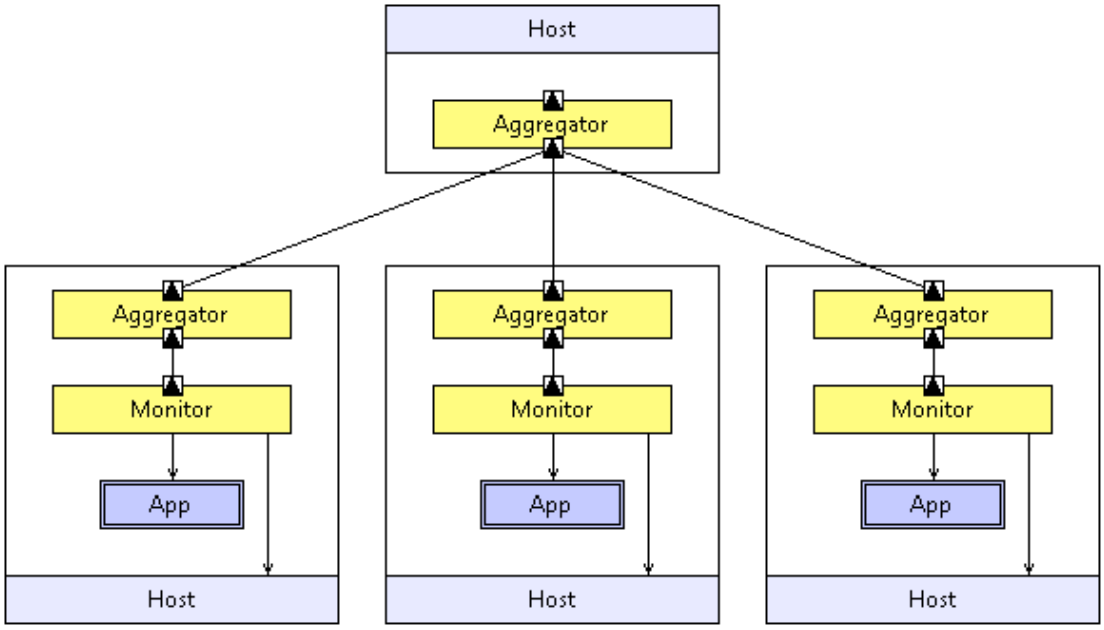


Figure 4.7: A deployment scenario for distributed applications with a local *Aggregator*, for local runtime architecture monitoring, and another centralized *Aggregator*, for distributed runtime architecture aggregation.

CHAPTER 5

Runtime Architecture Tracking: An Implementation

For an application to use our approach we have created a proof of concept implementation that is provided as a framework for architectural monitoring. It is composed of two loosely coupled applications which describe both parts of our approach, that is the *Monitor* and *Aggregator*. We will show how an application can be monitored and how the runtime architecture of an application is aggregated.

This chapter is based on the architecture description of our proof of concept implementation outlined in Section 4.3. Thus we will omit a description of the application's architecture and keep the focus on the implementation details. In the following sections we will describe the proof of concept implementation of our approach. This includes the monitoring of a Myx based application and the aggregation of an application's runtime architecture.

5.1 Implementation Specific Background

Our proof of concept implementation uses some technologies that have yet to be described. We will use this section to provide background knowledge of these technologies. Finally we will outline the technologies that have actually been used to implement our proof of concept application.

5.1.1 myx.fw

The Myx architectural style does not match a specific platform or programming language thus an architectural framework is required to bridge the gap between the concepts of an architectural style and a platform. *myx.fw* [2, 10, 36] is a framework for the Myx architectural style built in Java. It provides mechanisms to implement components and connectors and is used to instantiate them, thus yielding a running application.

In contrast to the Myx architectural pattern components and connectors are summarized under the term *brick*, which will be used to reference both component and connectors. Bricks are classes that implement an interface called *IMyxBrick*[1]. Thus the implementing class must provide two capabilities, a life-cycle provider and service objects. A life-cycle provider is a class implementing the interface *IMyxLifecycleProcessor*[2]. This interface defines four methods (*init*, *begin*, *end* and *destroy*) that are called by the framework as bricks are created, attached, detached and destroyed respectively. Service objects are implementations of provided interfaces. For each interface a brick provides it must create an object that implements the specified interface.

To provide support for runtime dynamism the framework provides an extended version if the *IMyxBrick* interface called *IMyxDynamicBrick*[3]. It requires a brick to implement callback methods for notifications once an interface has been connected or disconnected.

To simplify the process of creating bricks the framework provides an abstract class called *AbstractMyxSimpleBrick*[4] that already implements most of the methods of *IMyxBrick*. The only method an extended class is required to implement is *getServiceObject*. This method allows to specify the true implementation of each provided interface. It is up to the implementing class which life-cycle methods are needed and thus must be implemented.

To show how a brick is implemented using the means of *myx.fw* Listing 5.1 shows a brick that is used in our proof of concept implementation. It outlines how provided interfaces are returned using the *getServiceObject* method where the name of the interface is compared as well as how objects behind required interfaces are acquired by using the *MyxUtils*[5] class in the method *begin*.

This class also shows how applications may achieve concurrency by using the method *init* to initialize the logic, the method *begin* to begin the concurrent execution and the method *end* for shutdown.

To create and manage Myx applications the framework provides an interface called *IMyxRuntime*[6]. This interface provides methods to manipulate the runtime architecture. That is it allows us to add or remove bricks and it's associated interfaces and create links between bricks. There exists a default implementation that may be accessed by a caller outside the *myx.fw* (e.g. a *main* method). This default implementation is used by *myx.fw* itself but also by ArchStudio to instantiate a Myx based application. The design of the class allows for an easy extension of it and thus enables us to extend the default functionality.

[1] `edu.uci.isr.myx.fw.IMyxBrick`
[2] `edu.uci.isr.myx.fw.IMyxLifecycleProcessor`
[3] `edu.uci.isr.myx.fw.IMyxDynamicBrick`
[4] `edu.uci.isr.myx.fw.AbstractMyxSimpleBrick`
[5] `edu.uci.isr.myx.fw.MyxUtils`
[6] `edu.uci.isr.myx.fw.IMyxRuntime`

```
package at.ac.tuwien.dsg.pubsub.middleware.comp;

[...]

public abstract class Dispatcher<E> extends AbstractMyxSimpleBrick implements
    IDispatcher<E> {

    [...]

    public static final IMyxName IN_IDISPATCHER = MyxInterfaceNames.IDISPATCHER;
    public static final IMyxName OUT_MYX_ADAPTER = MyxInterfaceNames.IMYX_ADAPTER;

    private ExecutorService executor;
    private Runnable runnable;

    protected IMyxRuntimeAdapter myxAdapter;

    [...]

    @Override
    public Object getServiceObject(IMyxName interfaceName) {
        if (interfaceName.equals(IN_IDISPATCHER)) {
            return this;
        }
        return null;
    }

    @Override
    public void init() {
        executor = Executors.newSingleThreadExecutor();
        runnable = new Runnable() {
            public void run() {
                [...]
            }
        };
    }

    @Override
    public void begin() {
        myxAdapter = MyxUtils.<IMyxRuntimeAdapter> getFirstRequiredServiceObject(this,
            OUT_MYX_ADAPTER);
        executor.execute(runnable);
    }

    @Override
    public void end() {
        executor.shutdownNow();
    }

    [...]

}
```

Listing 5.1: A simplified implementation of a brick which is used in our proof of concept implementation. Here most of the *IMyxLifecycleProcessor* methods are used to control the behavior of the brick.

5.1.2 xADL Tool Support

xADL provides a set of libraries, that are implemented in Java, which provide an Application Programming Interface (API) to access xADL documents. These libraries combined are called Data Binding Library (DBL) [14] and provide interfaces and objects corresponding to the elements defined in a xADL schema. Each XML type defined in a xADL schema has an interface and a corresponding class implementing it. Listing 5.2 shows an excerpt of the interface *IArchStructure*[7] and shows how a xADL schema can be accessed.

xADL further provides a tool called *apigen* [13, 15] which can be used to generate Java code for a given set of xADL schemes. This tool is not specific to xADL but rather can be used to generate Java code to access all kind of XML files that conform to a given XSD. This tool, however, has some limitations as it only supports a subset of XSD. Thus it is best used to generate Java code for xADL and extensions of it.

```
public interface IArchStructure extends edu.uci.isr.xarch.IXArchElement{
    [...]
    /**
    * Gets the component from this ArchStructure with the given
    * id.
    * @param id ID to look for.
    * @return component with the given ID, or <code>null</code> if not found
    */
    public IComponent getComponent(String id);
    [...]
    /**
    * Gets the connector from this ArchStructure with the given
    * id.
    * @param id ID to look for.
    * @return connector with the given ID, or <code>null</code> if not found
    */
    public IConnector getConnector(String id);
    [...]
}
```

Listing 5.2: Excerpt of the interface *IArchStructure*

5.1.3 Chosen Technologies

We have chosen Java 7, ArchStudio 4 [12], the included *myx.fw* 4.1.50 and XADL 2.0 for our proof of concept implementation. ArchStudio [45] provides methods to create and run an application by it's architecture description. Due to it's integration into the Eclipse[8] platform it was used to design the architecture descriptions by the Myx architectural style in xADL. Eclipse was later used to implement our framework.

These technologies provide many advantages, most of them were already outlined in the previous sections. The advantage this theses benefited most is the flawless interaction of these technologies with each other.

With all the advantages these technologies provide they also come with some drawbacks and limitations:

[7] `edu.uci.isr.xarch.types.IArchStructure`
[8] `http://www.eclipse.org/`

- The editors included in ArchStudio do not handle xADL extensions very well, this concludes in unexpected errors while viewing and manipulating architecture descriptions.
- The DBL was originally written to be compatible with Java 5, thus it does not include modern language features such as generics.
- While *myx.fw* provides an API to manipulate an application it does not handle the dynamic creation of bricks very well. To connect newly created components or connectors to other specific bricks, the implementation needs to know the name of those which results in specifying architectural properties directly in the source code.

5.2 Monitoring an application's runtime architecture

The monitoring of an application, be it distributed or not, is done by our *Monitor* application. This application acts as a bootstrap for applications based on the Myx architectural pattern. The application is based on the architecture outlined in Section 4.3.1. The architecture and it's implementation is based on the source of ArchStudio 4 [12]. Mostly all of the code has been rewritten by using the DBL and our custom *myx.fw* extensions.

The monitoring of an application is composed of three steps:

1. Application Instantiation
2. Extraction of architectural properties
3. Propagation of architectural properties

In the following we will describe how our proof of concept implementation implements each of these steps.

5.2.1 Instantiating an Application

The first step for monitoring an application is it's startup also called instantiation. Our frameworks implements the logic to instantiate a Myx based application specified by a xADL document using *mxy.fw*. This instantiation process uses the xADL DBL to read the architecture description and *myx.fw* to instantiate the application. This process is based on an existing one which is used by ArchStudio 4. We have reimplemented the instantiation process and introduced our monitoring capabilities. In the following we will describe our customized process which is shown in Algorithm 5.1.

The first step is to read and parse the architectural description using the DBL, see Line 2, that is the types and structures inside the xADL document. We extract all bricks that are required for the instantiation of an application. Each brick is validated, including it's interfaces, and brought into a custom representation which allows for easy access to the properties needed by the instantiation process. It is important to note that we extract the bricks in a specific order that is given by analyzing the dependencies (i.e. links) of each brick.

input: An *IArchStructure* instance s

```
1  create base container;
2  bricks ← parse architecture of s;
3  foreach brick b in bricks do
4      if b has a sub-architecture then
5          call instantiate for sub-architecture of b;
6          map container interfaces of b;
7      end
8      else if b has implementation then
9          extend init properties of brick b;
10         add brick b;
11         extend properties of all interfaces of b;
12         add interfaces of b;
13         weld init links of b;
14         call init life-cycle method of b;
15     end
16 end
17 foreach brick b in bricks do
18     weld begin links of b;
19     call begin life-cycle method of b;
20 end
```

Algorithm 5.1: The algorithm for our instantiation process to instantiate a xADL structure.

In the next step we evaluate for each brick if it contains a sub-architecture which has to be handled specially. Due to the fact that a sub-architecture is represented as another structure in a xADL document we are able to call the algorithm once more, see Line 5. After the sub-architecture has been instantiated we have to map the interfaces of the brick to interfaces in the sub-architecture.

Once an atomic brick is encountered we create an instance of it using *myx.fw*, that is we use an instance of *IMyxRuntime* to add the brick to the runtime architecture, see Line 10. For our monitoring solution to work we have to extend the *init* properties of the brick and inject the blueprint identifier and it's type (component or connector) before it is added to the runtime architecture, see Line 9. The next step is to add all interfaces of the brick. Once again we have to extend the properties of each interface for our monitoring solution to work and inject the interface's type, see Line 11. The interface is then added to the brick using the manipulated properties. After the brick has been fully added to the runtime architecture it's *init* links are welded, i.e. created. Here we only create links which contain interfaces whose service type is given, that is they are defined as *required* or *provided*. As a last step we call the life-cycle method *init* of the brick.

To fully instantiate the architecture we have to process all bricks one more time, see Lines 17

to 20. Here we add the remaining links of each brick, named *begin* links, thus ensuring that all bricks are connected as described in the architectural description. The final step is to call the life-cycle method *begin* of the brick.

5.2.2 Extracting Architectural Properties

The next step in our monitoring process is the extraction of architectural properties. Here we use different means for extraction, that is we are able to extract most of the properties automatically but there are some that can only be extracted by the developer. In the following we will describe how we have implemented the extraction of each architectural property that is used by our approach.

Using myx.fw

The central entity of any application instantiated using *myx.fw* is an instance of the interface *IMyxRuntime*. The exposed methods of this interface allow us to add or remove bricks, interfaces and links. Due to the architecture of the *myx.fw* and the central characteristic of the interface *IMyxRuntime* we have chosen to use it to intercept certain method calls and extract some of our architectural properties. As the interface *IMyxRuntime* already has an implementation called *MyxBasicRuntime*[9] contained in *myx.fw* we have chosen to extend this class and override certain methods. The method signatures of the overridden methods can be seen in Listing 5.3. As *myx.fw* does not differentiate between components and connectors as well as it does not know of interface types we have to rely on our instantiation algorithm to inject this information. This behavior has been described in detail in the previous section.

We have created a class called *MyxMonitoringRuntime*[10] which extends *MyxBasicRuntime* and overrides the listed methods. This class is used to extract most of our architectural properties. In the following we will describe how it's overridden methods are used for extraction.

```
public interface IMyxRuntime {
    [...]
    public void addBrick(IMyxName[] path, IMyxName brickName, IMyxBrickDescription
        brickDescription) throws MyxBrickLoadException, MyxBrickCreationException;
    public void removeBrick(IMyxName[] path, IMyxName brickName);
    [...]
    public void addInterface(IMyxName[] path, IMyxName brickName, IMyxName
        interfaceName, IMyxInterfaceDescription interfaceDescription,
        EMyxInterfaceDirection interfaceDirection);
    public void removeInterface(IMyxName[] path, IMyxName brickName, IMyxName
        interfaceName);
    [...]
    public IMyxWeld createWeld(IMyxName[] requiredPath, IMyxName requiredBrickName,
        IMyxName requiredInterfaceName, IMyxName[] providedPath, IMyxName
        providedBrickName, IMyxName providedInterfaceName);
    public void addWeld(IMyxWeld weld);
    public void removeWeld(IMyxWeld weld);
    [...]
    public void begin(IMyxName[] path, IMyxName brickName);
    public void end(IMyxName[] path, IMyxName brickName);
```

[9] edu.uci.isr.myx.fw.MyxBasicRuntime

[10] at.ac.tuwien.dsg.myx.monitor.MyxMonitoringRuntime

```
    [...]
}
```

Listing 5.3: Method signatures of *IMyxRuntime* that are used by our proof of concept implementation.

Components and Connectors To extract information about bricks we use the methods *addBrick* and *removeBrick*. As their names suggest, these methods target a brick's creation and removal in the runtime architecture. The creation of a bricks is done by the method *addBrick*. By simply overloading it we cannot extract all information needed to monitor this architectural property. We only get the bricks runtime identifier, aka it's name, but we do not know it's blueprint identifier or if it is a component or connector. Our approach uses the instantiation process to inject additional information about a brick using the properties defined by the bricks description. By maintaining a mapping of runtime identifiers to their associated blueprint identifiers we are able handle the removal of bricks by overloading the method *removeBrick* as well. The described mapping provides us with the means to extract all required information for the removal of a brick, i.e. it's blueprint identifier as well as the type of brick. As we extract information about created or removed brick using these methods we are able to use them to associate the extracted component or connector with the host it is running.

Local Links Our monitoring approach for links, be it local or external, is based on the interfaces associated with a link. Thus we use the methods *addInterface* and *removeInterface* to create a mapping of the interfaces runtime identifier, i.e. the interface's name, and it's type and associate this mapping with the interface's brick. Here we also used our instantiation process to inject the interface's type into the description of the interface. Therefore we had to extend the interface *IMyxInterfaceDescription*[11] for the ability to inject properties into the method *addInterface*. This new interface is called *IMyxInitPropertiesInterfaceDescription*[12] and provides a single method to get associated properties. Our instantiation process uses an implementation of this interface to inject the interface's type. Information about the creation and removal of a local link is extracted by overloading the methods *addWeld* and *removeWeld*. Both of these methods are given an instance of the interface *IMyxWeld*[13] as an argument which represents the added or removed link. This interface contains nearly all required information about links, that is we are able to access the brick's- and the interface's runtime identifiers of both link endpoints. By accessing the previously created mapping to extract the types of both endpoint interfaces we are able to extract all information required for the architectural property of a local link, including it's creation and removal.

The Runtime Status The runtime status of a brick can be determined by the calls to it's lifecycle methods. Thus our approach overrides the methods *begin* and *end*. We assume that the given brick is in a running state once the *begin* method is called. This state is maintained until

[11] `edu.uci.isr.myx.fw.IMyxInterfaceDescription`
[12] `at.ac.tuwien.dsg.myx.fw.IMyxInitPropertiesInterfaceDescription`
[13] `edu.uci.isr.myx.fw.IMyxWeld`

the *end* method is called or the brick is removed from the runtime architecture. Each life-cycle method may be called for a single brick or all bricks inside of a container, thus allowing us to extract the runtime status of multiple bricks.

General Information about a Host Our custom *IMyxRuntime* implementation extracts the name of the application's host at the beginning of the instantiation process and the corresponding event is emitted. Whilst shutting down the application in a last step we once again emit an event marking the end of the application.

Outside Monitoring

Using *myx.fw* to extract our architectural properties does not yield a solution for all of our described architectural properties, thus we have to rely on other means to extract them. This mostly involves the developer of an application who has to integrate functionality which is specific to our solution. In the following we will describe how the remaining architectural properties are to be extracted and how our approach supports the developer on completing these tasks.

External Links Links between bricks to external services, be it bricks in other application instances or different communication endpoints, cannot be monitored by *myx.fw*. Information about these external connections have to be extracted directly from the brick which utilizes the connection. Thus it is the developers responsibility to correctly monitor these links. Here we support the developer by providing a class called *AbstractMyxExternalConnectionBrick*[14] which eases the propagation of this architectural property. In Section 5.5 we will show how an application may integrate this class to monitor an external link.

Host-Based Properties As some of the host-based properties can be extracted using *myx.fw*, see above, it is not possible to extract information about the host itself. We have created the possibility to integrate such extended monitoring capabilities next to the application itself. The *EventDispatcher* component in the architecture of our *Monitor* application is responsible for running extended monitoring capabilities next to the actual application. Here we use an interface called *EventDispatcher*[15] which is an extension to the *Runnable*[16] interface. This allows us to execute instances of our interface in their own threads. The provided method *run* should be used to execute the monitoring capabilities. Here we impose no limitation on the tools used inside this method and the extracted information. We rather provide an abstract class called *AbstractEventDispatcher*[17] which eases the propagation of host-based properties. For our proof of concept implementation we have created two such *EventDispatcher* implementations which extract information about the hosts Computational Processing Unit (CPU) utilization and it's memory usage using Java features. Here it would also be possible to execute Operating System (OS)-specific tools and programs to extract different kinds of information, e.g. the hosts Internet Protocol (IP)-addresses or the host-name.

[14] `at.ac.tuwien.dsg.myx.monitor.AbstractMyxExternalConnectionBrick`
[15] `at.ac.tuwien.dsg.myx.monitor.ed.EventDispatcher`
[16] `java.lang.Runnable`
[17] `at.ac.tuwien.dsg.myx.monitor.ed.AbstractEventDispatcher`

5.2.3 Propagation of Architectural Properties

The propagation of architectural properties is the process of transporting information from our *Monitor* application to our *Aggregator*. The *EventManager* component in our *Monitor* application exposes an interface named *EventManager*[18] which provides a single method called *handle*. This method takes an instance of the abstract class *Event*[19] and is responsible for handling the given event by e.g. forwarding it to the *Aggregator*. The *Event* class and it's child classes are based on the events defined in Section 4.1 and 4.2 and are outlined in Figure 5.1. Due to the structure and properties of the defined events we are able to directly create a class structure.

Our base event class contains four properties: *id*, *timestamp*, *archtiectureRuntimeId* and *eventSourceId*. The *id* is a (pseudo-)randomly generated identifier, namely an UUID v4 [41], which uniquely identifies the event. The *time stamp* contains the unix-timestamp which refers to the time the event was created. The *architectureRuntimeId* is and identifier that marks the current application, see Section 5.5 for further details. Finally the *eventSourceId* defines the source of the event, that is the location the event was created. In our proof of concept implementation this refers to the creating class. The full definition of our base class can be found in the appendix.

All other events encapsulate all the information required for the architectural property.

[18] `at.ac.tuwien.dsg.myx.monitor.em.EventManager`
[19] `at.ac.tuwien.dsg.myx.monitor.em.events.Event`

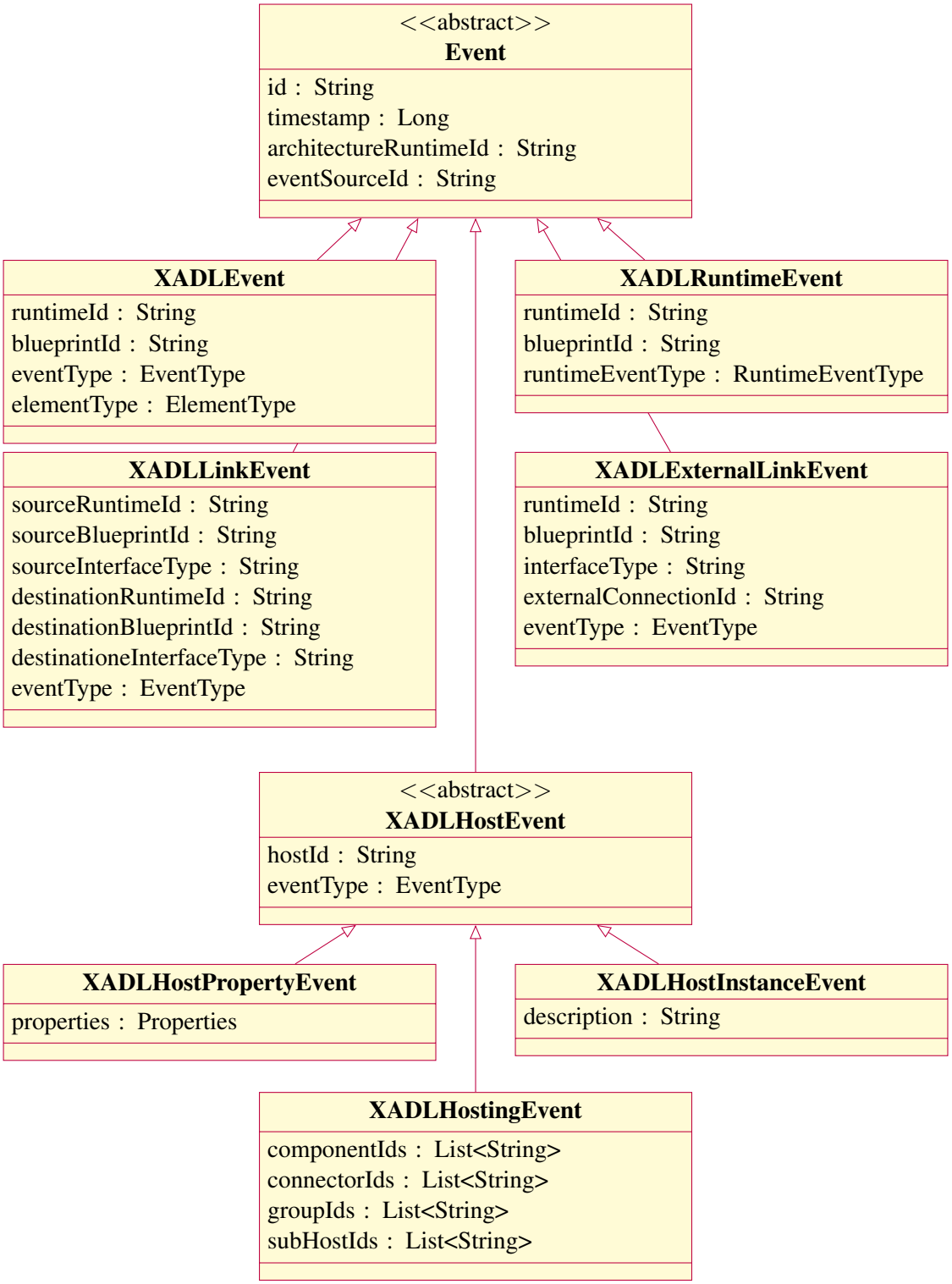


Figure 5.1: Class hierarchy for events containing architectural properties as an UML class diagram.

All of the listed event classes are used throughout our *Monitor* and *Aggregator* applications. The *Monitor* application creates the events, either automatically or by providing the means so allow the developer to do so, and emits them using the *EventManager* which is responsible for transporting the events to the *Aggregator*. The *Aggregator* listens for such incoming events and processes them to create the resulting runtime architecture.

As the event classes are used to describe architectural properties they can be associated with them, thus we create them whilst extracting all of our architectural properties. The extracted properties are directly forwarded to the *EventManager*. The default implementation of the *EventManager* simply drops all incoming events and should not be used for application monitoring. We provide an alternative implementation which can be used to communicate with the *Aggregator* component. This implementation also injects some properties into each event. That is the *architectureRuntimeId* and *hostId*, for *XADLHostEvent*s. After injecting these properties the event is associated with a topic and sent to the *Aggregator*. The topic is used in the publish-subscribe system of our *Aggregator*. Table 5.1 shows the event classes, their associated architectural property and topic.

The topic is specified as a simple string and identifies a certain kind of event. To allow for extended matching which is not only based on the type of event but on other properties as well our approach adds the blueprint- or host identifier to the shown base topic. For the events *XADLEvent*, *XADLRuntimeEvent* and *XADLExternalLinkEvent* the blueprint identifier is appended to the topic so it results in the string <*event-identifier*>.<*blueprint-identifier*>, e.g. *event.xadl.componentffa80065-dd76782b-71c4b832-18f7133c*. For *XADLLinkEvents* the blueprint identifiers of both source- and destination brick are appended to the topic resulting in <*event-identifier*>.<*source-blueprint-identifier*>.<*destination-blueprint-identifier*>, e.g. *event.xadlexternallink.componentffa80065-dd76782b-71c4b832-18f7133c.componentffa80065-dd76b5d8-c36ff1da-18f71360*. For *XADLHostEvent* and it's children the host identifier is appended to the event identifier which results in the string <*host-event-identifier*>.<*host-identifier*>, e.g. *event.xadlhost.property.fff20c0e-f569c1a7-b867d7fe-a26e08fd*. Using these topics allows for specialized subscriptions in our *Aggregator*.

Event Class	**Architectural Property**	**Topic**
Event		event
XADLEvent	Components & Connectors	event.xadl
XADLRuntimeEvent	The Runtime Status	event.xadlruntime
XADLLinkEvent	Local Links	event.xadllink
XADLExternalLinkEvent	External Links	event.xadlexternallink
XADLHostEvent		event.xadlhost
XADLHostingEvent	Hosted Components & Connectors	event.xadlhost.hosting
XADLHostInstanceEvent	Hosts	event.xadlhost.instance
XADLHostPropertyEvent	Host Properties	event.xadlhost.property

Table 5.1: Event classes, their architectural properties and topics.

As some of our architectural properties cannot be extracted automatically and require a developer we still describe features of our proof of concept implementation which ease the propagation

process of these architectural properties.

External Links

For our approach to be able to monitor externally created links the application has to emit a *XADLExternalLinkEvent*. We support the developer by providing an abstract class called *AbstractMyxExternalConnectionBrick*[20] which can be used as the basis of a brick utilizing external connections. This class is an extension of the *AbstractMyxSimpleBrick* class and provides two methods for the dispatching of *XADLExternalLinkEvent*s, see Listing 5.4.

```
package at.ac.tuwien.dsg.myx.monitor;

[...]

public abstract class AbstractMyxExternalConnectionBrick extends AbstractMyxSimpleBrick
     {
    [...]
    protected void dispatchExternalLinkConnectedEvent(String interfaceType, String
        externalConnectionIdentifier) { [...] }
    protected void dispatchExternalLinkDisconnectedEvent(String interfaceType, String
        externalConnectionIdentifier) { [...] }
    [...]
}
```

Listing 5.4: The two exposed methods to propagate external link events.

These two methods indicate either that an external connection as been established or shut down. They both require the interface's type and connection identifier to be present. Once the brick has established an external connection it should call the *dispatchExternalLinkConnectedEvent* method with the interface's type and the corresponding connection identifier. After the connection's shutdown the method *dispatchExternalLinkDisconnectedEvent* needs to be called. Listing 5.5 shows how we have used the provided methods in our evaluation application to propagate events about an external connection being created and shut down. It is important to note that the interface type has to be known by the implementation. It is currently not possible to inject this type dynamically so the interface's type has to be defined directly in the source code.

[20] `at.ac.tuwien.dsg.myx.monitor.AbstractMyxExternalConnectionBrick`

```
package at.ac.tuwien.dsg.pubsub.subscriber.comp;

[...]

public abstract class Subscriber<E> extends AbstractMyxExternalConnectionBrick {

    [...]

    private ExecutorService executor;
    private Runnable runnable;

    [...]

    @Override
    public void init() {

        [...]

        executor = Executors.newSingleThreadExecutor();
        runnable = new Runnable() {
            @Override
            public void run() {
                [...]
                String connectionIdentifier = getExternalConnectionIdentifier();
                dispatchExternalLinkConnectedEvent("interfaceType0bcf68ee-6bf6-488c-
                    af3f-105447849d8e", connectionIdentifier);
                [...]
                dispatchExternalLinkDisconnectedEvent("interfaceType0bcf68ee-6bf6-488c-
                    af3f-105447849d8e", connectionIdentifier);
                [...]
            }
        };
    }

    @Override
    public void begin() {
        [...]
        executor.execute(runnable);
    }

    [...]

    /**
     * Get the external connection id of the connected {@link Endpoint}.
     *
     * @return
     */
    protected abstract String getExternalConnectionIdentifier();
}
```

Listing 5.5: Example usage of the *AbstractMyxExternalConnectionBrick* class by a componentn of our subscriber evaluation application.

The tricky part in handling external connections is the generation of the connection identifier which has to be the same for all interfaces involved in the external connection. Listing 5.6 shows how a connection identifier can be created for connections involving sockets.

```
package at.ac.tuwien.dsg.pubsub.subscriber.comp.socket;

[...]

public class SocketByteSubscriber extends Subscriber<byte[]> {
    [...]

    @Override
    protected String getExternalConnectionIdentifier() {
        Socket s = [...];
        // from,to
        return s.getLocalAddress().getHostAddress() + ":" + s.getLocalPort() + "," + s.
            getInetAddress().getHostAddress() + ":" + s.getPort();
    }

    [...]
}
```

Listing 5.6: Generating a connection identifier for a socket based connection.

Host-Based Properties

The extraction of host-based properties is not directly handled in our *Monitor* application but can be run next to the actual application. Our approach uses the interface *EventDispatcher* to execute such monitoring utilities. To support the developer in propagating the extracted properties we have created an abstract class called *AbstractEventDispatcher* which provides two methods which ease the creating of *XADLHostPropertyEvent*s and their propagation. Listing 5.7 shows the method signatures of these two methods. The method *initHostPropertyEvent* creates a fully configured instance of *XADLHostPropertyEvent* and the method *dispatch* forwards the event to the *EventManager* which further propagates it.

As an example implementation we have provided our approach to monitor the CPU utilization of a host. Listing 5.8 shows the class and how we have created a continuous monitoring utility.

```
package at.ac.tuwien.dsg.myx.monitor.ed;

[...]

public abstract class AbstractEventDispatcher implements EventDispatcher {
    [...]
    /**
     * Dispatch a {@link Event} to the {@link EventManager}.
     * @param event
     */
    final protected void dispatch(Event event) { [...] }

    /**
     * Initialize a {@link XADLHostPropertyEvent}.
     * @return
     */
    protected XADLHostPropertyEvent initHostPropertyEvent() { [...] }
}
```

Listing 5.7: The two exposed methods to handle host property events.

```
package at.ac.tuwien.dsg.myx.monitor.ed.dispatchers;

[...]

@SuppressWarnings("restriction")
public class CPUMonitor extends AbstractEventDispatcher {
    [...]
    @Override
    public void run() {
        OperatingSystemMXBean osb = (OperatingSystemMXBean) ManagementFactory.
            getOperatingSystemMXBean();
        while (true) {
            // create event and set properties
            XADLHostPropertyEvent cpuLoadEvent = initHostPropertyEvent();
            if (osb.getSystemCpuLoad() >= 0) {
                cpuLoadEvent.getHostProperties().put(XADLHostProperties.CPU_SYSTEM_LOAD
                    , osb.getSystemCpuLoad());
            }
            if (osb.getProcessCpuLoad() >= 0) {
                cpuLoadEvent.getHostProperties().put(XADLHostProperties.
                    CPU_PROCESS_LOAD, osb.getProcessCpuLoad());
            }
            // dispatch the event
            if (!cpuLoadEvent.getHostProperties().isEmpty()) {
                dispatch(cpuLoadEvent);
            }
            // sleep for some time
            try {
                Thread.sleep(5 * 1000);
            } catch (InterruptedException e) {
                return;
            }
        }
    }
}
```

Listing 5.8: Our example implementation to extract the CPU utilization of the current host.

5.3 Aggregating a runtime architecture

Our proof of concept implementation of the *Aggregator* is, as the *Monitor* run using *myx.fw* with it's architecture outlined in Section 4.3.2. It is based on the message broker of our evaluation application which is described in Section 6.1. Due to this fact we will skip most of the details of our publish-subscribe implementation and focus on the aggregation of the runtime architecture.

The *Aggregator* is designed as a loosely coupled application that can be run with or without a *Monitor* instance. It acts as an endpoint for receiving events about architectural properties and is responsible for creating the resulting runtime architecture. Due to the nature of a publish-subscribe system we are not limited to creating the runtime architecture but are also able to directly act as a message broker and forward the received architectural properties.

In the following we will describe how the different events are handled throughout the *Aggregator*, how the runtime architecture is created and how we used the underlying publish-subscribe system to allow for further property propagation.

5.3.1 Handling of Architectural Events

The *Aggregator* uses the underlying publish-subscribe system to receive architectural properties. Each connected *Monitor* instance is handled by an instance of the *PublisherEndpoint* component which forwards the received events to the *MessageDistributor*.

There exist two components in the architecture of the *Aggregator* that are responsible for handling the incoming architectural events to construct the runtime architecture of an application. The *XADLRuntimeManager* and *HostedRuntimeManager* are designed and integrated as simple subscribers, thus directly receiving all incoming events. They both filter the received events by predefined topic which allow them to only receive certain events.

They both are able to read and write the persisted design- and runtime architecture using the *ModelRoot* component. This component exposes these architectures using the DBL. That is it exposes the design-time architecture by instances of *IArchStructure*. The runtime architecture is exposed by instances of *IArchInstance*.

The *Aggregator* is able to construct the runtime architecture of multiple applications at the same time, under the assumption that they use the same design-time architecture so that all elements are known. This behavior is achieved using the *architectureRuntimeId* property that is contained in each event. We create a runtime architecture representation, namely an instance of *IArchInstance*, for each received identifier and populate it using the associated events.

Our approach allows to aggregate a distributed application using a single or multiple *architecutreRuntimeId*s. For a logic grouping it is encouraged that all application instances in a distributed application use the same *architectureRuntimeId* so that all architectural elements are contained inside the same *IArchInstance* instance.

XADLRuntimeManager

The *XADLRuntimeManager* is responsible for handling all non-host-based architectural events, that is *XADLEvent*s, *XADLRuntimeEvent*s, *XADLLinkEvent*s and *XADLExternalLinkEvent*s. The

events are filtered by using patterns that match these classes. Each event is handled differently but accordingly to the specification that was described in Section 4.1 and 4.2. Here we excessively use our introduced xADL extensions.

Whilst processing one of the above events we follow a defined structure. At first an *IArchInstance* instance is extracted from the existing runtime architecture or a new instance is created. The changes described by the event are then done in this specific instance. It is possible to persist the runtime architecture after each processed architectural event we encourage the periodic persit to outsource and persist it periodically.

HostedRuntimeManager

The *HostedRuntimeManager* handles all host-based architectural events, that is all subclasses of *XADLHostEvent*. Like the *XADLRuntimeManager* we use predefines topics to only receive events containing host-based architectural properties.

The host-based properties are persisted in the runtime architecture using the xADL *xArch HostProperty* extension. As this extension provides an extension to the *IArchInstance* we have to promote the extracted *IArchInstance* to a *IHostedArchInstance*[21] instance using methods of the DBL.

Once an instance of *IHostedArchInstance* is created we are able to process the received events and update the runtime architecture as specified in Section 4.2.

5.3.2 Using the Publish-Subscribe System

By using a publish-subscribe system as the base of our *Aggregator* allows us to use it as a message broker and thus allow received architectural events to be further propagated. That is we allow subscribers to connect to the *Aggregator* as well which are handled by instances of the component *SubscriberEndpoint*. This component waits for the subscriber to publish it's subscribed topics and then connects itself to the *MessageDistributor* and *ModelRoot*. By connecting to the *MessageDistributor* we receive all incoming events and match their topic against the subscribed ones. If the topic matches the event is forwarded to the subscriber.

By providing different topics we are able to only subscribe to certain events, bricks or hosts. Our approach provides three means to specify a topic: by a simple string, a regular expression or by a glob pattern [39]. By using the topics for each event we are able to subscribe only to certain events. E.g. by specifying the glob pattern *event.xadl.** we only subscribe to *XADLEvent*s. By using the glob pattern **.componentffa80065-dd76782b-71c4b832-18f7133c* we are able to receive only events associated with the component identified by the blueprint identifier *componentffa80065-dd76782b-71c4b832-18f7133c*. By providing multiple such topics it is possible to receive only certain events or events about certain bricks or hosts.

For subscriber that have connected after most of the instantiation events have been propagated and thus the subscriber does not have all required information, we provide the means for

[21] `edu.uci.isr.xarch.hostproperty.IHostedArchInstance`

a subscriber to request further information about a brick. That is the *SubscriberEndpoint* implementation listens for incoming requests about certain bricks and extracts this information from the runtime architecture using the *ModelRoot* component. Here we currently support general information about the brick, that is we return the bricks description and all it's interfaces. The interfaces are specified by their runtime- and type identifier. By allowing subscribers to request additional information about bricks gives them the ability to construct the runtime architecture of an application even if some events have not been delivered to them.

5.4 Communication

After both applications of our framework have described we will visualize the event propagation process in more detail and present the communication between *Monitor* and *Aggregator*. Therefore we have created a number of diagrams which show the event propagation process for each application and between both of them.

The *Monitor*'s event propagation centers around the *EventManager* component which is responsible for forwarding events to the *Aggregator*. Most of the architectural properties are extracted and the corresponding events are propagated whilst instantiating or shutting down an application. Figure 5.2 shows a sequence diagram where the instantiation of an application is shown. We have sectioned the sequence diagram into the different stages of the instantiation process. This demonstrates when events are propagated by the *MyxRuntime* component and by a brick itself.

We also show how the *EventDispatcher* is running next to the actual application and is able to propagate events at any time during the instantiation process or afterwards.

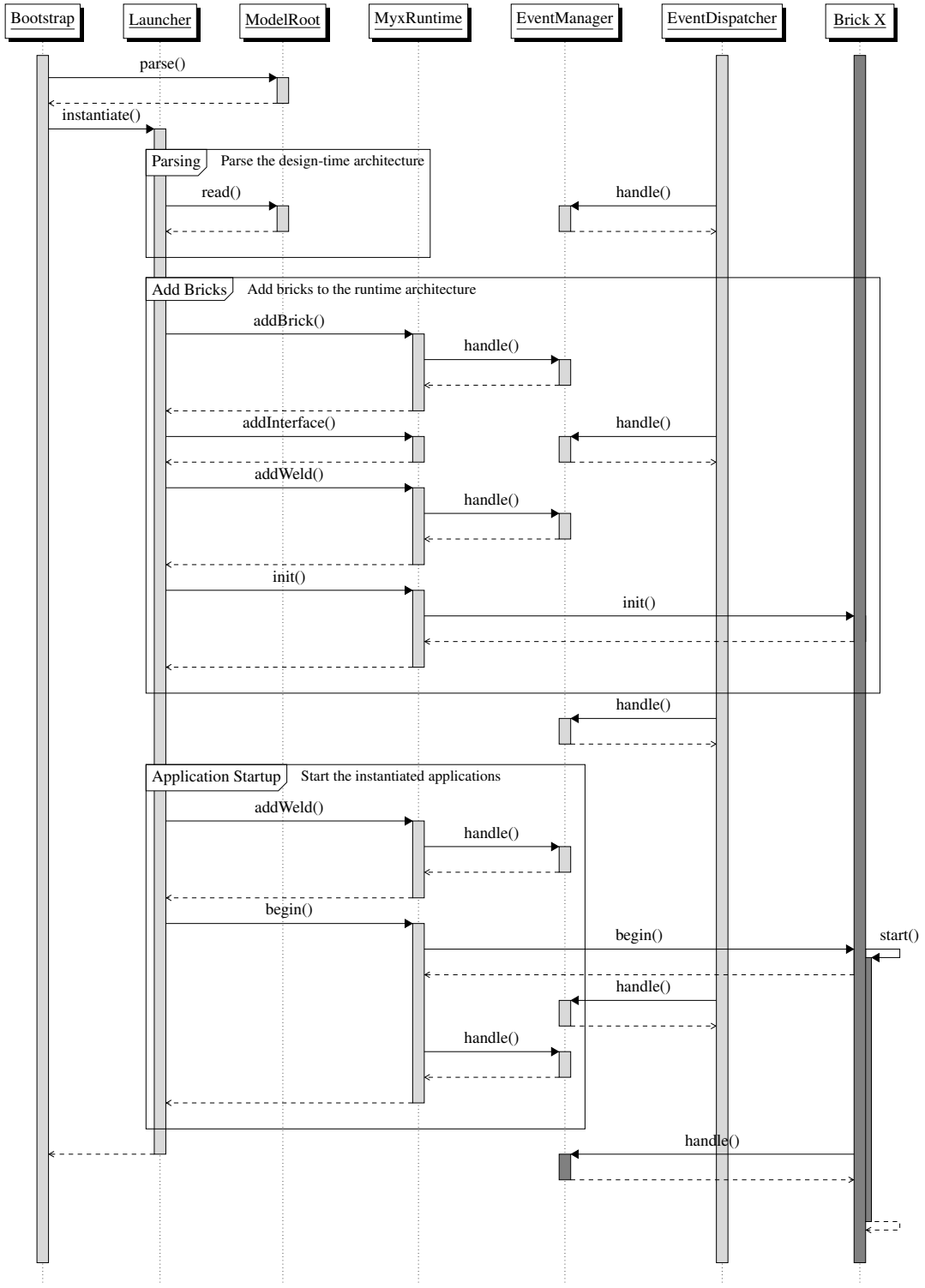


Figure 5.2: The *Monitor* application's instantiation process and the included event propagation depicted as an UML sequence diagram.

The event handling of the *Aggregator* is shown in Figure 5.3. Each incoming connection, that is the *EventManager* instance of a monitored application, is represented by an instance of the *PublisherEndpoint* component. This component simply listens and forwards all incoming events to the *MessageDistributor*. The *MessageDistributor* handles the event in an asynchronous manner and forwards it to all connected subscriber. The *XADLRuntimeManager* and *HostedRuntimeManager* are such subscribers and take care of the architectural aggregation, outlined in the diagram. As the *Aggregator* acts as a message broker and allows the further propagation of events. Each other subscriber is handled by an instance of the *SubscriberEnpoint* which simply forwards all received events to it.

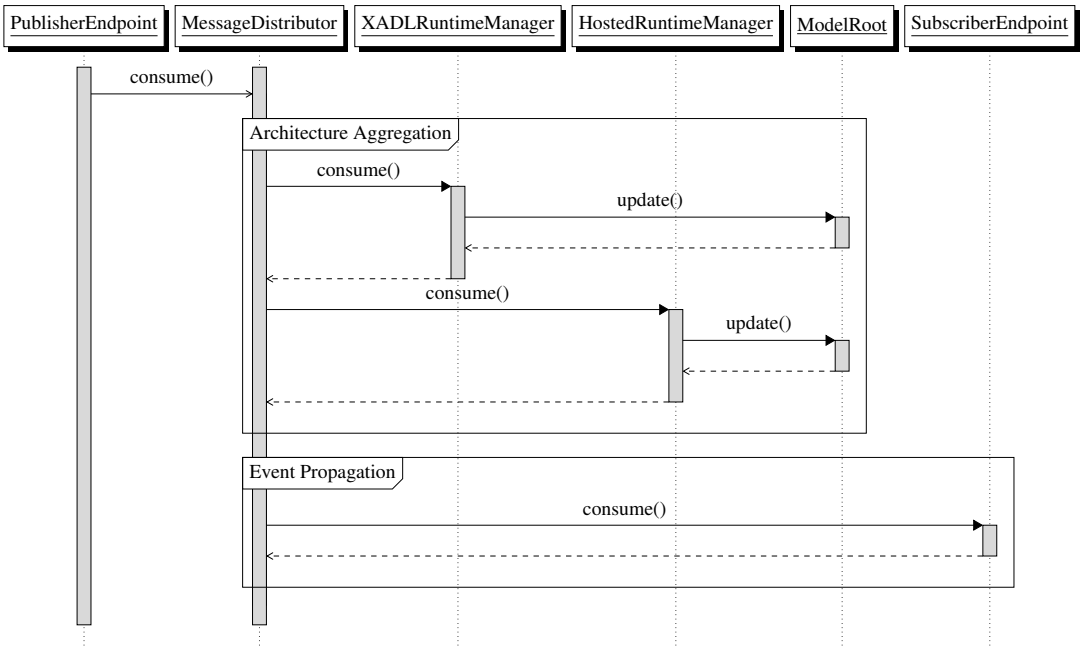


Figure 5.3: The *Aggregator*'s event handling including the runtime architecture aggregation and further event propagation as an UML sequence diagram.

Finally we will show how the *Aggregator* communicates with two *Monitor* instances and other subscribers in Figure 5.4. This diagram contains two *EventManager* instances that are associated with two distinct monitored applications. It shows how the incoming events are handled asynchronously by the *Aggregator*'s *MessageDistributor* and forwarded to the subscriber in a sequential way.

5.5 Framework Integration

In this section we will describe how our framework may be integrated into a Myx and *myx.fw* based application. Here we show which steps are already handled by the framework and which steps a developer must take to fully integrate our framework.

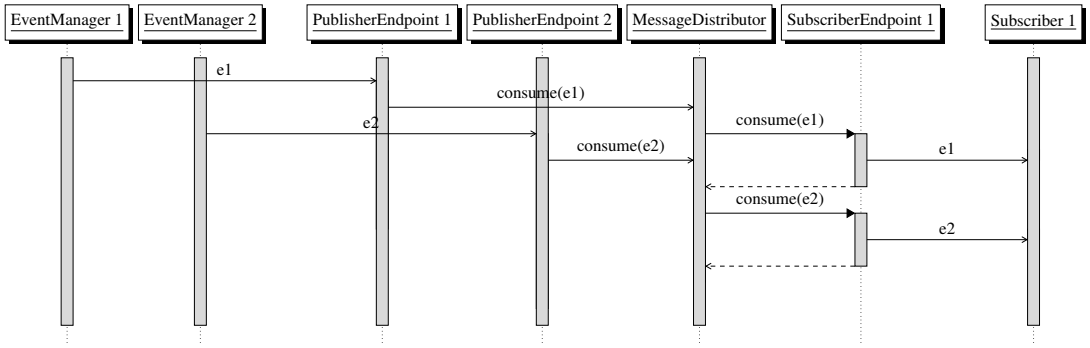


Figure 5.4: The communication between the *Monitor* and the *Aggregator* applications as an UML sequence diagram.

We will show this by using two application scenarios, a simple (non-distributed) application, where connections are not dynamically established, and a distributed application with dynamic connection establishment.

5.5.1 Instantiation

The first step to integrate our framework into an application is to use it to instantiate the application to be monitored. Therefore the framework has to be added to the application's *classpath*, which enables the framework to interact with the classes of the application. Once the *classpath* has been adapted we are able to launch the application using our command line bootstrapping application, called *Bootstrap*[22].

This application instantiates our *Monitor* which instantiates the application itself. The *Bootstrap* allows to define certain settings at the application's startup. These settings are given as command line parameters. Listing 5.9 shows the usage of the *Bootstrap* and gives an overview of the customizable settings.

```
Usage:
  java at.ac.tuwien.dsg.myx.monitor.Bootstrap file [--structure structureName] [--id
      architectureRuntimeId] [--event-dispatcher className] [--event-manager className
      ] [--event-manager-connection-string connectionString]

where:
  file: the name of the xADL file to bootstrap
  --structure structureName: the name of the structure to bootstrap
  --id architectureInstanceId: the architecture runtime id
  --event-dispatcher className: the event dispatcher class name that should be
      instantiated
  --event-manager className: the event manager class name that should be used to
      propagate events
  --event-manager-connection-string connectionString: the connection string that should
       be used to propate events
```

Listing 5.9: Usage of our command line bootstrapping application

[22] `at.ac.tuwien.dsg.myx.monitor.Bootstrap`

The *Bootstrap* application only requires a single parameter to instantiate an application, all other parameters are optional:

- *file* — The only required parameter is the file where the design-time architecture of the application is defined, given as a xADL document. This file is forwarded to the *ModelRoot*.
- *structure* — Using this parameter allows to define the *ArchStructure* which should be instantiated. As a xADL document may contain multiple *ArchStructure* elements this parameter gives us the ability to choose which *ArchStructure* should be instantiated. If this parameter is not given we use the first *ArchStructure* that is found in the xADL document.
- *id* — This parameter allows to define a custom *architectureRuntimeId* which is used by the *EventManager* component and injected into each propagated event. By using the same *architectureRuntimeId* we are able to group the runtime architecture of a distributed application in the aggregation process. If the parameter is omitted a random *architectureRuntimeId* is generated.
- *event-dispatcher* — Using this parameter allows us to add custom *EventDispatcher* instances by providing the class name of the instance. Here we allow to define multiple instances by repeatedly using the parameter. The *Bootstrap* application forwards all given class names to the *EventDispatcher* component which creates instances of these classes and linking them to the *EventManager*. It is important to note that the application halts if an *EventDispatcher* instance could not be created.
- *event-manager* — As the default implementation of the *EventManager* drops all incoming events we have defined an easy way to customize the used *EventManager* class. By providing the class name of the instance to be used our *Bootstrap* takes care of instantiating the *EventManager* instance. It is important to note that the application halts if the *EventManager* instance could not be created.
- *event-manager-connection-string* — To enable the *EventManager* instance to connect to different *Aggregator* instances it is possible to pass a simple connection string to it. The *EventManager* instance has to handle the usage of the connection string itself as the *Bootstrap* only passes it to it.

If the provided settings are not sufficient for correctly instantiating an application it is possible to extend the *Bootstrap* class and modify the behavior as required. As an example we have used an extended class for running the *Aggregator* application which can be seen in the next section.

To get a better understanding of the usage of the described *Bootstrap* application we show it's usage by running the message broker of our evaluation scenario. Listing 5.10 shows the command which allows us to run the application. Here we have not specified any of the extended settings and simply told the *Bootstrap* application to instantiate it.

```
$ java at.ac.tuwien.dsg.myx.monitor.Bootstrap pubsub.xml --structure pubsub
```

Listing 5.10: Instantiation of the message broker of our evaluation application

Listing 5.11 shows a command to instantiate the message broker as well. Here we have used all of our optional settings to configure the monitoring of the application. We have specified a custom *EventManager* implementation with an associated connection string. We have also used the previously described *EventDispatcher* for monitoring the CPU utilization of the host.

```
$ java at.ac.tuwien.dsg.myx.monitor.Bootstrap pubsub.xml --structure pubsub --id
    ArchitectureRuntimeId-72d4a895-4960-4bcb-af73-d0c1036cee55 --event-dispatcher at.
    ac.tuwien.dsg.myx.monitor.ed.dispatchers.CPUMonitor --event-manager at.ac.tuwien.
    dsg.pubsub.em.EventManagerImpl --event-manager-connection-string "tcp://localhost
    :9000"
```

Listing 5.11: Instantiation of the message broker of our evaluation application with included monitoring capabilities

5.5.2 Monitoring

To be able to monitor and aggregate the runtime architecture of an application it is required to start the *Aggregator* application before the application to be monitored. To prepare the *Aggregator* for the aggregation of an application it is required to adapt the initialization parameters of the *ModelRoot* component. Here the parameter *file* must point to the xADL document containing the design-time architecture of the application to be monitored. After these adaptations have been done the *Aggregator* can be started by it's bootstrapping class *MyxMonitoringAggregator*[23], see Listing 5.12. This class is an extension of the *Bootstrap* class and eases the startup of the application. Listing 5.13 shows how the application may be run using the *Bootstrap* class.

```
$ java at.ac.tuwien.dsg.myx.monitor.aggregator.MyxMonitoringAggregator
```

Listing 5.12: Running the *Aggregator*

```
$ java at.ac.tuwien.dsg.myx.monitor.Bootstrap myx-monitor-aggregator.xml --structure
    aggregator
```

Listing 5.13: Running the *Aggregator* using the *Bootstrap* application

After the *Aggregator* is running the used *EventManager* instance has to be pointed to the *Aggregator*. Therefore it is required that a compatible *EventManager* implementation is used and the injected connection string is used to direct it to the *Aggregator*. Listing 5.11 shows how this behavior may be achieved.

5.5.3 Integration for Distributed Applications

The simple monitoring of the runtime architecture may be sufficient for small applications but may not suffice for distributed applications. Distributed applications often utilize connections between each other or other external services that cannot be described using Myx or created using *myx.fw*. These applications may use the abstract class *AbstractMyxExternalConnectionBrick* described in Section 5.2.3.

[23] `at.ac.tuwien.dsg.myx.monitor.aggregator.MyxMonitoringAggregator`

Distributed applications may handle connections in different ways. Whilst some approaches may bundle external connections into a single brick others may use one brick per connection. This behavior cannot be described using Myx and is difficult to implement using *myx.fw*. The message broker of our evaluation application is designed to handle each connection, be it incoming or outgoing, using it's own brick. Therefore we had to extend *myx.fw* to provide a way to manipulate the runtime architecture of an application and allow for the dynamic creation of bricks. We have achieved this behavior by exposing the used *IMyxRuntime* instance and allow the application to access it. Therefore we have created an abstract class *AbstractMyxMonitoringRuntimeAdapter*[24] which allows it's subclasses to access the current *IMyxRuntime* instance using the method *getMyxRuntime*. Listing 5.14 shows an excerpt of the class and the exposed method. As we can see the class is intended to be the basis of a simple brick.

As it would be possible to fully expose the *IMyxRuntime* instance it is encouraged to only allow the modification of the runtime architecture using a defined interface. In the message broker of our evaluation application we have designed an interface named *IMyxRuntimeAdapter*[25] which exposes methods for the allowed runtime architecture modifications. We then created a brick named *MyxRuntimeAdapter*[26] that implements this interface and thus exposes the runtime architecture manipulation. To allow other bricks in the runtime architecture to use it we have created a component *MyxRuntimeAdapter* which uses this class. Listing 5.15 shows this brick and how we have used the *IMyxRuntime* instance to manipulate the runtime architecture.

```
package at.ac.tuwien.dsg.myx.monitor;

[...]

public abstract class AbstractMyxMonitoringRuntimeAdapter extends
    AbstractMyxSimpleBrick {

    [...]

    protected final IMyxRuntime getMyxRuntime() { [...] }
}
```

Listing 5.14: The exposed methods of the *AbstractMyxMonitoringRuntimeAdapter* class.

[24] `at.ac.tuwien.dsg.myx.monitor.AbstractMyxMonitoringRuntimeAdapter`
[25] `at.ac.tuwien.dsg.pubsub.middleware.interfaces.IMyxRuntimeAdapter`
[26] `at.ac.tuwien.dsg.pubsub.middleware.comp.MyxRuntimeAdapter`

```
package at.ac.tuwien.dsg.pubsub.middleware.comp;

public class MyxRuntimeAdapter extends AbstractMyxMonitoringRuntimeAdapter implements
    IMyxRuntimeAdapter {

    [...]

    @Override
    public void createPublisherEndpoint(String publisherEndpointClassName, Dispatcher
        <?> dispatcher) {
        IMyxName publisherEndpoint = MyxUtils.createName([...]);
        [...]
        getMyxRuntime().addBrick(PATH, publisherEndpoint, publisherEndpointDesc);
        [...]
        getMyxRuntime().addInterface(PATH, publisherEndpoint, MyxInterfaceNames.
            IDISPATCHER, dispatcherDesc, EMyxInterfaceDirection.OUT);
        [...]
        getMyxRuntime().init(PATH, publisherEndpoint);
        [...]
        getMyxRuntime().addWeld(getMyxRuntime().createWeld(PATH, publisherEndpoint,
            MyxInterfaceNames.IDISPATCHER, PATH,
        MyxUtils.getName(dispatcher), MyxInterfaceNames.IDISPATCHER));
        [...]
        getMyxRuntime().begin(PATH, publisherEndpoint);

    }

    @Override
    public void shutdownPublisherEndpoint(PublisherEndpoint<?> endpoint) {
        [...]
        getMyxRuntime().end([...]);
        getMyxRuntime().destroy([...]);
        [...]
        getMyxRuntime().removeWeld([...]);
        [...]
        getMyxRuntime().removeInterface([...]);
        [...]
        getMyxRuntime().removeBrick(PATH, brickName);
    }

    @Override
    public void createSubscriberEndpoint(String subscriberEndpointClassName, Dispatcher
        <?> dispatcher) {
        [...]
    }

    @Override
    public void wireSubscriberEndpoint(SubscriberEndpoint<?> subscriber) {
        [...]
    }

    @Override
    public void shutdownSubscriberEndpoint(SubscriberEndpoint<?> endpoint) {
        [...]
    }
}
```

Listing 5.15: The usage of the *AbstractMyxMonitoringRuntimeAdapter* class in our evaluation application.

CHAPTER 6

Evaluation

Our monitoring approach was created to be easily integrated into new and existing *myx.fw* applications. To evaluate that our approach is working as described we have constructed a publish-subscribe system which will be used to show the feasibility of our approach. As an extension of this evaluation application we have applied our tools on ourself and used the message broker as the basis for our *Aggregator* application.

This chapter will describe the general implementation details of our evaluation application, the different scenarios as well as their results. This chapter will close with the definition of best-practices and eventual shortcomings of the approach.

6.1 Implementing a Publish-Subscribe System

We have chosen to use a simple publish-subscribe system [3] as our evaluation scenario which currently supports the topic-based publish-subscribe scheme [20]. As it is a proof of concept implementation we have created a system that does not persist messages and is thus using the concept of send-and-forget. The system is tailored for the transmission of audio streams which leads to some interesting challenges:

- Meta-data of audio streams has to be transmitted to each connecting client to ensure the playback.
- The stream itself has to be transferred in the correct order to each client.
- The dynamic handling of many clients using the Myx architectural style and it's associated tools.

This section will give an overview of the design-time architecture for each part of the evaluation application and will describe the general message structure as well as the routing process.

6.1.1 Architecture

Our evaluation application is not only composed of a message broker but also a publisher and subscriber. Each of these is constructed as an application of it's own and thus has it's own design-time architecture. In the following we will describe the design-time architecture of each application with the focus lying on the architecture of the message broker as it is reused in our *Aggregator* application.

Message Broker

The message broker implements the described publish-subscribe pattern utilizing the topic-based scheme. Our approach handles each client by using it's own connector. This characteristic is one of the main challenges of our approach. Each incoming connection is accepted by a dispatcher unit which delegates the creation of the connector to the underlying runtime. In the following we will describe the design-time architecture of the application and how we implemented the dynamic creation of connectors.

Figure 6.1 shows the design-time architecture of the application. We will describe each component and connector that is contained in this architecture to get a better understanding of it's tasks.

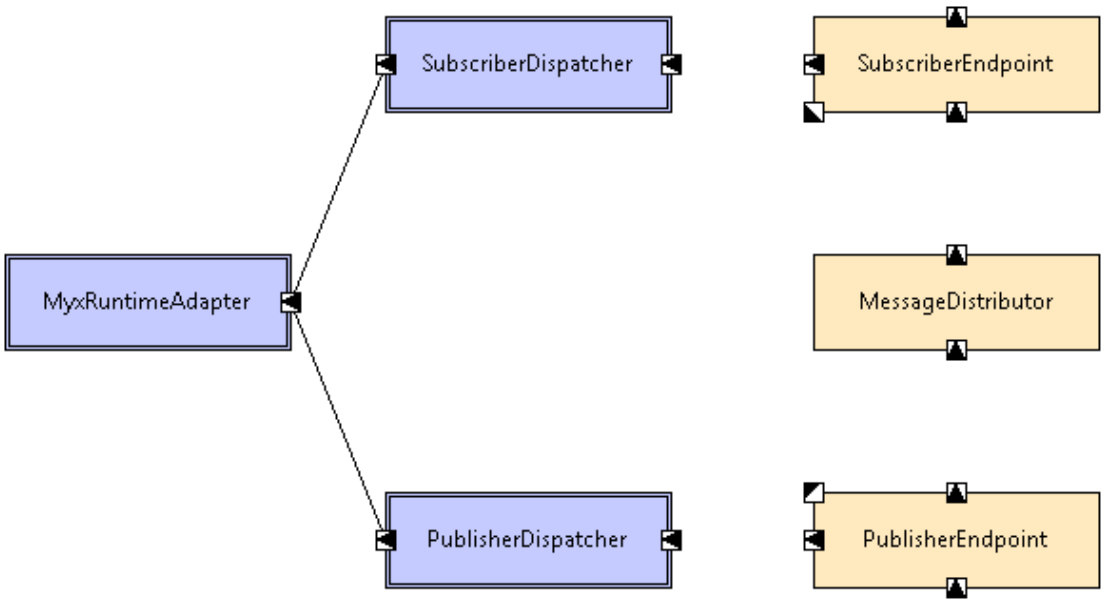


Figure 6.1: The architecture of the message broker.

PublisherDispatcher and SubscriberDispatcher These two components are responsible for handling of incoming connections. They share a common functionality with the only difference being the creation of different connectors. Each component waits for an incoming connection and instructs the underlying runtime to create a new handling instance. This instance is either

a *PublisherEndpoint* or *SubscriberEndpoint*. The incoming connection is then provided to the created handling instance by using an interface that allows the handling instance to poll the connection. Due to the inner workings of the *myx.fw* it is not possible to easily forward the connection to the created connector, thus we used the polling approach to forward it. The polling itself is defined by an interface called *IDispatcher* which is described in Listing 6.1.

The creation of the handling instances is done via the *MyxRuntimeAdapter* component.

```
package at.ac.tuwien.dsg.pubsub.middleware.interfaces;

import at.ac.tuwien.dsg.pubsub.network.Endpoint;

public interface IDispatcher<E> {
    /**
     * Get the next pending endpoint.
     *
     * @return
     */
    public Endpoint<E> getNextEndpoint();
}
```

Listing 6.1: The *IDispatcher* interface definition.

MyxRuntimeAdapter This component exposes the underlying runtime to the other components and connectors, that is we expose a portion of the used *IMyxRuntime* instance. This allows us to directly manipulate the runtime architecture of the application and dynamically create and remove specific components or connectors. This behavior is exposed via an interface that defines the allowed runtime architecture manipulation. Here it would be possible to expose the complete runtime architecture but it is encouraged to only specify the behavior that is allowed by the application.

Currently our approach uses this feature to create instances of *PublisherEndpoint* and *SubscriberEndpoint* instances.

PublisherEndpoint This connector is one of the two dynamically created connectors in our design-time architecture. It is not instantiated whilst starting up the application but once a publisher connects. The creation is done using the *MyxRuntimeAdapter* component. It fetches the incoming connection from the *PublisherDispatcher* instance and then listens for incoming messages. Each message is directly forwarded to the *MessageDistributor* connector which takes care of forwarding them to the appropriate subscribers. The connector listens for messages until the connection is closed or an message of type *CLOSE* is received. In this case the connector removes itself from the runtime architecture using the *MyxRuntimeAdapter* component.

SubscriberEndpoint This connector, as the *PublisherEndpoint*, is dynamically created using the *MyxRuntimeAdatper* component. It is instantiated once a subscriber connects and fetches the connection from the *SubscriberDispatcher* instance. The connector first consumes the subscribed topics from the connection identified by a message of type *TOPIC*. Once the topics are received it fully connects itself to the *MessageDistributor* connector using the *MyxRuntimeAdapter* thus receiving all messages from the connected publishers. As the *PublisherEndpoint* if the connection is closed it removes itself from the runtime architecture.

The *SubscriberEndpoint* exposes a generic interface called *ISubscriber* which defines a single method to consume messages named *consume*, see Listing 6.2. Using a generic interface once more allows us to transfer all kinds of messages. This interface does not yet impose any restrictions on the received messages thus it is the responsibility of the implementing class to handle the filtering of messages. This method is invoked by the *PublisherEndpoint* through the *MessageDistributor* connector.

```
package at.ac.tuwien.dsg.pubsub.middleware.interfaces;

import at.ac.tuwien.dsg.pubsub.message.Message;

public interface ISubscriber<E> {
    /**
     * Consume a received message.
     *
     * @param message
     */
    public void consume(Message<E> message);
}
```

Listing 6.2: The *ISubscriber* interface definition.

MessageDistributor This connector is the central exchange point between the connected publishers and the subscribers. It is based on the *event pump* that is already a part of the Myx architectural style but implements a special kind of asynchronous notifications that allows us to forward all incoming messages in the order they were received whilst using features of concurrency. The origins of this special requirement is our audio streaming scenario where it is important that the messages are delivered in the correct order to ensure the playback of a stream.

Publisher and Subscriber

The message broker is the main part of our evaluation scenario but consists of the publisher and subscriber as well which send and receive messages respectively. In the following we will describe the architectures of both applications.

The publisher is used to send messages to the message broker. As our evaluation scenario is about transmitting audio streams it sends audio based messages. Figure 6.2 shows it's design-time architecture which consists of one component and one connector. The *MessageCreator* component is responsible for generating messages and sending it to the *Publisher* connector. This connector forwards the received messages to the message broker or any other compatible endpoint.

As our evaluation scenario suggests we have created a *MessageCreator* implementation that allows to transmit an audio stream to the message broker. Here we use the name of the stream as the topic. Using this topic a subscriber is capable of receiving the audio stream. As a first step we send an initial message containing the meta data of the stream. This message is stored until the stream is finished and sent to each connected subscriber.

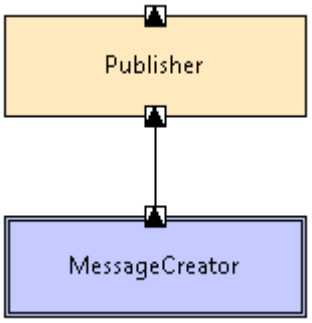


Figure 6.2: The architecture of publisher.

The subscriber on the other side is used to receive messages. In our evaluation case we play back a received audio stream. Figure 6.3 shows the design-time architecture of the subscriber, it consists of a connector and a component. The *Subscriber* connector is responsible for connecting to the message broker or a any other compatible client. It publishes the subscribed topics and waits for incoming messages which are directly forwarded to the *MessageConsumer*. The *MessageConsumer* component is responsible for consuming the received messages.

For our evaluation scenario we have created a *MessageConsumer* implementation that uses the received messages to play back the transmitted audio stream. The *Subscriber* connector publishes the subscribed audio stream name as the topic so the message broker forwards the correct audio data.

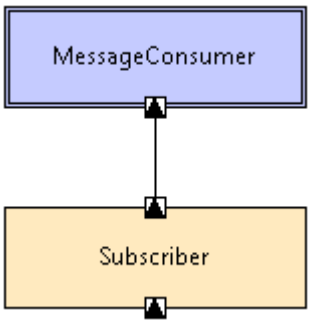


Figure 6.3: The architecture of the subscriber.

6.1.2 Messages

Our publish-subscribe systems uses simple messages for communication. A message consists of a type, a topic and the payload. The payload of a message may hold any kind of data, thus our approach imposes no restrictions on the transferred data.

Our evaluation application is based on a generic class called *Message*[1]. This class contains the described properties of a message, i.e. it's type, topic and payload. As this class is generic

[1] `at.ac.tuwien.dsg.pubsub.message.Message`

we are able to hold any kind of data as long as there exists a Java representation of it. The source code of this class may be found in the appendix.

We currently support five types of messages: *TOPIC*, *INIT*, *DATA*, *CLOSE* and *ERROR* that arise from the audio streaming scenario. Each type of message is handled in a differnt may:

- *TOPIC* — This message represents the subscribing topics of a subscriber. Only if a subscriber sends this kind of message it is fully connected to the messaging infrastructure.
- *INIT* — Publishers are able to send these kind of messages to instruct the message broker to send them to each connected subscriber on connection establishment. The origins of this kind of message lies in the transport of audio streams, where the first bytes of a stream contain meta-data about it and thus need to be received by the subscriber to ensure the correct output.
- *DATA* — This kind of message represents the usual data transfered by a publish-subscribe system.
- *CLOSE* — Another message type originating in the transport of audio streams. The publisher is able to signal the end of a stream by publishing this kind of message, thus removing all messages of type *INIT* associated with the stream.
- *ERROR* — Represents a general error emitted by the publisher, message broker or the underlying transport protocol.

Along with the type of a message the corresponding topic is contained. This topic is represented as a simple string and is used in the message routing process to select the receiving subscribers.

6.1.3 Client Handling

Each client in our publish-subscribe system is represented by a generic interface called *Endpoint*[2]. A client can be described as the end of an external connection which is capable of sending and receiving messages. Listing 6.3 shows the interface definition and thus the two exposed methods *receive* and *send*.

Our current implementation uses sockets and a simple text based protocol for the transportation of messages. Here we are able to transfer any kind of message by serializing the message itself.

[2] `at.ac.tuwien.dsg.pubsub.network.Endpoint`

```java
package at.ac.tuwien.dsg.pubsub.network;

import java.io.Closeable;
import java.io.IOException;

import at.ac.tuwien.dsg.pubsub.message.Message;

/**
 * Interface that specifies the methods of a real network endpoint.
 *
 * @param <E>
 */
public interface Endpoint<E> extends Closeable {
    /**
     * Receive a message.
     *
     * @return
     */
    public Message<E> receive() throws IOException;

    /**
     * Send a message.
     *
     * @param msg
     */
    public void send(Message<E> msg) throws IOException;
}
```

Listing 6.3: The *Endpoint* interface definition.

6.1.4 Message Routing

The differences between publishers and subscribers allows for a custom handling of these two clients, combining blocking- and non-blocking Input/Output (IO).

Once a publisher connects an instance of *PublisherEndpoint* is created. This connector starts a new thread which listens for incoming messages. Once a message is received it is directly forwarded to the *MessageDistributor*. We listen for messages until a closing message is received, that is a message of type *CLOSE*, or the connection was closed.

If a subscriber connects an instance of *SubscriberEnpoint* is dynamically created. As a first step we wait for the subscriber to publish it's subscribed topics. These topics are published using a message of type *TOPIC* and are specified as a simple list. We support different kinds of topics, that is simple strings, regular- or glop pattern expressions [39]. Each topic is represented by an interface called *Topic*[3] which exposes a single method *matches*, see Listing 6.4. We have created matching implementations for all our supported topic types which can be found in the appendix.

[3] `at.ac.tuwien.dsg.pubsub.message.topic.Topic`

```
package at.ac.tuwien.dsg.pubsub.message.topic;

/**
 * Interface providing the methods used to match a topic pattern.
 */
public interface Topic {
    /**
     * Return if the given topic matches the pattern.
     *
     * @param topic
     * @return
     */
    public boolean matches(String topic);
}
```

Listing 6.4: The *Topic* interface definition.

Once the subscriber has published it's subscribed topics it is connected to the *MessageDistributor* and is thus able to receive messages. If a message is published the *PublisherEndpoint* invokes the *consume* method of the *ISubscriber* interface, implemented by the *SubscriberEndpoint*, through the *MessageDistributor*. Our implementation first validates if one of the subscribers topics matches the topic of the message and forwards it to the subscriber. If the topics does not match the message is simply ignored.

The *MessageDistributor* is the central exchange point for messages as it forwards the method calls of all *PublisherEndpoint* instances to all *SubscriberEndpoint* instances. It acts as a connector between the two connectors and forwards all calls from the *PublisherEndpoint* instances to the *consume* method of the *ISubscriber* interface to all *SubscriberEndpoint* instances.

The connector is based on the *EventPumpConnector*[4], which is the *myx.fw* implementation of the asynchronous notification pattern. As all connectors in the *myx.fw* it implements the *InvocationHandler*[5] interface which allows the instance to be used by a proxy object and thus allows the implementation of the communication patterns described by Myx. By implementing the interface *IMyxDynamicBrick* the connector gets notified once a new Myx interface is connected or disconnected using the methods *interfaceConnected* and *interfaceDisconnected*. We use these methods to keep track of all connected outgoing interfaces which allows us to forward all method calls to them. The *EventPumpConnector* uses a single thread, which is represented by an instance of the interface *Executor*[6] or *ExecutorService*[7], for asynchronous execution which guarantees the required ordering. Using a single thread to forward messages is not feasible if many publishers or subscribers are connected. As any other implementation of the *Excecutor* using multiple threads does not guarantee that the messages are delivered in the correct order we had to create an extension of the *Executor* and *ExecutorSercive* interface called *IdentifiableExecutorService*[8]. This interface extends the *ExecutorService* by a single method *execute* that allows to specify an identifier for a given *Runnable*, see Listing 6.5. This identifier guarantees that all tasks submitted with the same identifier are executed by the same thread.

[4] `edu.uci.isr.myx.conn.EventPumpConnector`
[5] `java.lang.reflect.InvocationHandler`
[6] `java.util.concurrent.Executor`
[7] `java.util.concurrent.ExecutorService`
[8] `at.ac.tuwien.dsg.concurrent.IdentifiableExecutorService`

The *MessageDistributor* uses the called object itself, that is the subscriber, as an identifier thus guaranteeing that the messages are delivered sequentially.

The *MessageDistributor* allows to store initial messages, that is messages of type *INIT*, which may hold meta-data, e.g. the used audio codec for a stream, and send them to each connected outgoing connection. Here we save all such *Message* instances and forward them once a subscriber is connected to the *MessageDistributor*. Here we use the method *interfaceConnected* of the *IMyxDynamicBrick* interface to directly forward all stored initial messages to the subscriber.

```
package at.ac.tuwien.dsg.concurrent;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.RejectedExecutionException;

/**
 * An extension to the {@link ExecutorService} interface that allow to
 * execute tasks on specific {@link Thread}s using identifiers.
 */
public interface IdentifiableExecutorService extends ExecutorService {
    /**
     * Executes the given command at some time in the future. The command may
     * execute in a new thread, in a pooled thread, or in the calling thread,
     * at the discretion of the {@code Executor} implementation. This method
     * guarantees that a {@link Runnable} with the same identifier is always
     * executed by the same {@link Thread}.
     *
     * @param command
     *            the runnable task
     * @param identifier
     *            the identifier
     * @throws RejectedExecutionException
     *             if this task cannot be accepted for execution
     * @throws NullPointerException
     *             if command is null
     */
    void execute(Runnable command, int identifier);
}
```

Listing 6.5: The *IdentifiableExecutorService* interface definition.

6.2 Problem Instances

For the evaluation of our framework we have created different scenarios which demonstrate it's feasibility and stability. In this section we will first describe the goal of each scenario and their instantiation order including the amount of instances that are actually created.

6.2.1 Audio-Streaming Based Publish-Subscribe

The first evaluation scenario is based on our motivated scenario. We have created a publish-subscribe implementation that is capable of streaming simple audio messages based on the Java package *javax.sound.sampled*. Thus we are able to play sounds stored in Waveform Audio File Format (WAVE), which is natively supported by Java.

In this scenario the publisher reads an audio file into a simple byte buffer, packs the data into an instance of the *Message* class and sends it to the message broker. Due to WAVE we have to treat the first read message as meta-data thus publishing it with the type *INIT* which enables all subscribers to correctly play the streamed sound. Once the whole file has been published we publish a message of type *CLOSE* enabling all subscribers to stop the playback and shut down correctly. We use the name of the audio file as our topic to allow for the concurrent streaming of multiple audio streams.

The subscriber is also constructed in a very simple way. It waits for a message of type *INIT* and creates an audio stream based on the Java package *javax.sound.sampled*. It forwards all received messages of type *DATA* to the audio stream, enabling the audio playback. The subscriber listens for messages until a message of type *CLOSE* is received or the connection is closed. Here we subscribe to the previously named filename of the audio file so we receive the correct stream.

For this scenario we have used two audio files each with a length of around five minutes which results in a maximum subscriber runtime of around five minutes.

The amount of publishers and subscribers in this scenario is fixed. We define two sub-scenarios that consist of the following instantiation orders:

- **Scenario 1**
  1. The message broker.
  2. One subscriber instance subscribing to the first audio file.
  3. One publisher instance publishing the first audio file.
- **Scenario 2**
  1. The message broker.
  2. Three subscriber instances, two subscribing to the first audio file and one subscribing to the second one.
  3. Two publisher instances, one publishing the first audio file and the other one publishing the second one.

We have verified that the audio streaming was working as expected by ensuring that the stream was played correctly. Scenario 2 was verified by playing a single stream at a time by only allowing one subscriber to actually output sound. The volume of all other subscribers has been reduced to zero. This required us to execute the scenario multiple times.

6.2.2 Event Based Publish-Subscribe

Our second evaluation scenario is our framework itself, in more detail our *Aggregator* application. As the message broker of our evaluation scenario is the basis for the *Aggegator* application we use the publish-subscribe pattern to propagate architectural properties and thus we have created an additional evaluation scenario. Here we can validate the feasibility by comparing the aggregated runtime architecture with the expected one.

6.2.3 Monitoring Distributed Applications

To show that our evaluation application is working correctly in a distributed environment with many application instances we have created different scenarios to demonstrate that our approach is capable of handling them. For these scenarios we have created custom implementations of a publisher and a subscriber.

Each publisher that is used in these scenarios is transmitting empty dummy messages. Here we used a custom *MessageCreator* component which generates a specific amount of messages and transmits them to the message broker. It is possible to define a waiting period between each transmitted message. This allows us to send small messages over a long period of time.

The used subscriber has also been adapted for this scenario. Here we use a custom *MessageConsumer* component that simply drops all incoming messages that are transferred to it.

To ease the testing process we have extend our bootstrapping application that has been described in Section 5.5.1 and is named *LoadTestBootstrap*[9]. This application extends the known settings by allowing the instantiation of multiple application instances. Listing 6.6 shows the usage of this application and gives an overview of the newly customizable settings.

```
Usage:
  java at.ac.tuwien.dsg.myx.monitor.evaluation.LoadTestBootstrap file [--structure
      structureName] [--id architectureRuntimeId] [--event-dispatcher className] [--
      event-manager className] [--event-manager-connection-string connectionString]
      [--amount count] [--ramp-up-time seconds] [--run-time seconds]

where:
  file: the name of the xADL file to bootstrap
  --structure structureName: the name of the structure to bootstrap
  --id architectureInstanceId: the architecture runtime id
  --event-dispatcher className: the event dispatcher class name that should be
      instantiated
  --event-manager className: the event manager class name that should be used to
      propagate events
  --event-manager-connection-string connectionString: the connection string that should
       be used to propate events
  --amount count: the amount of instances to create
  --ramp-up-time seconds: the time in which the instances should be launched
  --run-time seconds: the time in seconds how long the application is kept running
```

Listing 6.6: Usage of our command line load testing bootstrapping application

The *LoadTestBootstrap* application extends *Bootstrap* by the following parameters, which once again are optional:

- *amount* — This parameter defines the applications instances that should be instantiated. The default behavior is to just start a single instance which matches the behavior of the *Bootstrap* application.

- *ramp-up-time* — By specifying a ramp-up time it is possible to define the timespan in which all application instances are to be instantiated. The default behavior is to start all instances as soon as possible which is a ramp-up time of zero.

[9] `at.ac.tuwien.dsg.myx.monitor.evaluation.LoadTestBootstrap`

- *run-time* — This parameter specifies the maximum runtime of all application instances. Once the maximum runtime is reached the shutdown of all instances is forced. If this parameter is omitted the instances may run indefinitely.

To get a better understanding of the usage of the described bootstrapping application we will show it's usage by running the publisher of our evaluation scenario. Listing 6.7 shows the command which allows to run an application like with the *Bootstrap* application, in this case the publisher of our evaluation scenario. By not specifying any parameters just a single application instance is created.

```
$ java at.ac.tuwien.dsg.myx.monitor.evaluation.LoadTestBootstrap pubsub.xml --structure
    pub
```

Listing 6.7: Instantiation of the publisher of our evaluation application using the *LoadTestBootstrap* application

Listing 6.8 shows the extended command line usage to instantiate multiple instances of the application. Here we have specified that five instances should be started in ten seconds. After 120 seconds the shutdown of all five application instances is forced.

```
$ java at.ac.tuwien.dsg.myx.monitor.evaluation.LoadTestBootstrap pubsub.xml --structure
    pub --amount 5 --ramp-up-time 10 --runtime 120
```

Listing 6.8: Instantiation of the publisher of our evaluation application using the *LoadTestBootstrap* application with usage of the described parameters

We have created different scenarios that show that our framework and the evaluation application are running correctly in an environment with many publishers and subscribers:

- **Scenario 1** — For our first scenario we use a fixed amount of 200 publishers and 200 subscribers. The ramp-up time for publishers is set to 10 seconds and the corresponding runtime is set to 50 seconds which yields a maximum runtime for our publishers of 60 seconds. For subscribers we use a ramp-up time of 30 seconds and a runtime of 30 seconds resulting in a maximum runtime of 60 seconds as well.

- **Scenario 2** — For the second scenario we keep the fixed amount of publishers and subscribers from the previous scenario. The ramp-up time is randomized between 20 and 40 seconds for both publishers and subscribers and the runtime is set to 30 seconds for both instances which results in a maximum runtime of 70 seconds.

- **Scenario 3** — The third scenario is fully randomized. We use a randomized amount of publishers and subscribers between 150 and 200 instances. We again use a random ramp-up time of 20 to 40 seconds. The runtime of all instances is now randomized between 20 and 40 seconds resulting in a maximum runtime of 80 seconds.

All of the three scenarios are executed using our *LoadTestBootstrap* application. This allows us to limit the maximum runtime of the instantiated applications as well as run them multiple times. To increase the total runtime of each scenario we run the described scenarios for publishers and subscribers 20 times, yielding a total runtime for each scenario of more than 20 minutes. We

thus simulate application instances that connect to and disconnect from the message broker over time.

6.2.4 Environment

All parts of our evaluation application are running in different environments. The message broker and publisher are being run on a server in the cloud and the subscriber is run locally. The *Aggregator* of our framework is also run on a server in the cloud environment.

For our distributed application simulation we chose to run all parts of the application in the cloud environment.

The cloud instances are based on OpenStack[10] and are using an Ubuntu 14.04 LTS Image with Java 7 installed. We used the *m2.medium* instance flavor which provides 3 VCPUs with 5760 MB of memory for each virtual machine.

The local environment is represented by a Dell Latitude E6410 laptop which contains an Intel Core i7 M620 CPU with 2.67 GHz and 8 GB memory and is running Windows 7 with Java 7 installed.

It is important to note that for all our evaluation runs we have started the aggregation component as a first step. Once it was in a running state the other instantiation steps were executed.

6.3 Results

The results of our evaluation scenarios consist of two parts, the runtime architecture of the evaluation application and different statistics of the execution.

6.3.1 Runtime Architecture

The main result while executing our aggregation component is a runtime architecture. This architecture is described using the xADL schema *xArch Instances* and our introduced extensions.

For each of our evaluation scenarios we will show the resulting runtime architecture once all application instances are running, that is all components and connectors are instantiated and linked.

6.3.2 Statistics

To extract more sophisticated data from the evaluation scenarios and it's execution we have created a special component called *StatisticsSubscriber*[11], which is directly integrated into the *Aggregator* as a subscriber. That is, it is connected to the *MessageDistributer* which enables us to inspect all received architectural properties.

The component is able to extract the following statistics and save them in the Comma Separated Values (CSV) format for them to be further processed:

[10] `http://www.openstack.org`

[11] `at.ac.tuwien.dsg.myx.monitor.aggregator.evaluation.StatisticsSubscriber`

- **Amount of instantiated bricks** — We are able to show the amount of instantiated brick in the monitored application over it's runtime.

- **Amount of active external connections** — The amount of active external connections between parts of an application is shown over it's runtime.

- **Amount of instantiated application instances** — Here we show the amount of application instances, i.e. parts of the monitored application that might be connected to each other.

- **Amount of hosts** — We show the amount of currently involved hosts in the runtime architecture.

- **Consumed events** — For each kind of event we show the amount that was consumed by the *Aggregator* over time which allows us to further analyze the aggregation process.

The values for the described statistics are saved for every second of the application's runtime which allows us to display them over time and easily extract aggregated values.

6.4 Evaluation of Feasibility

By executing the scenarios described in Section 6.2.1 and validating the extracted results we can assure that our approach is working as expected. We also validate the scenario described in Section 6.2.2 by using our *Aggregator* to extract the evaluation results which further proves the feasibility of our approach.

We will show this by first describing the runtime architecture of the scenario's application and comparing it with it's design-time architecture. As a last step we will show our custom statistics extracted by the *Aggregator*.

6.4.1 Scenario 1

To be able to execute the first scenario described in Section 6.2.1 we have to take a look at the design-time architectures of all used application parts. Thus we will show the message broker, the publisher and the subscriber, see Section 6.1.1 Figure 6.4 shows the combined design-time architecture of all three applications. To show where each application is executed we have added the hosts to the Figure, which are not part of the design-time architecture.

Once the scenario has been started, that is all applications were in a running state, we have extracted the runtime architecture from the *Aggregeator* which is shown in Figure 6.5. If we compare it to the described design-time architecture we can see that all bricks have been created and linked as expected. We can see that the message broker has dynamically instantiated and linked the *PublisherEndpoint* and *SubscriberEndpoint* connectors by using the *MyxRuntimeAdapter* which shows us that the dynamic creation of brick is working as described. The runtime status of each brick has been appended to it's description and is represented by the string *[RUNNING]*. We also notice that the publisher and subscriber have been linked with the

message broker, which can be seen by the connections between the bricks *Publisher* and *PublisherEndpoint* as well as between *SubscriberEndpoint* and *Subscriber*. To distinguish these external links from locally created links they are displayed as a dotted line in the Figure. The hosts the applications are running on are also shown in the Figure. If we compare them with the hosts described in the design-time architecture we can see that the information about the hosts was extracted correctly.

Our comparison between the design-time- and runtime architecture shows that everything was instantiated correctly thus we assume that the evaluation application as well as the *Aggregator* are working as described. To further prove the feasibility we will validate our custom statistics that were extracted over the runtime of the application using the *Aggregator*.

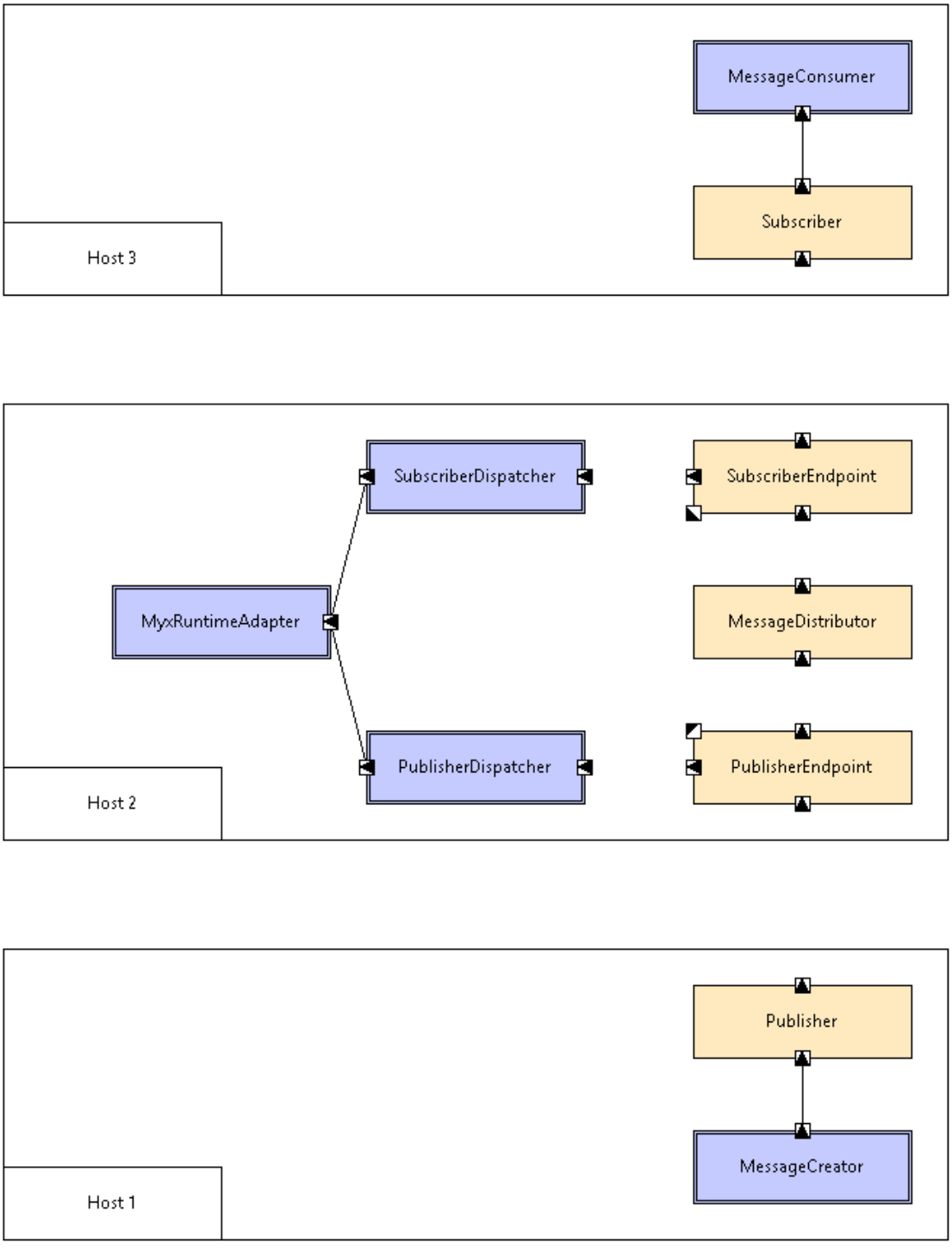


Figure 6.4: The design-time architecture of the audio-based publish-subscribe scenario using one publisher and subscriber.

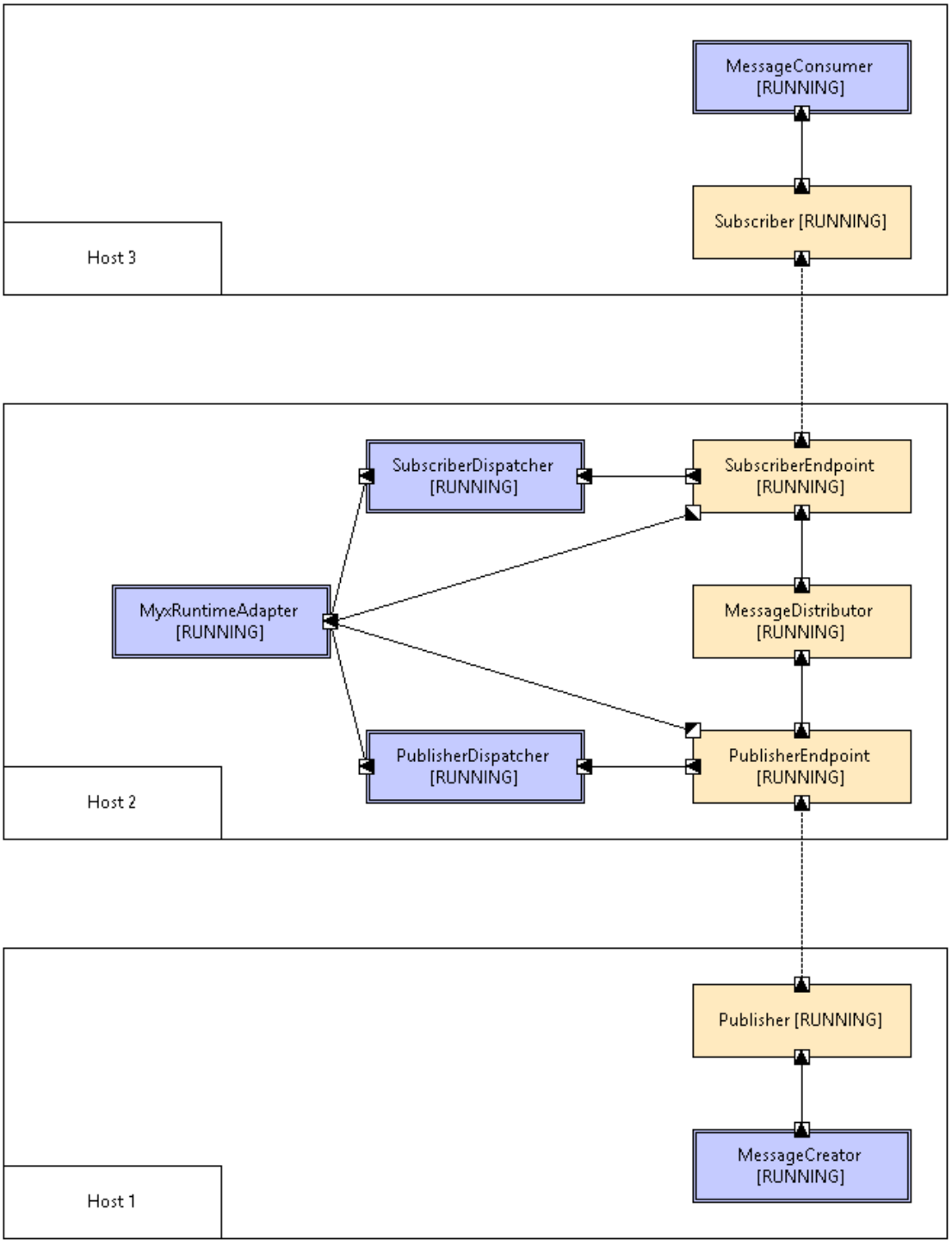


Figure 6.5: Resulting runtime architecture of the audio-based publish-subscribe scenario using one publisher and subscriber.

Over the runtime of our evaluation application different events arise. In the first step all applications are started and the the audio streaming begins. At around 200 seconds the publisher finishes the streaming of the file. Because reading an audio file and sending it over the network does not take as long as replaying the stream, the publisher finishes it's work way before the subscriber. After a runtime of 300 seconds, which is the length of the audio stream, the subscriber has finished the replay thus shutting itself down. Some time later we manually shut down the message broker labeling the end of our evaluation run.

We have thus described four major events in our evaluation run which are associated with changes in our runtime architecture. These changes can be shown using our custom statistics:

1. Application startup.
    - The amount of instantiated bricks is increased to 10 (see Figure 6.6), which is the correct amount of bricks for our running evaluation application.
    - The amount of external connections is increased to two (see Figure 6.7). These represent the connections between the publisher and subscriber to the message broker.
    - The amount of instantiated application instances increases to three (see Figure 6.8). These instances represent the instantiated message broker, publisher and subscriber.
    - The amount of hosts that run parts of our application increases to three (see Figure 6.9). This results due to the fact that we use one host for each application part.
2. Publisher shutdown at around 200 seconds of runtime.
    - At this time we can see that the amount of instantiated bricks drops to seven. That is the two bricks of the publisher are shut down as well as the *VirtualPublisherEndpoint* instance of the message broker.
    - Due to the shutdown the amount of external connections drops to one.
    - The amount of instantiated applications parts as well as hosts running them drops to two respectively.
3. Subscriber shutdown at around 300 seconds of runtime.
    - With the shutdown of the subscriber we can see that the amount of instantiated bricks drops to five. Here the *VirtualSubscriberEndpoint* is not shut down directly. That is the result of the asynchronous subscriber architecture which will leave the subscriber open until the next message is published.
    - Due to the shutdown the amount of external connections drops to zero, thus no such connections are contained in the runtime architecture anymore.
    - The amount of instantiated applications parts as well as hosts running them drops to one respectively.
4. Message broker shutdown at around 330 seconds of runtime.

- After the message broker is shut down all parts of the evaluation application are shut down, dropping the amount of instantiated bricks, external connections, application instances and hosts to zero.

The extracted statistics further prove the feasibility of our approach and we can see that the implemented publish-subscribe pattern works as expected in both applications using it, that is the message broker and the *Aggregator*.

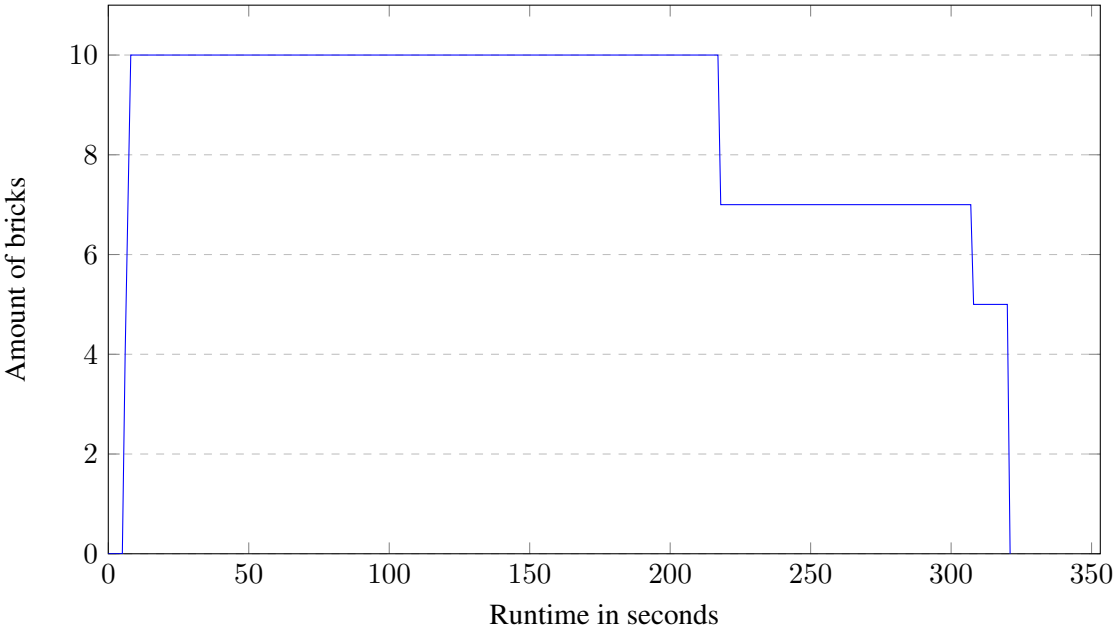


Figure 6.6: The amount of instantiated bricks over the runtime of the application.

To show how our approach performed we have show the average memory usage of each host whilst running the evaluation scenario. Figure 6.10 shows the memory usage in Mega Bytes (MB) over the runtime of the evaluation application. The memory usage of the message broker results from our architecture which allows publishers to publish message as soon as possible thus resulting in messages that are held back at the message broker as the subscribers are able to receive all these messages instantly.

For creating the runtime architecture and extracting the described statistics we solely relied on our architectural properties. Each change in the application's runtime architecture is accompanied by an event about the change, see Section 5.2.3. The transmitted events of this scenario can be seen in Figure 6.11 and are categorized by their type and their amount is shown over the runtime of the evaluation application.

As we can see most of the events were received once the state of the applications is changed, e.g. at the startup or shutdown of an application. The only type of event that is received continuously are events of type *XADLHostPropertyEvent*, which are generated periodically by the *EventDispatcher* instances described in Section 5.2.

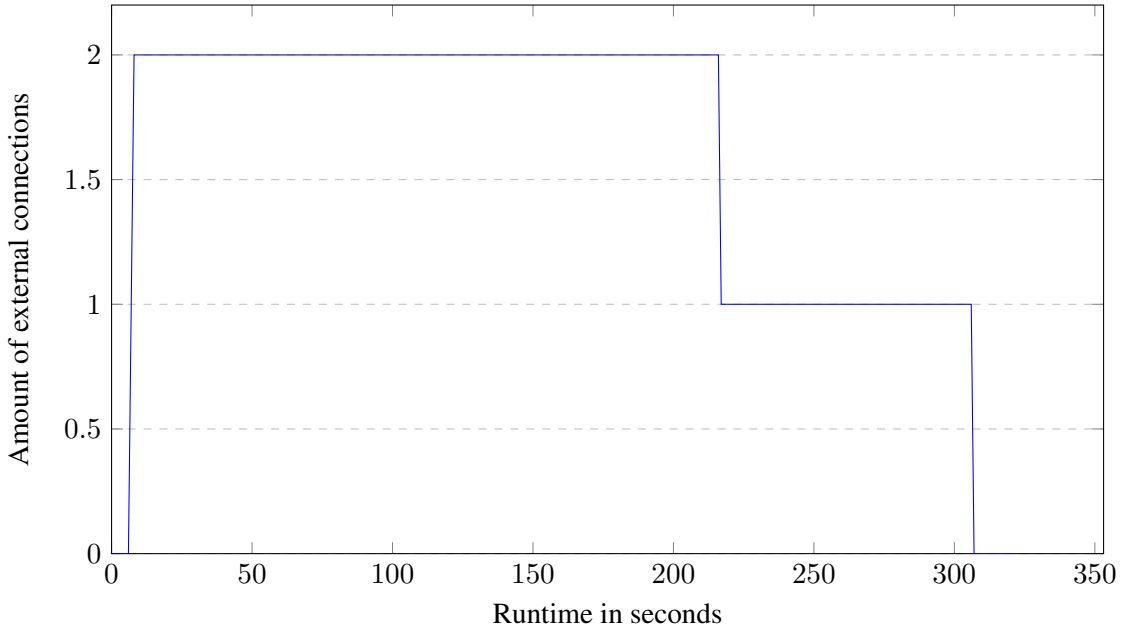


Figure 6.7: The amount of active external connections over the runtime of the application.

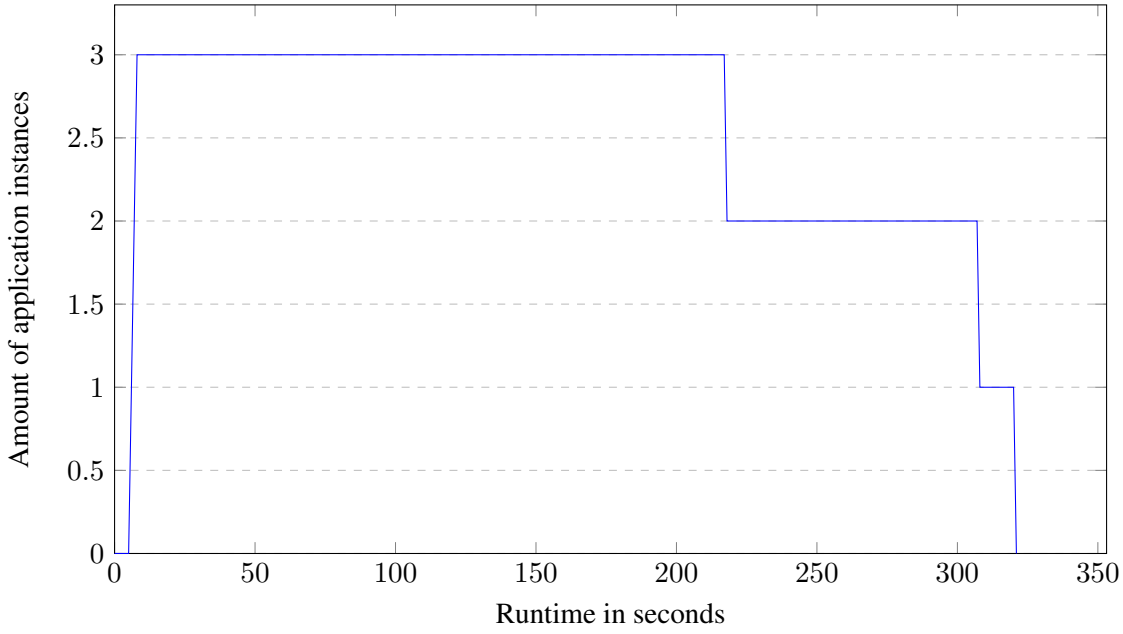


Figure 6.8: The amount of instantiated application instances over the runtime of the application.

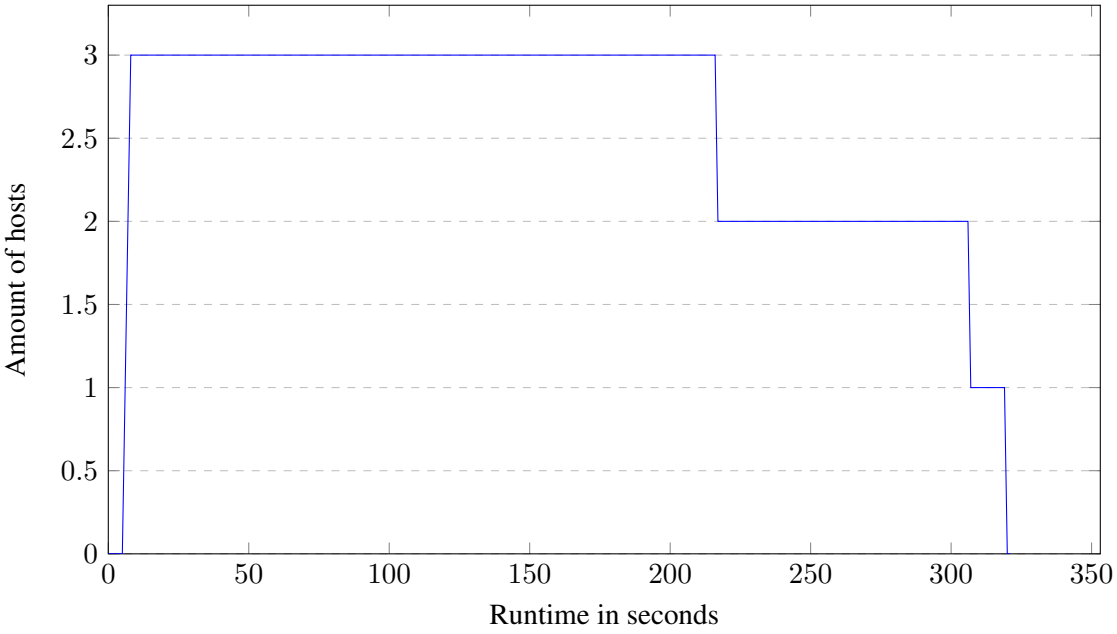
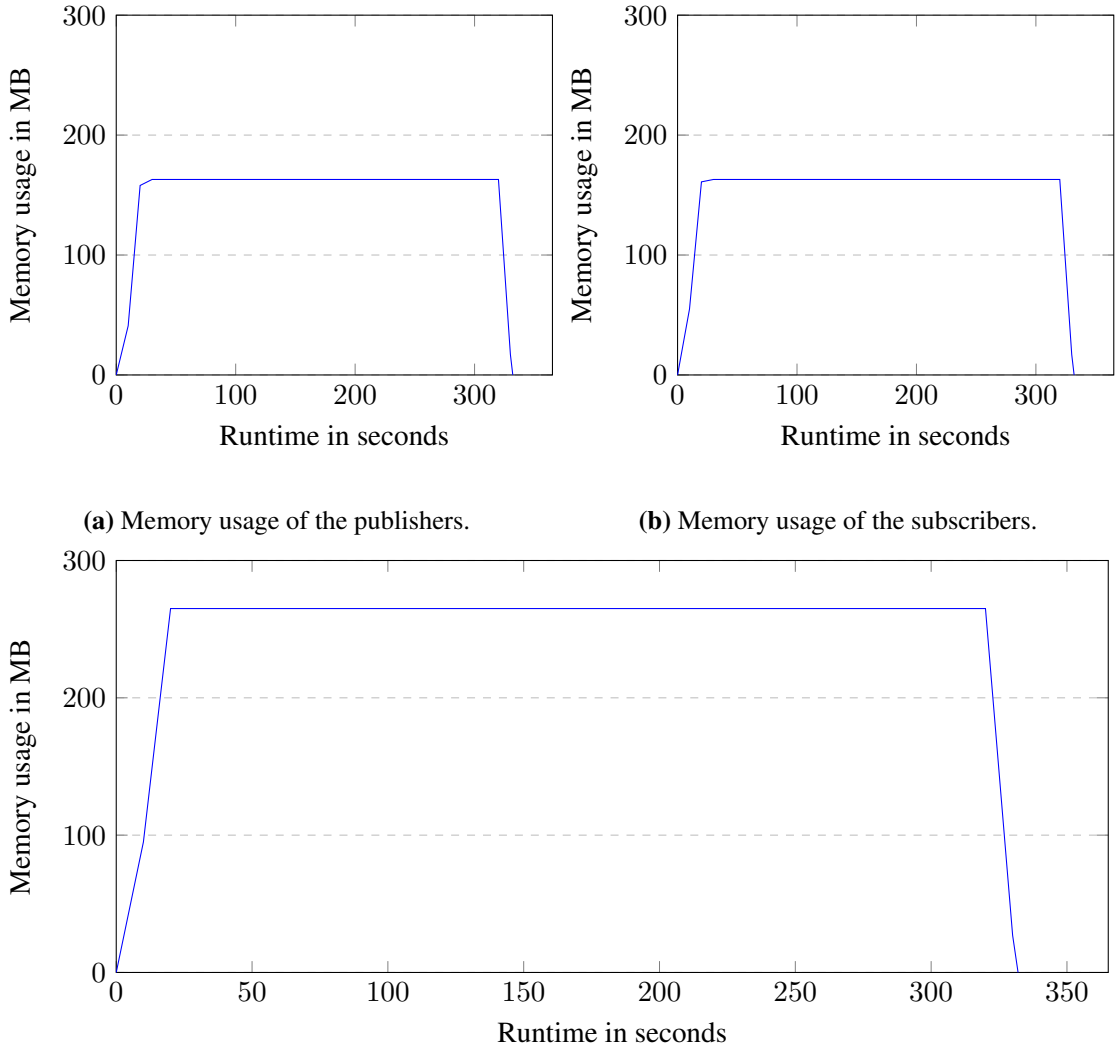


Figure 6.9: The amount of hosts that run parts of the application over it's runtime.

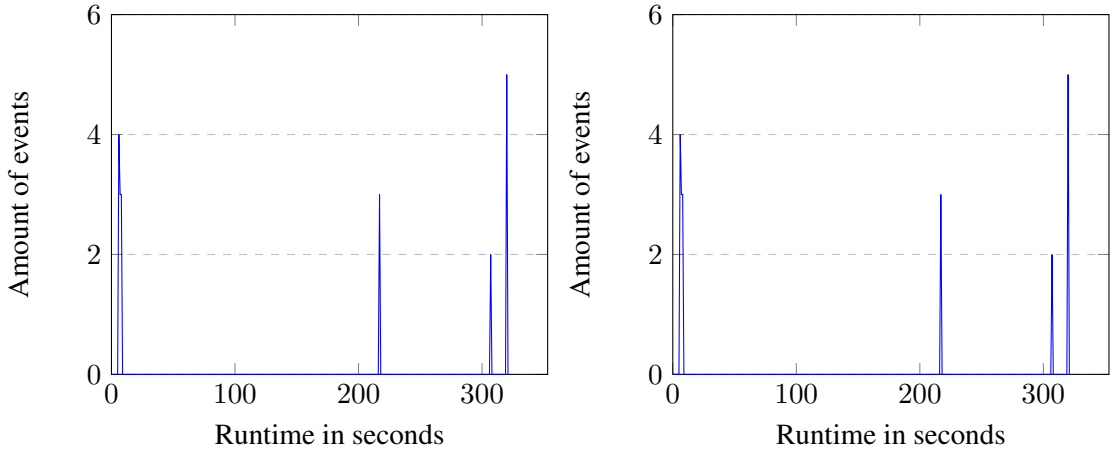


(a) Memory usage of the publishers.

(b) Memory usage of the subscribers.

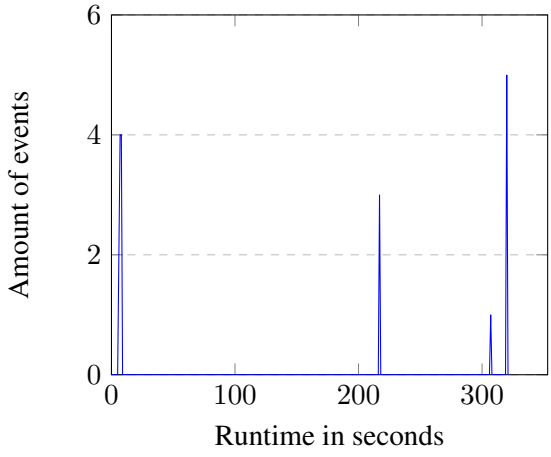
(c) Memory usage of the message broker.

Figure 6.10: The memory usage for each of the uses hosts in our evaluation application over time.

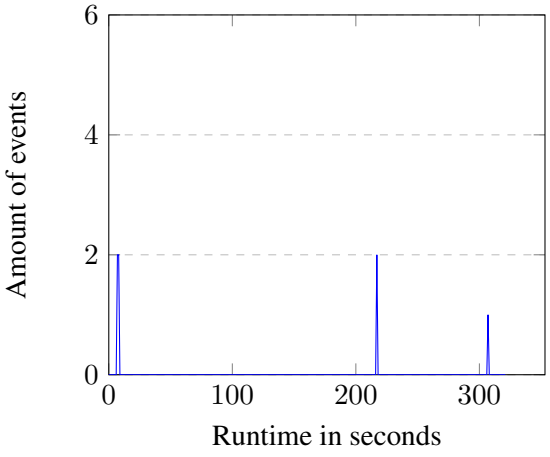


(a) Received *XADLEvent*s.

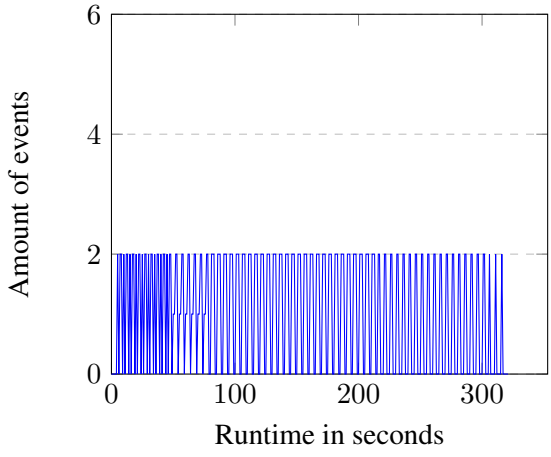
(b) Received *XADLRuntimeEvent*s.



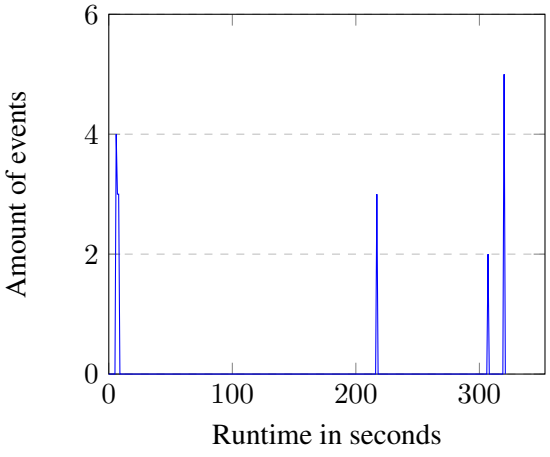
(c) Received *XADLLinkEvent*s.



(d) Received *XADLExternalLinkEvent*s.



(e) Received *XADLHostPropertyEvent*s.



(f) Received *XADLHostingEvent*s.

Figure 6.11: The different events received by the *Aggregator* over time.

6.4.2 Scenario 2

The second scenario described in Section 6.2.1 is an extension of our first audio streaming scenario. Here we use multiple publishers and subscribers to transport different audio streams. Figure 6.12 again shows the combined design-time architecture of all applications including the hosts which are not part of the design-time architecture. Here we can see that two publisher- and three subscriber applications are started on one host respectively. Given these additional connections the design-time architecture of the message broker has not been changed as all connections are handled dynamically.

Again we have extracted the runtime architecture from the *Aggregator* once all applications were started which is shown in Figure 6.13. If we compare it to the design-time architecture we can once again see that all bricks have been created an linked as expected. The message broker has created two instances of the *PublisherEndpoint* connector and three instances of the *SubscriberEndpoint* connector dynamically. These dynamically created connectors show that currently our approach handles each connection with it's own connector. It is not possible to use one connector for multiple incoming or outgoing connections. Each publisher and subscriber have once again been connected to the message broker with links between the the bricks *Publisher* and *PublisherEndpoint* as well as between *SubscriberEndpoint* and *Subscriber*. These connections are again shown as dotted lines so we are able to distinguish them from locally created links. Our current implementation of publishers and subscribers does not allow the streaming of multiple audio streams via one application. The runtime status of each brick has again been appended to their description. We again have added the hosts the applications are running on to the Figure. Comparing them to the hosts outlined in the design-time architecture we can again see that the information about the hosts have been extracted correctly.

This comparison shows that our approach is working correctly with multiple instances of the same application part. It further proves the feasibility of our approach. As in Section 6.4.1 we will validate our custom statistics that were extracted by the *Aggregator*.

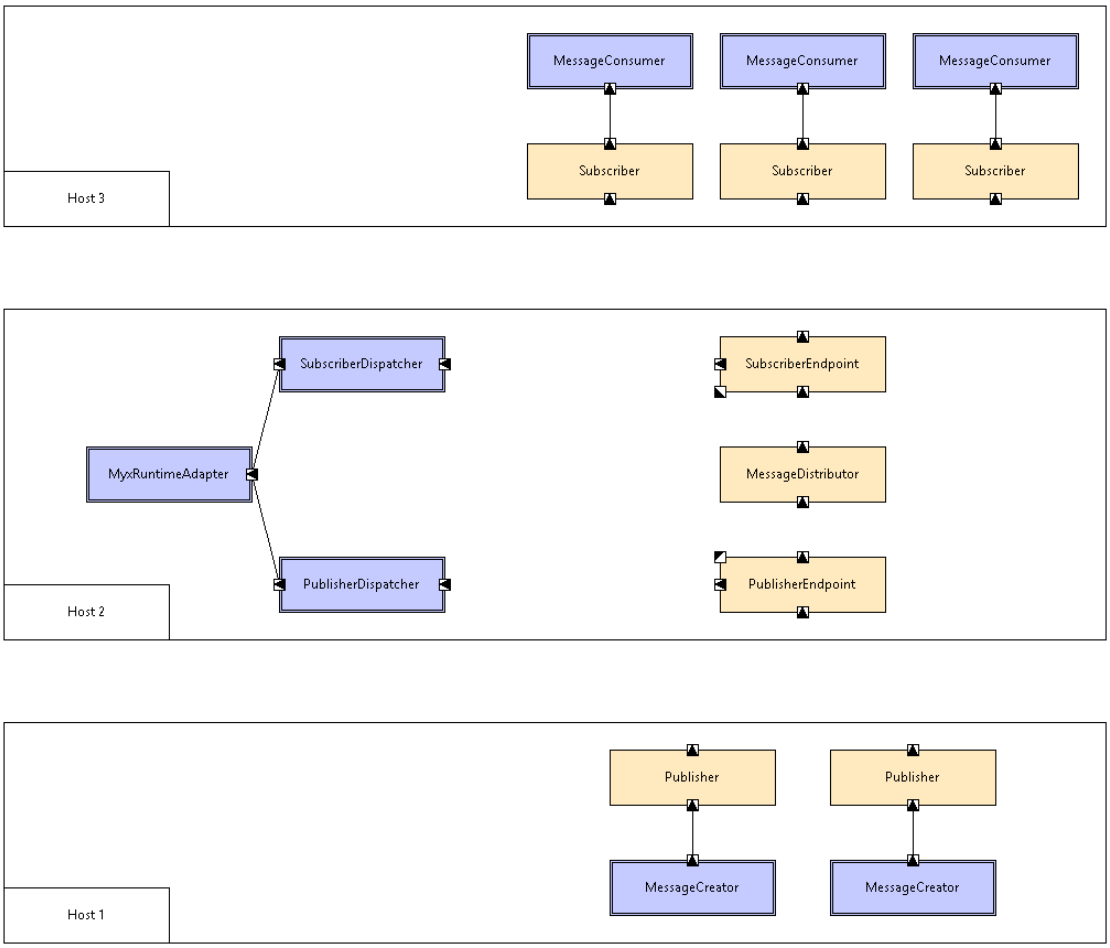


Figure 6.12: The design-time architecture of the audio-based publish-subscribe scenario using multiple publishers and subscribers.

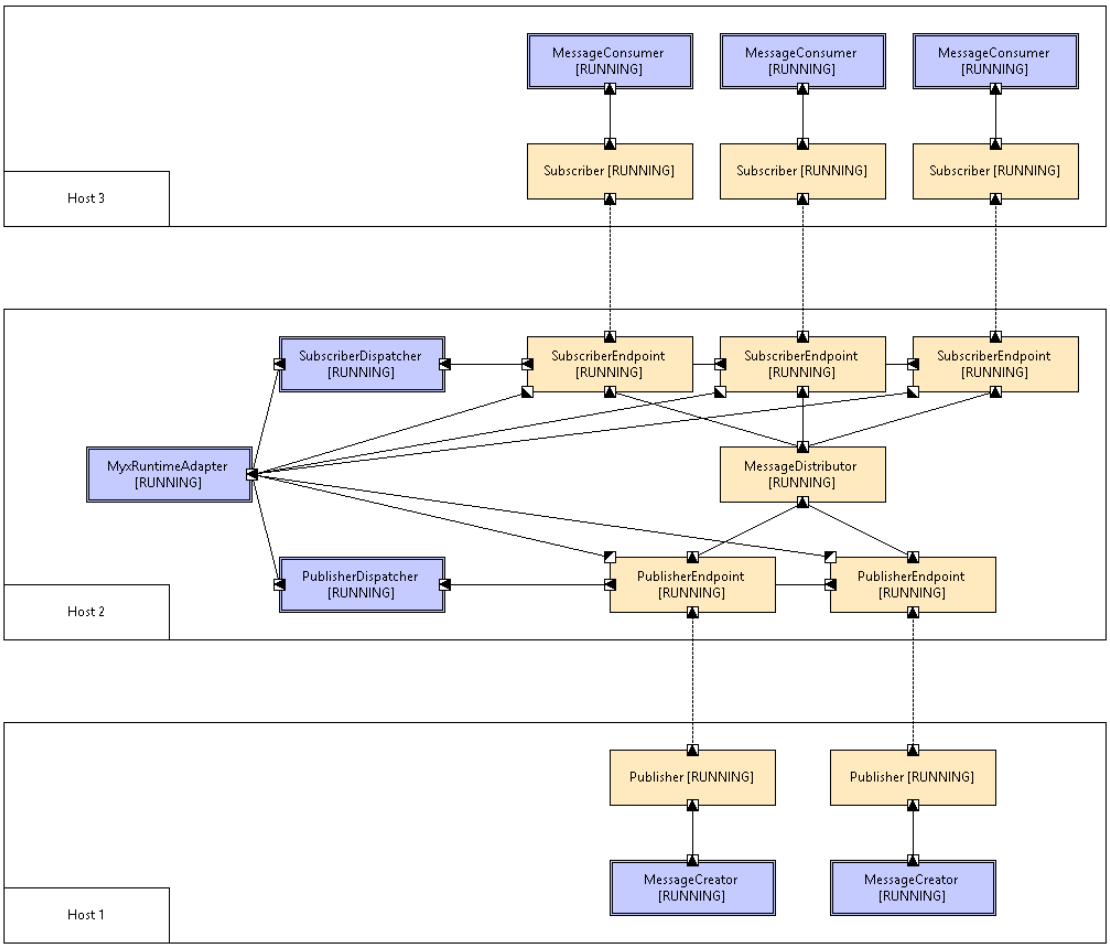


Figure 6.13: Resulting runtime architecture of the audio-based publish-subscribe scenario using multiple publishers and subscribers.

We can once again extract specific events that label changes in our runtime architecture. In the first step all applications are started. For us to be able to see the changes of the runtime architecture more clearly we delay the startup of each subscriber and publisher. At around 200 seconds the publishers have all finished streaming their used audio file. After 300 seconds all subscribers have finished replaying the streamed audio file, thus shutting themselves down. As both of our audio files have a length of around 5 minutes the runtime of publishers and subscribers does not change. Some time later we again shut down the message broker manually labeling the end of our second evaluation run.

We have thus described four major events in our evaluation run which are associated with changes in our runtime architecture. These changes can be shown using our custom statistics:

1. Application startup.
    - The amount of instantiated bricks is increased over time to 20 (see Figure 6.14), which is the correct amount of bricks for our running evaluation application.
    - The amount of external connections is increased to five (see Figure 6.15). These represent the connections between the publisher- and subscriber instances to the message broker.
    - The amount of instantiated application instances increases to six (see Figure 6.16). These instances represent the instantiated message broker, publisher- and subscriber instances.
    - The amount of hosts that run parts of our application increases to three (see Figure 6.9). Due to the fact that all instances of publishers and subscribers are run on a single host respectively this amount does not change in comparison of the previous scenario.
2. Publisher shutdown at around 200 seconds of runtime.
    - At this time we can see that the amount of instantiated bricks drops to 13. All publisher instances as well as their *VirtualPublisherEndpoint* bricks at the message broker are shut down.
    - Due to the shutdown the amount of external connections drops to three.
    - The amount of instantiated applications drops to four.
    - The amount of hosts running parts of the application drops to two.
3. Subscriber shutdown at around 300 seconds of runtime.
    - With the shutdown of the subscriber instances we can see that the amount of instantiated bricks drops to seven. Note that the *VirtualSubscriberEndpoint* bricks are not shut down due to the asynchronous architecture.
    - Due to the shutdown the amount of external connections drops to zero.
    - The amount of instantiated applications parts as well as hosts running them drops to one respectively.

4. Message broker shutdown at around 330 seconds of runtime.

    - After the message broker is shut down the whole evaluation application is shut down, dropping the amount of instantiated bricks, external connections, application instances and hosts to zero.

The extracted statistics again prove the feasibility of our approach. We also prove that our handling of hosts is done correctly independent of the amount of applications running on one host.

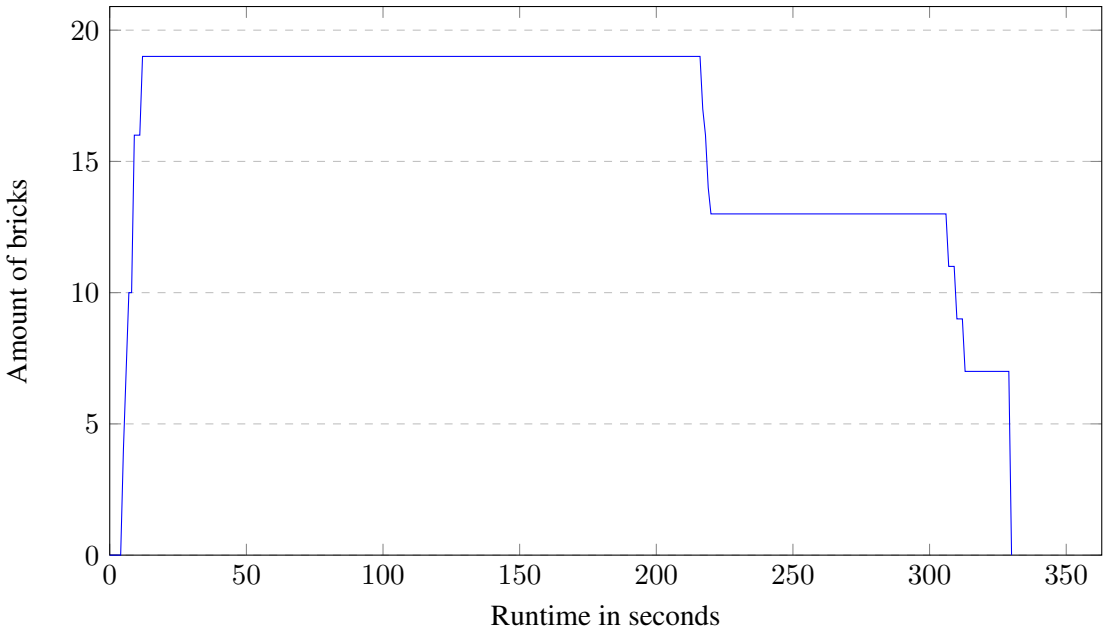


Figure 6.14: The amount of instantiated bricks over the runtime of the application.

Again we show the average memory usage of each host whilst running the evaluation scenario, see Figure 6.18. Here we can see that the increased amount of publishers and subscribers yields a higher memory usage for the message broker.

As in Section 6.4.1 we show the used events in Figure 6.19 which are again categorized by their type.

We can see a general increase of received events resulting from more monitored applications. As in the previous scenario we can see that most events were received once the state of the application is changed. Due to the startup delay we can see that the events created at the startup and destruction of an application is distributed over the startup delay itself.

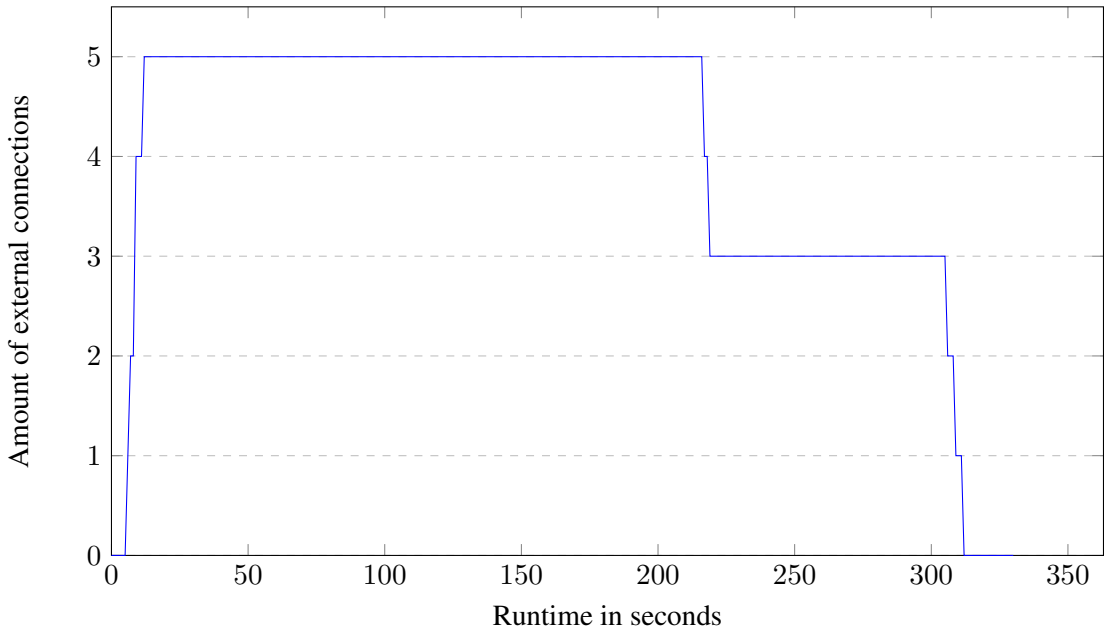


Figure 6.15: The amount of active external connections over the runtime of the application.

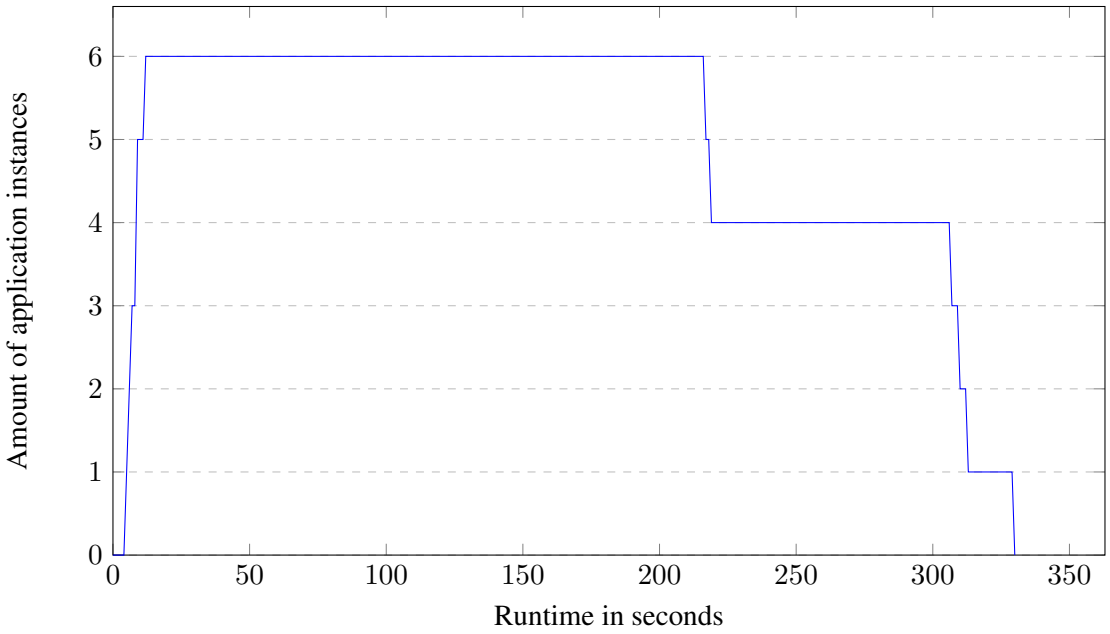


Figure 6.16: The amount of instantiated application instances over the runtime of the application.

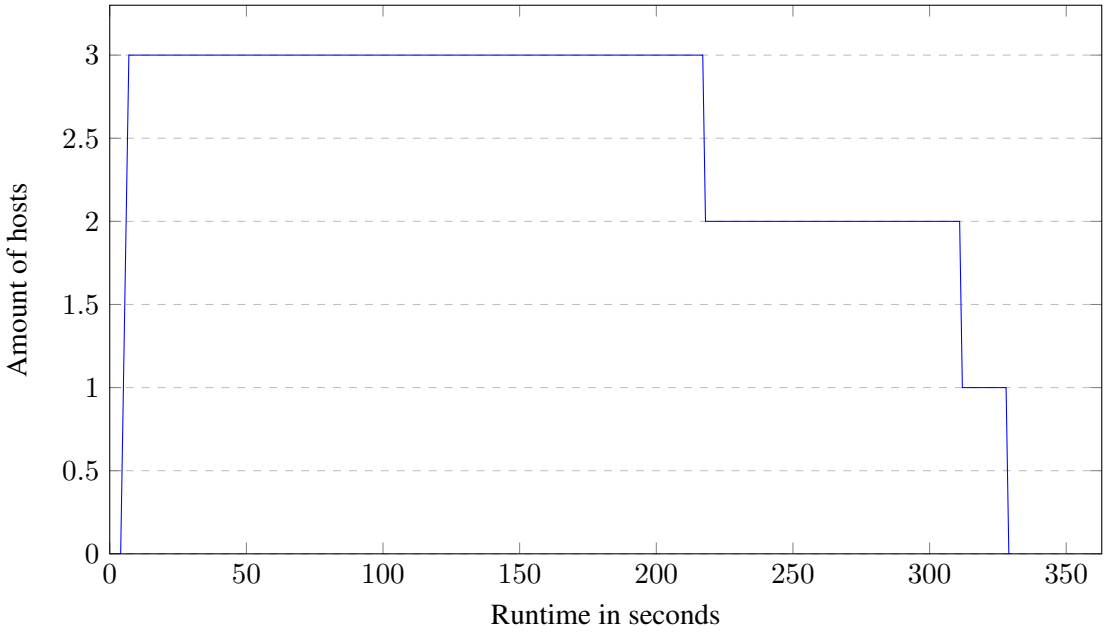
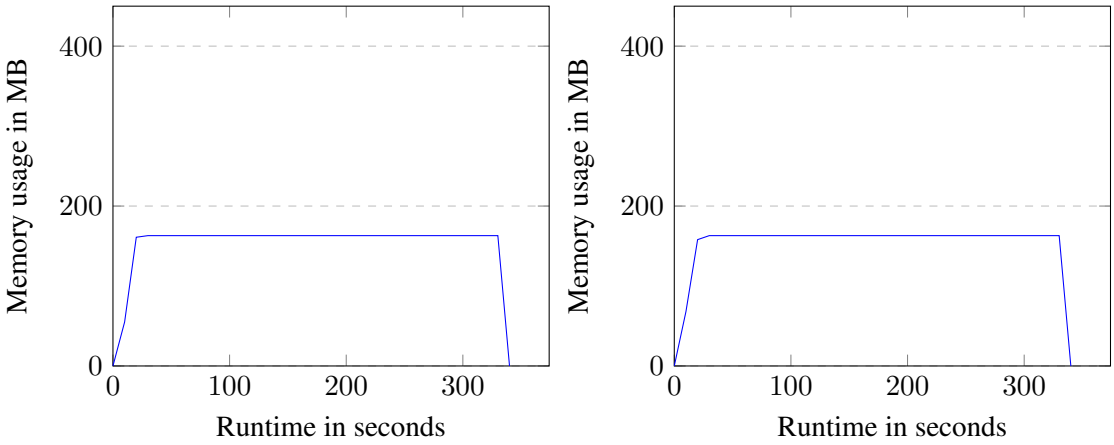
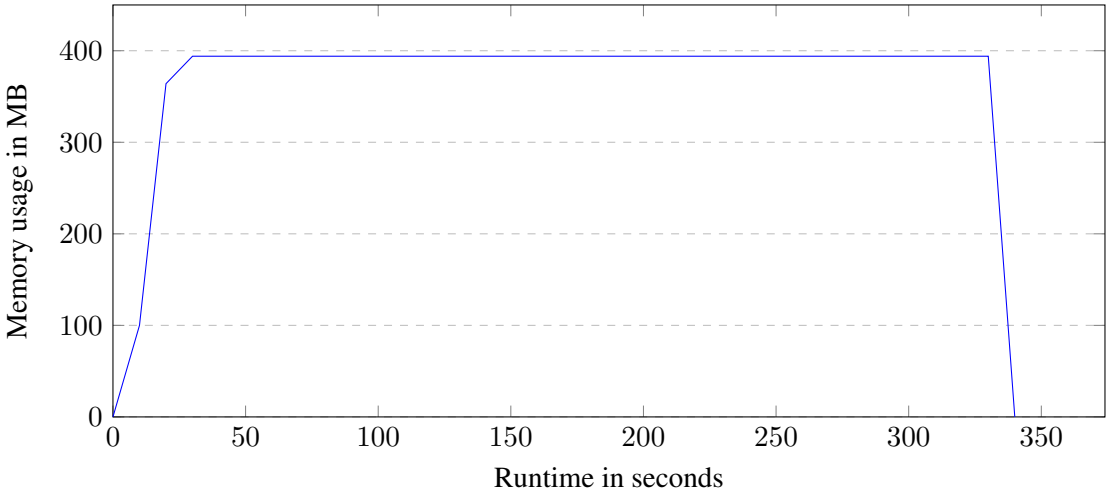


Figure 6.17: The amount of hosts that run parts of the application over it's runtime.



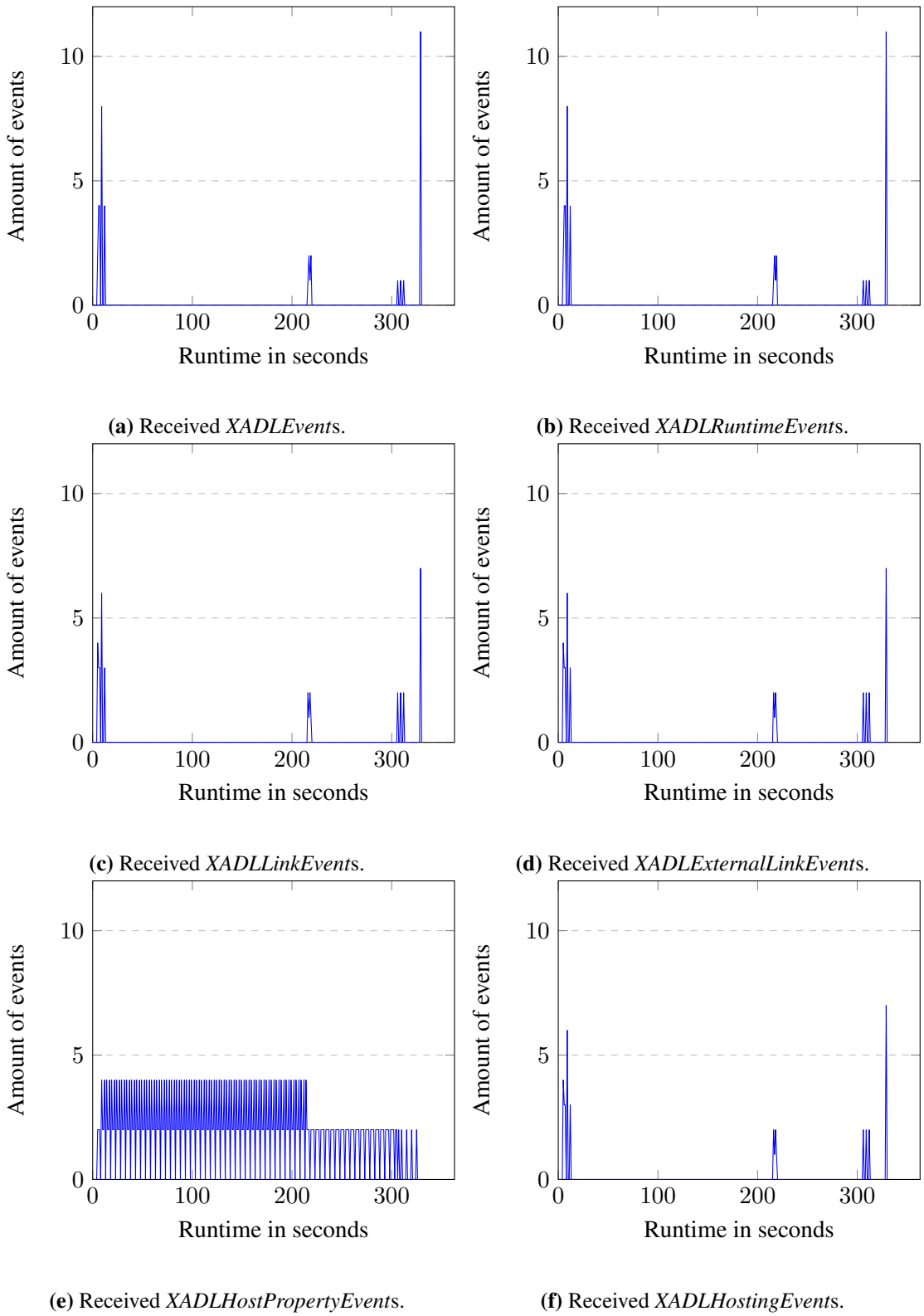
(a) Memory usage of the publishers.

(b) Memory usage of the subscribers.



(c) Memory usage of the message broker.

Figure 6.18: The memory usage for each of the uses hosts in our evaluation application over time.



(a) Received *XADLEvent*s.

(b) Received *XADLRuntimeEvent*s.

(c) Received *XADLLinkEvent*s.

(d) Received *XADLExternalLinkEvent*s.

(e) Received *XADLHostPropertyEvent*s.

(f) Received *XADLHostingEvent*s.

Figure 6.19: The different events received by the *Aggregator* over time.

6.5 Evaluation of Performance

After proving the feasibility of our approach we will show that it works well for distributed architectures by executing the scenarios described in Section 6.2.3 and again validate the extracted results.

Because of the amount of instantiated application parts we have omitted the visualization of the design-time- and resulting runtime architecture once all application parts are running. The validation will solely be based on our custom statistics extracted by the *Aggregator*.

By comparing the scenarios and the results described in Section 6.4 we can predict some of the results to be extracted. Due to the fact that the scenarios have a long runtime we will not be able to see single events in the resulting diagrams, we will only be able to see the startup and shutdown of application parts. Due to amount of transferred events we have aggregated the resulting statistics and used the mean values that have been extracted over ten seconds of runtime.

6.5.1 Scenario 1

The first scenario described in Section 6.2.3 executes both publishers and subscribers with the same maximum runtime. This means we should be able to see periodic architectural changes, that is each time the publishers and subscribers are started up or shut down. The different ramp-up times should yield a slower startup of application instances. But due to the maximum runtime, that publishers and subscribers share, we should see that the shutdown of all instances happens within a small time window. Because we start these application instances 20 times we should be able to observe this behavior 20 times.

We can again describe all the events that cause changes in the runtime architecture that were used in the previous scenarios. But due to the length of our evaluation run not all of these events are clearly shown in our statistics. We thus describe only four major events that cause the runtime architecture to change significantly:

1. Message broker startup.
    - The amount of instantiated bricks is increased to four.
    - The amount of external connections remains at zero.
    - The amount of instantiated application instances increases to one.
    - The amount of hosts that run parts of our application increases to one.
2. Publisher and subscriber startup
    - The amount of instantiated bricks is increased to 1204 (see Figure 6.20). If we compare this number to the scenario description we can see that this number is correct, that is there are four bricks created by the message broker, two bricks for each of the 200 publishers and subscriber and one is dynamically created for each publisher and subscriber that connects to the message broker.

- The amount of external connections is increased correctly to 400 (see Figure 6.21) representing the connections between publishers and subscribers to the message broker.
- The amount of instantiated application instances increases to 401 (see Figure 6.22).
- The amount of hosts that run parts of our application increases to three (see Figure 6.23).

3. Publisher and subscriber shutdown

   - With the shutdown of the publishers and subscribers the number of bricks decreases to four, leaving only the message broker running.
   - The amount of external connections decreases to zero.
   - The amount of application instances and hosts running parts of our application decrease to one, that is the message broker.

4. Application shutdown.

   - After all publishers, subscribers and the message broker are shut down the amount of instantiated bricks, external connections, application instances and hosts drops to zero.

The startup and shutdown of publishers and subscribers are executed exactly 20 times as the scenario descriptions stated. Because publishers and subscribers are executed separately from each other the startup and shutdown of application instances begins to overlap over time, thus the amount of instantiated bricks, external connections, application instances and hosts does not decrease to the lowest possible value.

To show how our approach is performing with many publishers and subscribers we again show the average memory usage of each host, see Figure 6.24. If we compare the memory usage to the previous two scenarios all hosts require by far a higher amount of memory. Yet with the used number of publishers and subscribers the memory usage of our message broker is smaller than expected. As in the other figures we are able to see when the publisher or subscribers have been started or shut down due to the flapping memory usage.

If we take a closer look at the transmitted events, shown in Figure 6.25 we can see that the startup and shutdown of publishers and subscribers is quite different. Due to the ramp-up time we use to start up all instances we can see that events are transmitted over time, whilst the shutdown on the other hand happens instantly thus transmitting many events in a short period of time. For the evaluated scenario a total number of 405 816 events were received by the aggregation component leading to an average of 308 events per second. The majority of those are *XADLHostPropertyEvent*s that are received continuously. Because each application instances is transmitting these events the number of events increases with each additional application that is running on a host which is already hosting a monitored application. A total of 169 802 *XADLHostPropertyEvent*s were received. If we reduce the number of transmitted *XADLHostPropertyEvent*s in a way that there is only one sender per host would result in a total number of only 528 transferred *XADLHostPropertyEvent*s which would reduce the total number

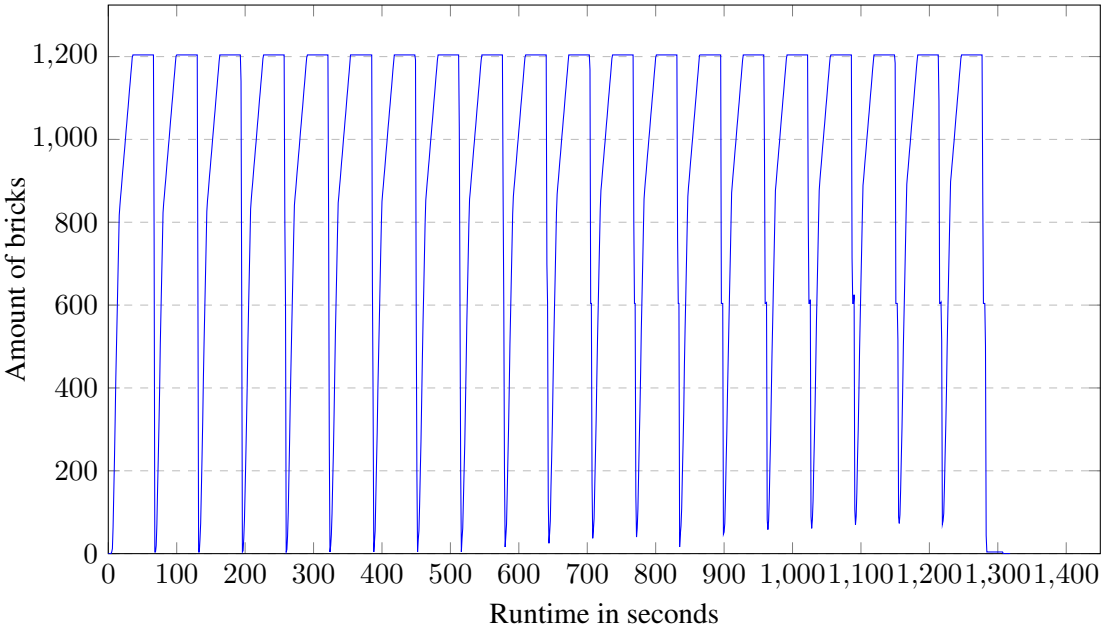


Figure 6.20: The amount of instantiated bricks over the runtime of the application.

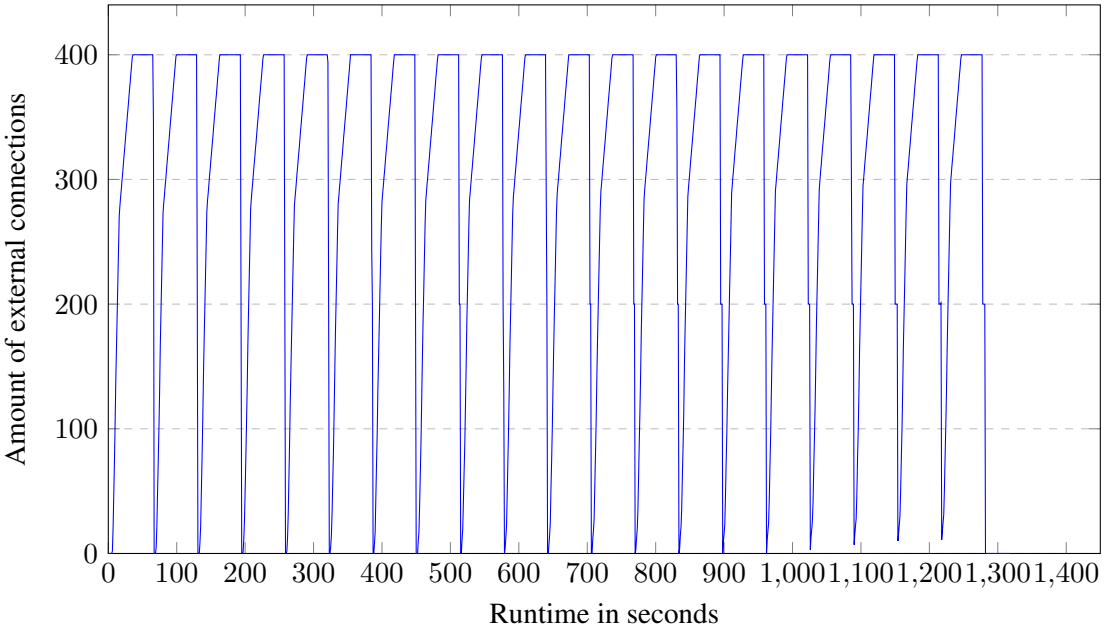


Figure 6.21: The amount of active external connections over the runtime of the application.

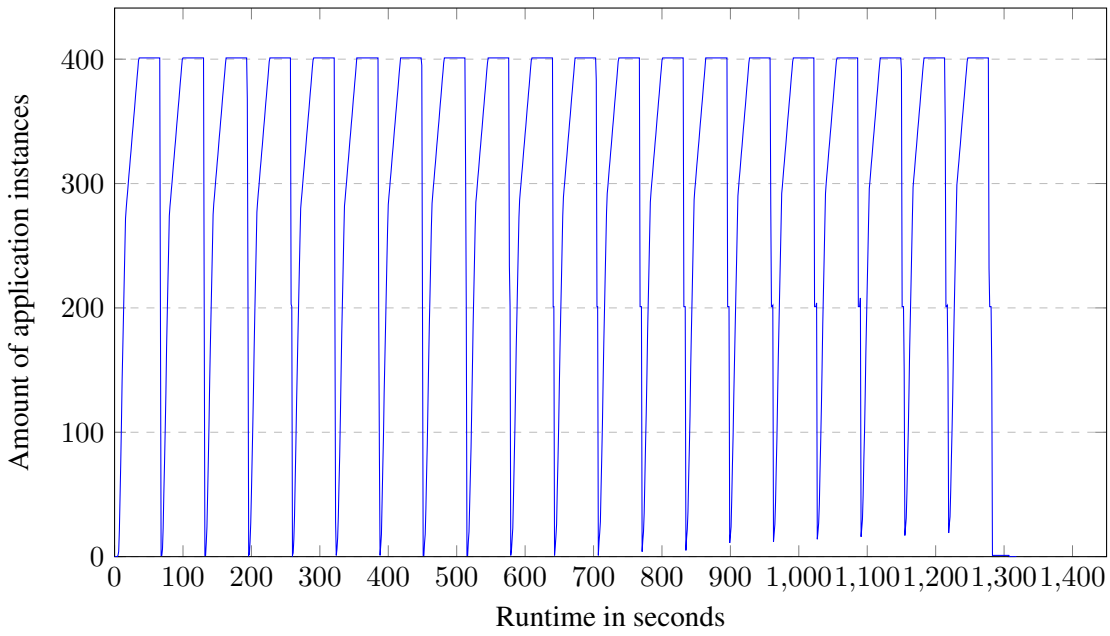


Figure 6.22: The amount of instantiated application instances over the runtime of the application.

of received events to 236 542. This leads to an average of only 180 per second. All of this could reduce the number of received events and thus network usage by over 40%.

As the *Aggregator* extracts general statistics about all received events which also includes the received events over the application's runtime we are able to perform these calculations.

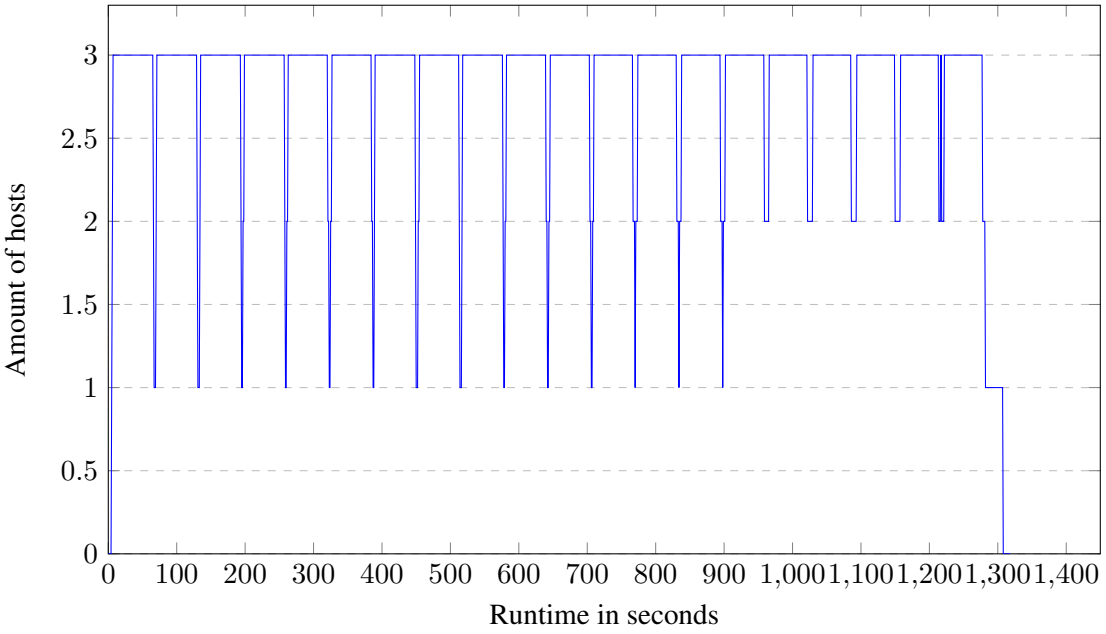
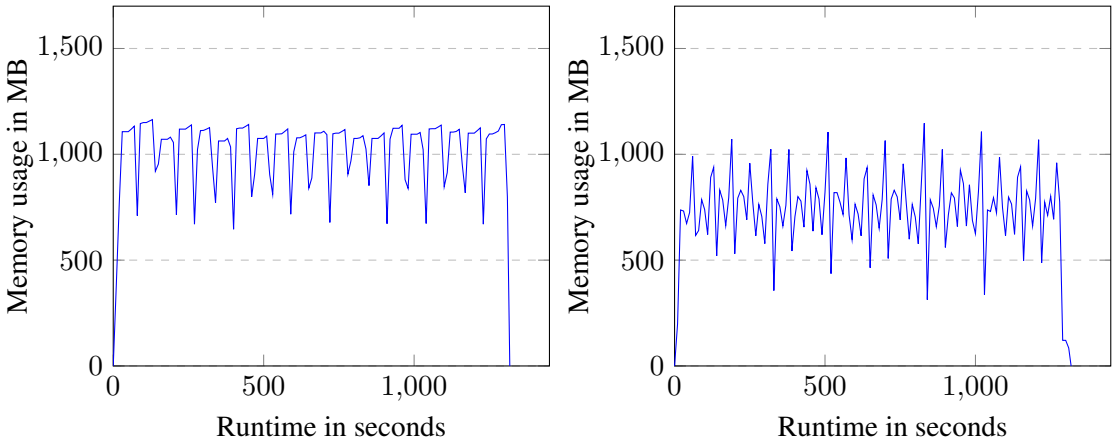
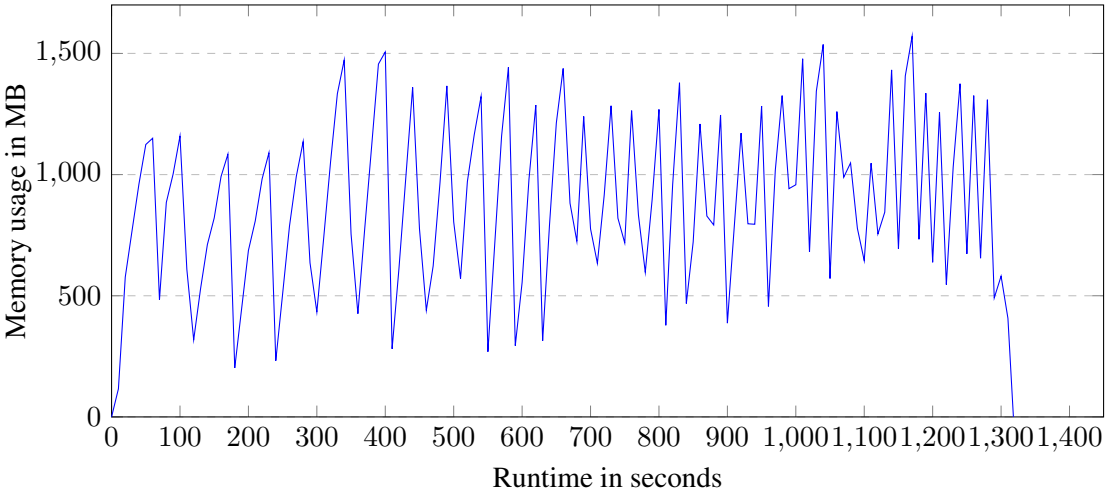


Figure 6.23: The amount of hosts that run parts of the application over it's runtime.



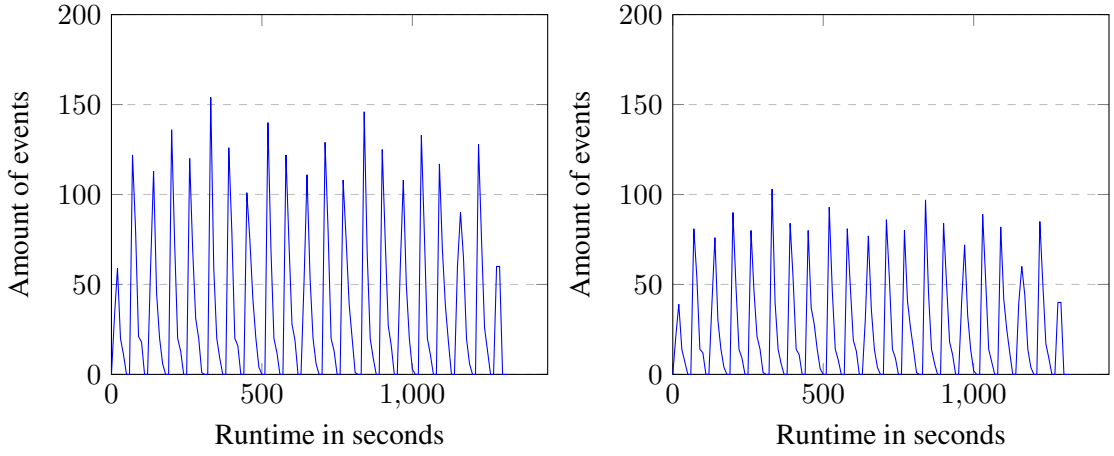
(a) Memory usage of the publishers.

(b) Memory usage of the subscribers.



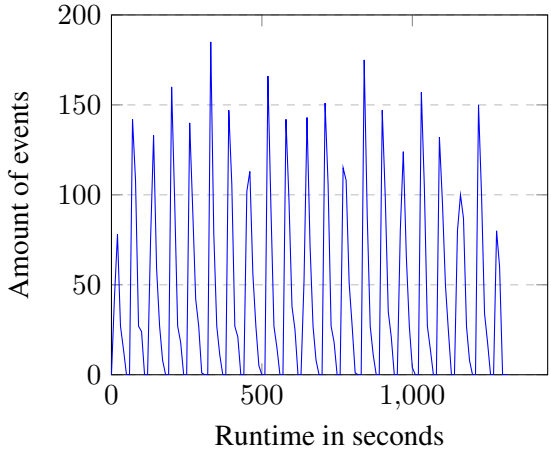
(c) Memory usage of the message broker.

Figure 6.24: The memory usage for each of the uses hosts in our evaluation application over time.

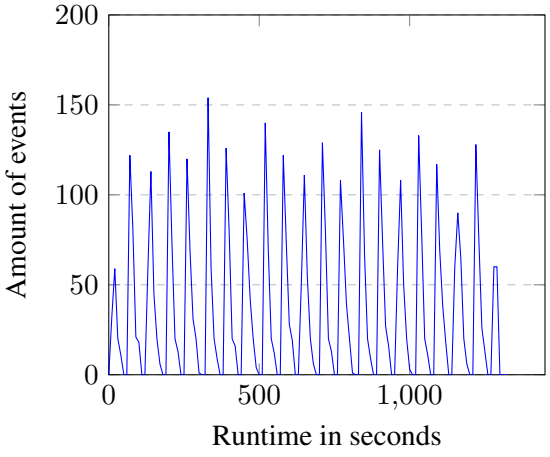


(a) Received *XADLEvent*s.

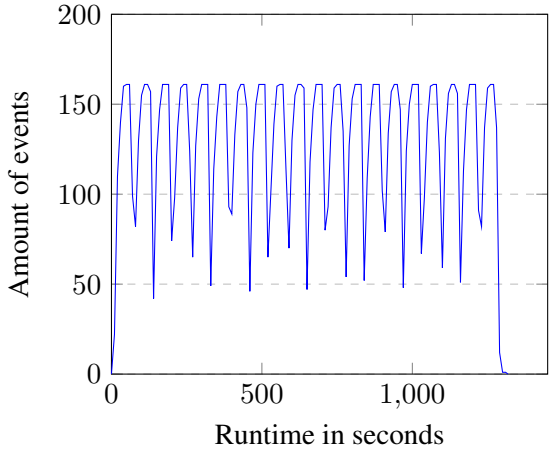
(b) Received *XADLRuntimeEvent*s.



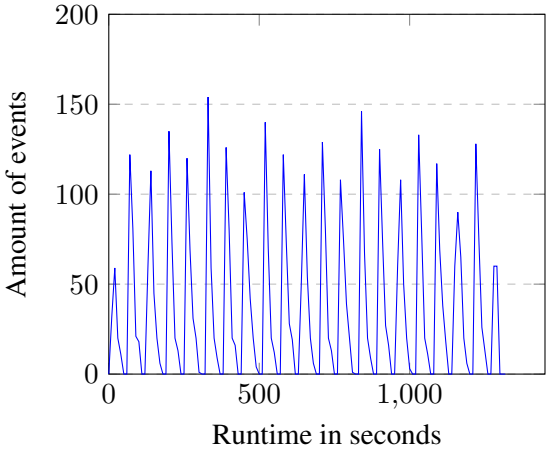
(c) Received *XADLLinkEvent*s.



(d) Received *XADLExternalLinkEvent*s.



(e) Received *XADLHostPropertyEvent*s.



(f) Received *XADLHostingEvent*s.

Figure 6.25: The different events received by the *Aggregator* over time.

6.5.2 Scenario 2

The seconds scenario described in Section 6.2.3 resembles the first one in nearly all properties. The only change is that we have added some randomness to the ramp-up time of the startup process of the publishers and subscribers. This leads to an evaluation run that is closer to a real-world application where not all applications start respectively shut down at the same time.

As before we can identify the four major events that cause significant changes to the runtime architecture:

1. Message broker startup.
    - The amount of instantiated bricks is increased to four.
    - The amount of external connections remains at zero.
    - The amount of instantiated application instances increases to one.
    - The amount of hosts that run parts of our application increases to one.
2. Publisher and subscriber startup
    - The amount of instantiated bricks is again increased to 1204 (see Figure 6.26), the amount of external connections increases to 401 (see Figure 6.27), the amount of instantiated application instances increases to 400 (see Figure 6.28) and the number of hosts running parts of our application increases to three (see Figure 6.29). Because we have not changed the number of publishers and subscribers these numbers are the same as in the previous scenario.
3. Publisher and subscriber shutdown
    - With the shutdown of the publishers and subscribers the number of bricks again decreases to four, leaving only the message broker running.
    - The amount of external connections decreases to zero.
    - The amount of application instances and hosts running parts of our application decrease to one, that is the message broker.
4. Application shutdown.
    - After all publishers, subscribers and the message broker are shut down the amount of instantiated bricks, external connections, application instances and hosts drops to zero.

Due to the randomness of the ramp-up time we can see that the startup and shutdown of publishers and subscribers are again overlapping over time. Thus yielding results that are closer to a real-world scenario for publish-subscribe systems.

The startup and shutdown of publishers and subscribers are executed exactly 20 times once more. Because publishers and subscribers are executed separately from each other and the added randomness the startup and shutdown of application instances begins to overlap earlier over time,

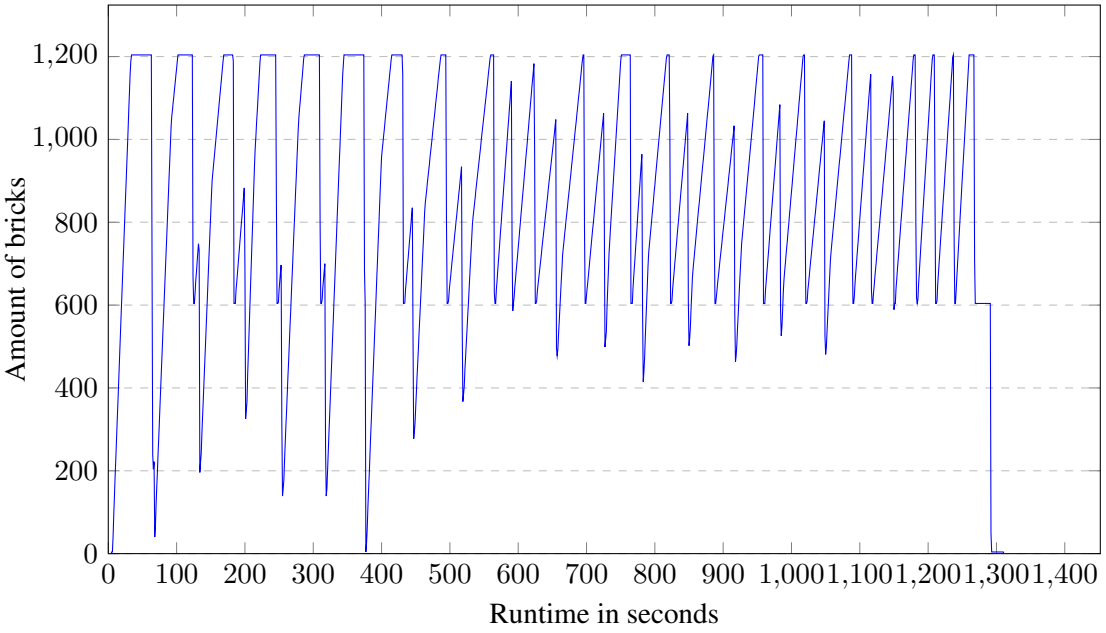


Figure 6.26: The amount of instantiated bricks over the runtime of the application.

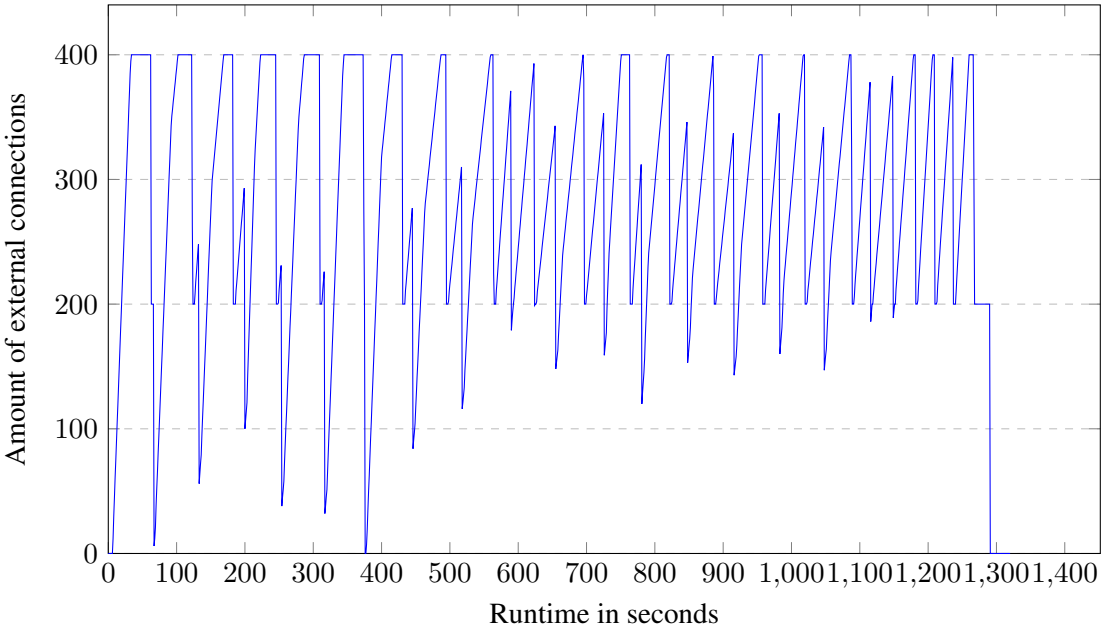


Figure 6.27: The amount of active external connections over the runtime of the application.

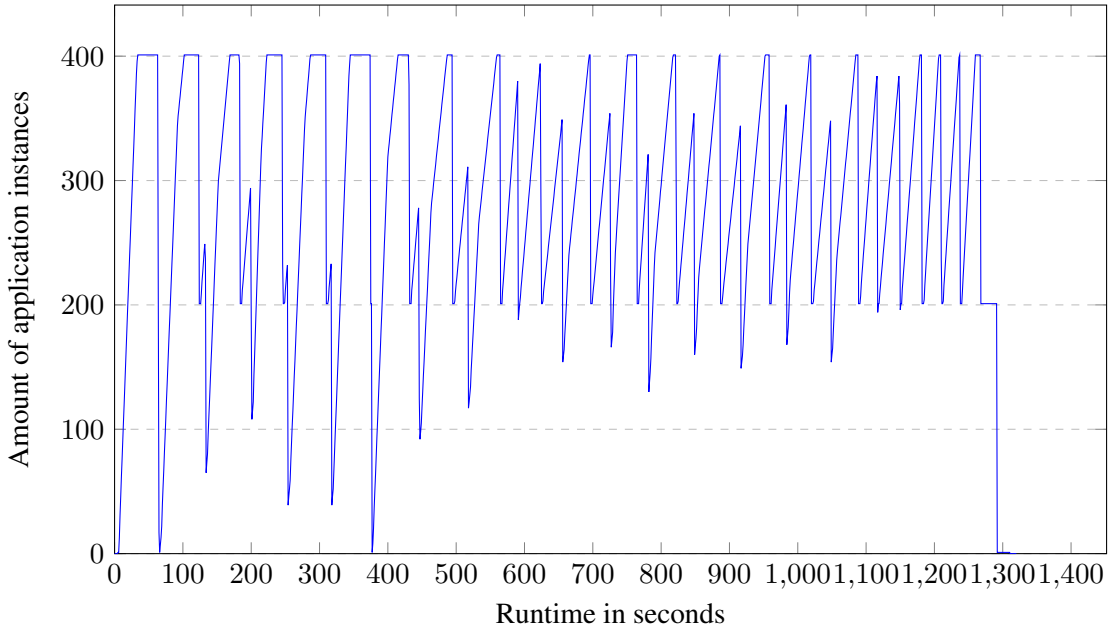


Figure 6.28: The amount of instantiated application instances over the runtime of the application.

again not reducing the amount of instantiated bricks, external connections, application instances and hosts to the lowest possible value.

Again we show the average memory usage of each host whilst running our evaluation scenario, see Figure 6.30.

The transmitted events show that the startup and shutdown of publishers and subscribers overlap with increased runtime as well, see Figure 6.31. The *Aggregator* received total number of 390 391 events for the evaluated scenario, yielding an average of 296 events per second. Once again most of the transmitted events were of type *XADLHostPropertyEvent*. If we can again reduce the amount of continuously transmitted events to one sender per host we would only receive around 236 556 events leading to an average of only 180 events per second. This once again reduces the amount of transmitted events by around 40%.

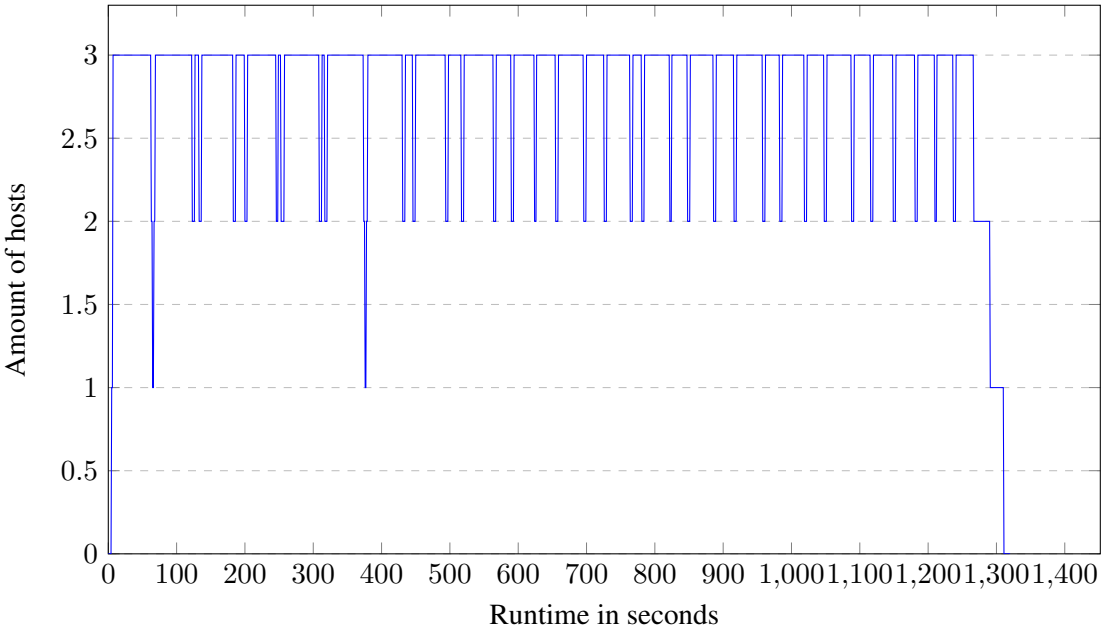
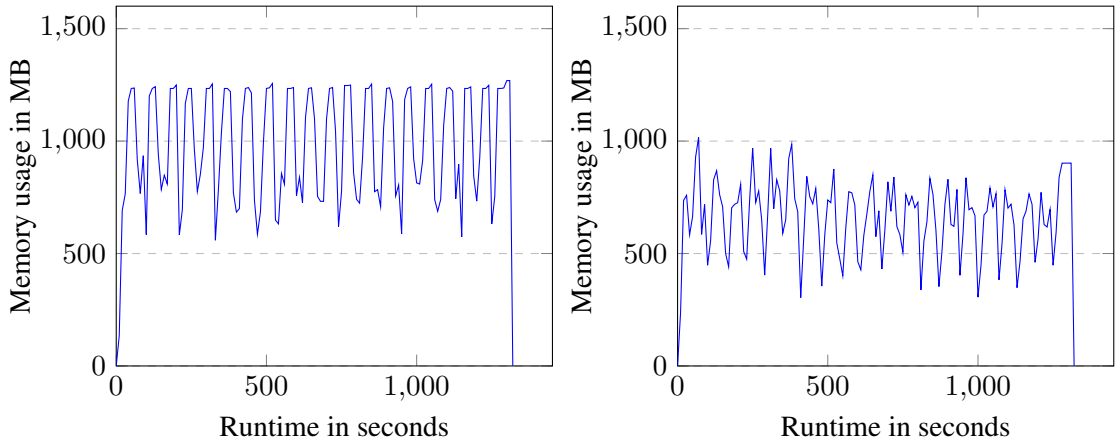
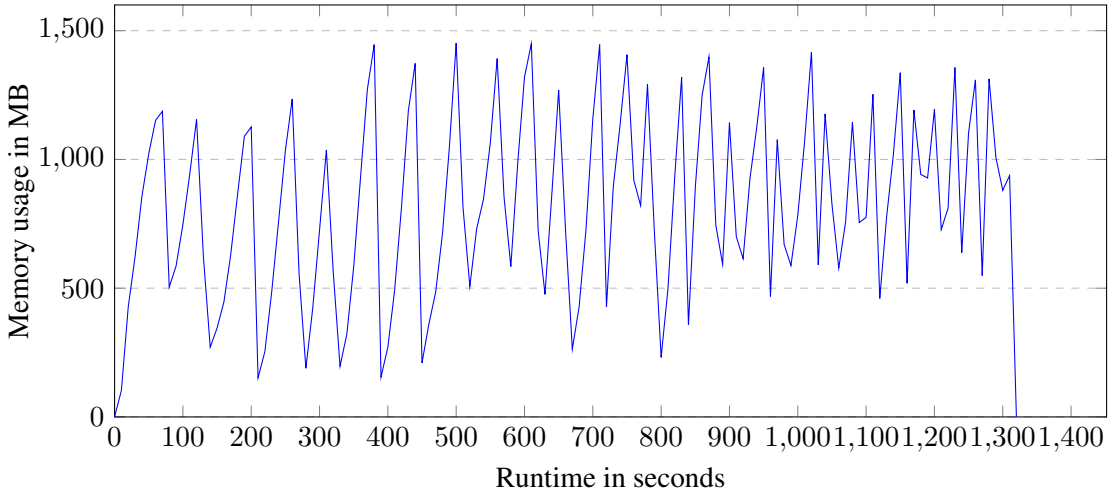


Figure 6.29: The amount of hosts that run parts of the application over it's runtime.



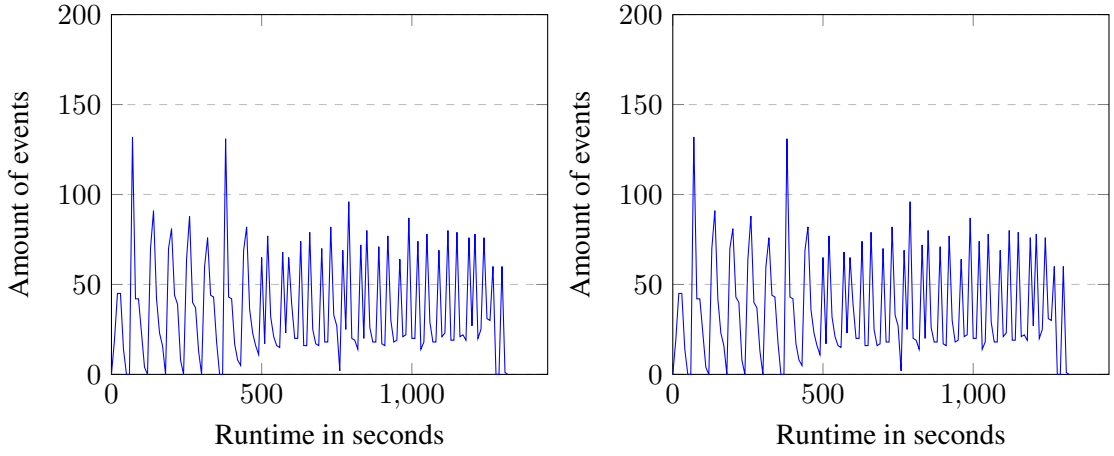
(a) Memory usage of the publishers.

(b) Memory usage of the subscribers.



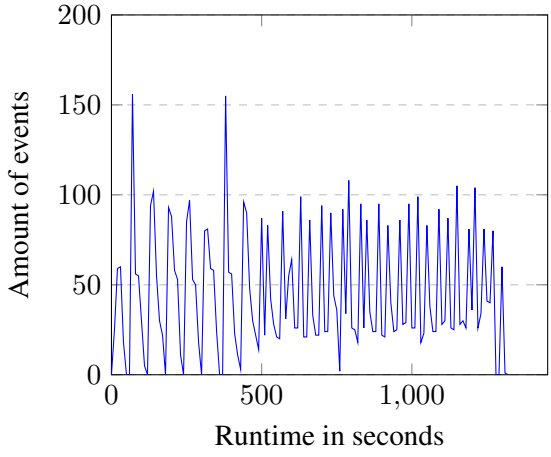
(c) Memory usage of the message broker.

Figure 6.30: The memory usage for each of the uses hosts in our evaluation application over time.

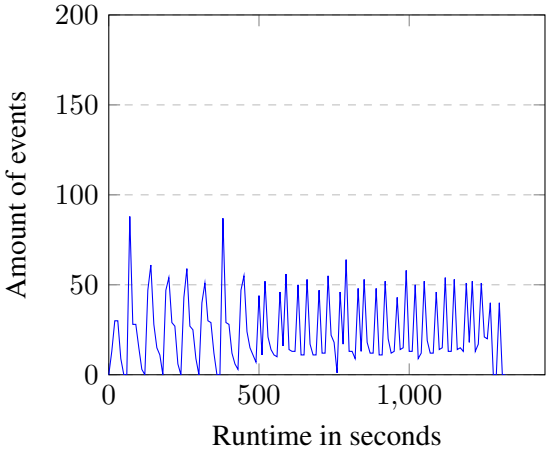


(a) Received *XADLEvent*s.

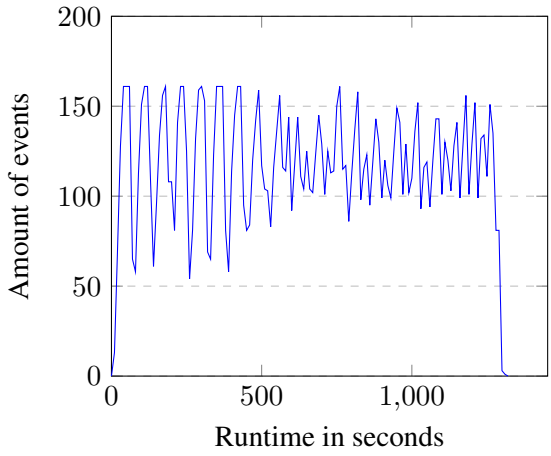
(b) Received *XADLRuntimeEvent*s.



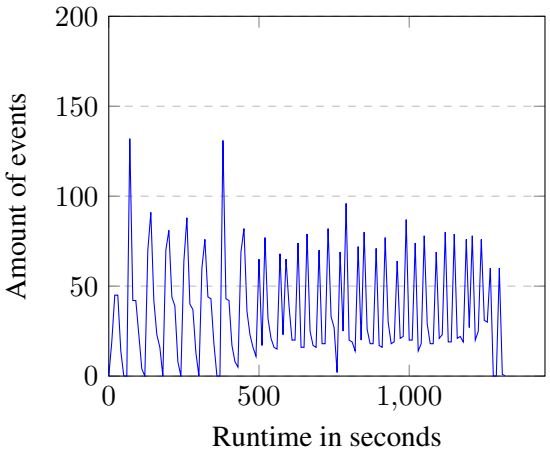
(c) Received *XADLLinkEvent*s.



(d) Received *XADLExternalLinkEvent*s.



(e) Received *XADLHostPropertyEvent*s.



(f) Received *XADLHostingEvent*s.

Figure 6.31: The different events received by the *Aggregator* over time.

6.5.3 Scenario 3

The third and last scenario described in Section 6.2.3 is fully randomized. That is we randomize not only the ramp-up time of publishers and subscribers but also the amount of instantiated instances as well as their runtime.

The previous two scenarios have shown that our approach is working for large distributed architectures. By randomizing all available options we can evaluate a rapidly changing architecture over it's runtime. These rapid changes are visualized in most of our extracted statistics like instantiated bricks (Figure 6.32), external connections (Figure 6.33) and application instances (Figure 6.34). Even the amount of hosts running application parts (Figure 6.35) shows these changes, but not as clearly as the other statistics. That is we have not changed the amount of hosts compared to the other two scenarios.

All statistics show that our approach is working feasible for rapidly changing distributed architectures. Because we have already discussed the major events that cause these rapid changes in the previous two scenarios they will be omitted here.

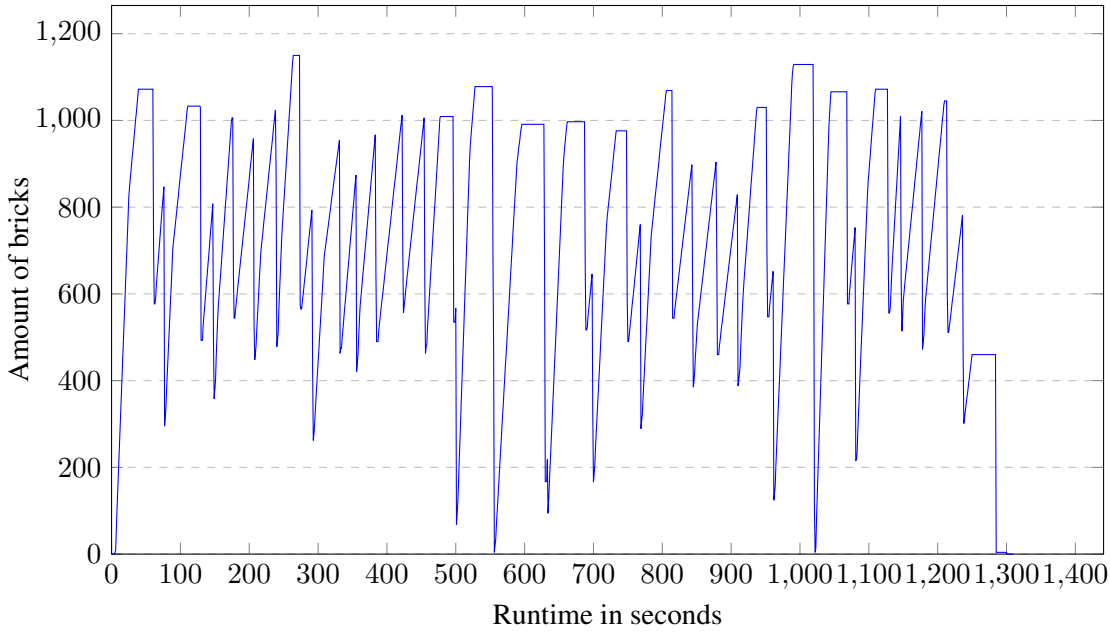


Figure 6.32: The amount of instantiated bricks over the runtime of the application.

Again we show the average memory usage of each host whilst running our evaluation scenario, see Figure 6.36.

If we take a closer look at the transmitted events one last time we can see that the rapid changes are also reflected as our statistics are based on these events, see Figure 6.37.

The *Aggregator* received a total number of 335 191 events for the evaluated scenario, yielding an average of 256 events per second. Again most of the transmitted events were of type *XADLHostPropertyEvent*. If we can reduce the amount of continuously transmitted events to

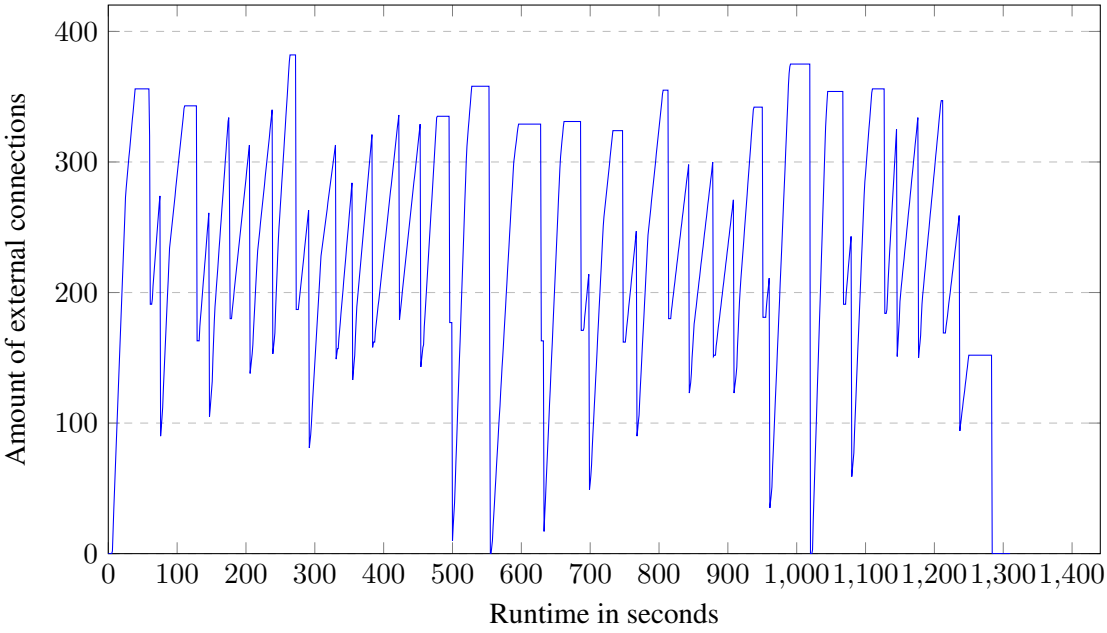


Figure 6.33: The amount of active external connections over the runtime of the application.

one sender per host we would only receive 204 262 events leading to an average of 156 events per second. This reduces the amount of transmitted events by around 40%, which is close to the savings extracted from the other scenarios.

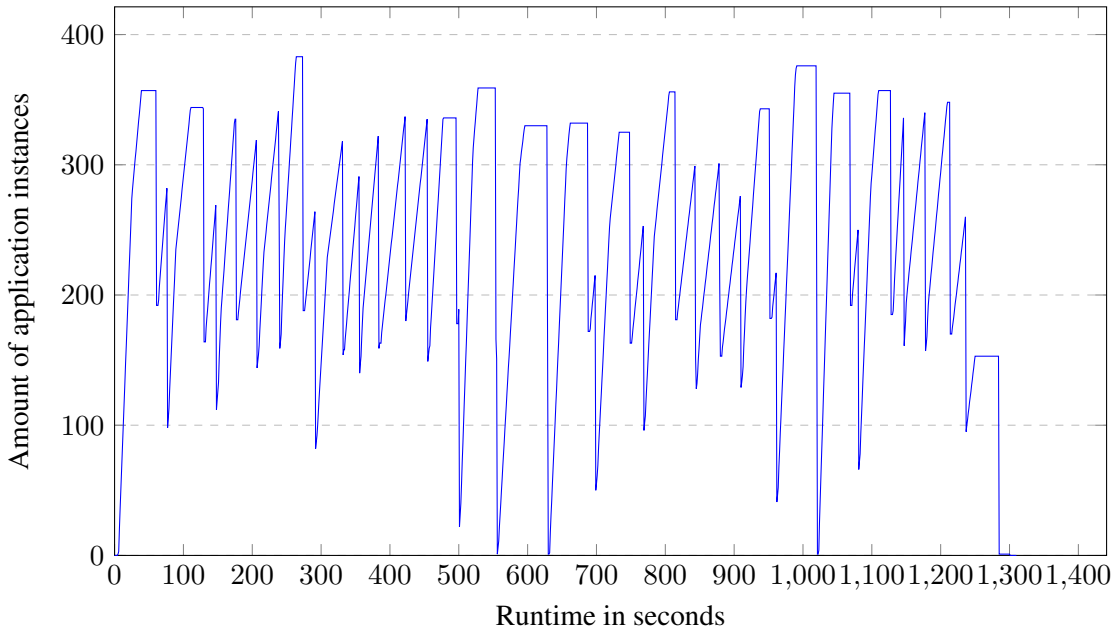


Figure 6.34: The amount of instantiated application instances over the runtime of the application.

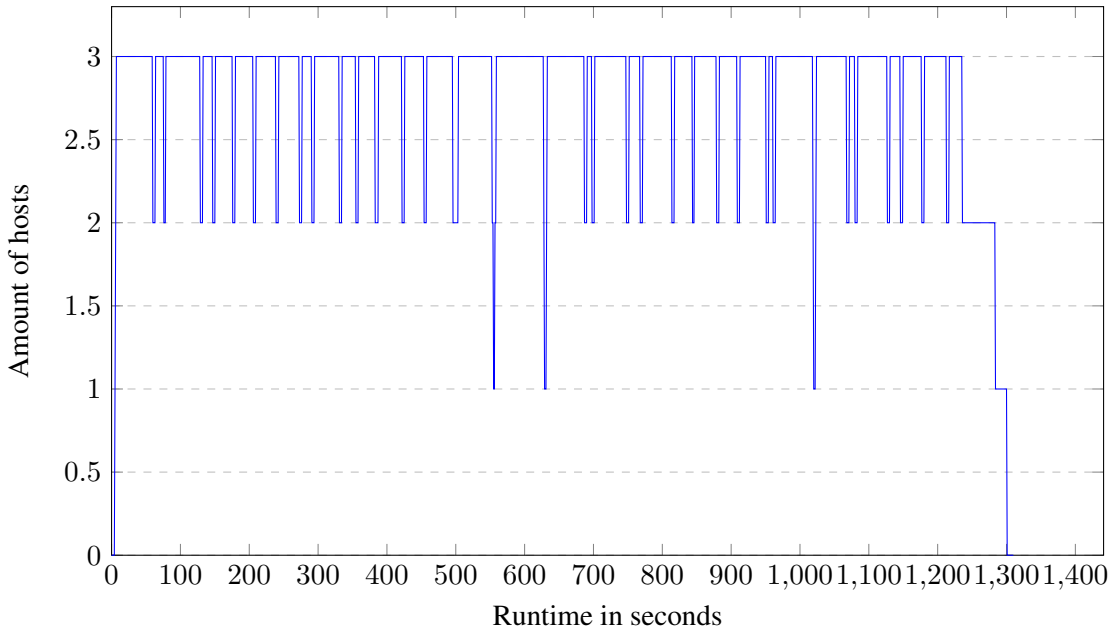
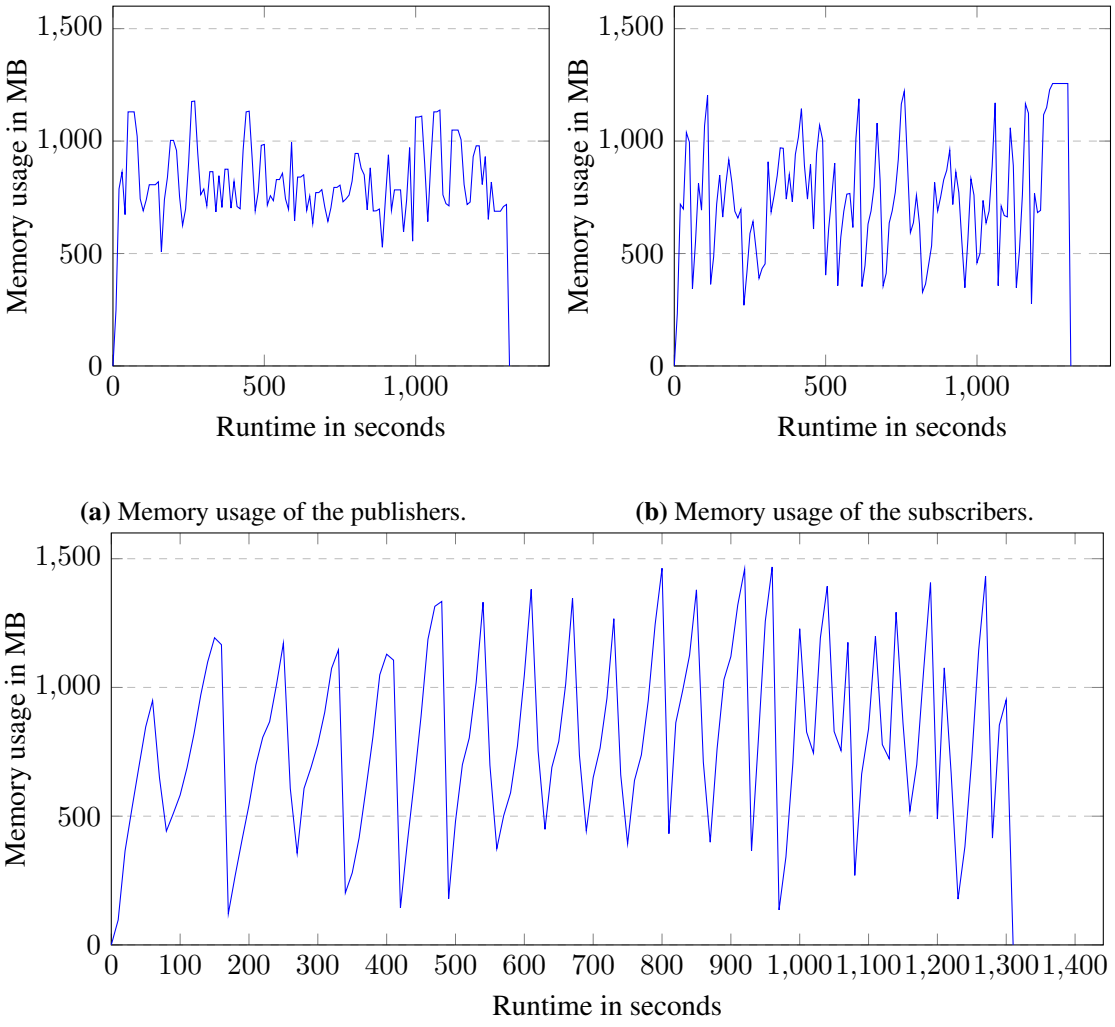


Figure 6.35: The amount of hosts that run parts of the application over it's runtime.

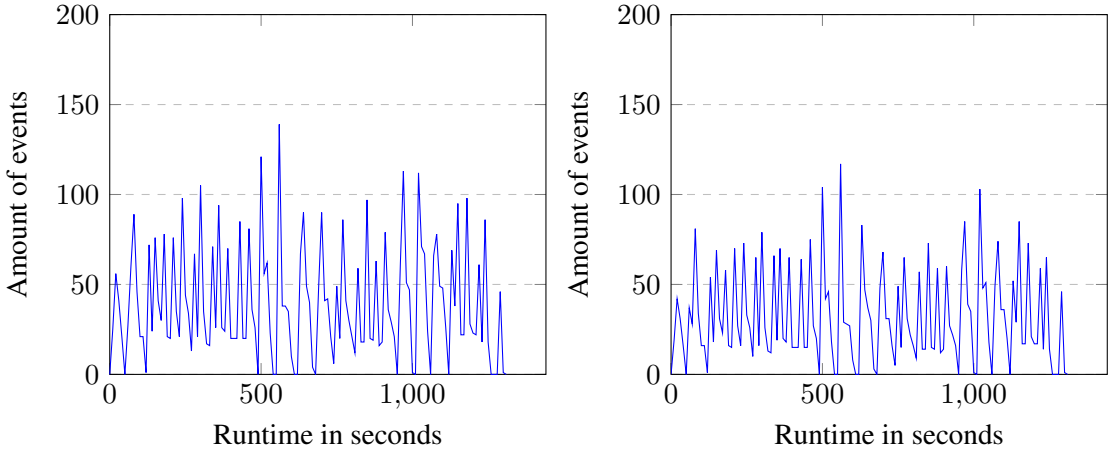


(a) Memory usage of the publishers.

(b) Memory usage of the subscribers.

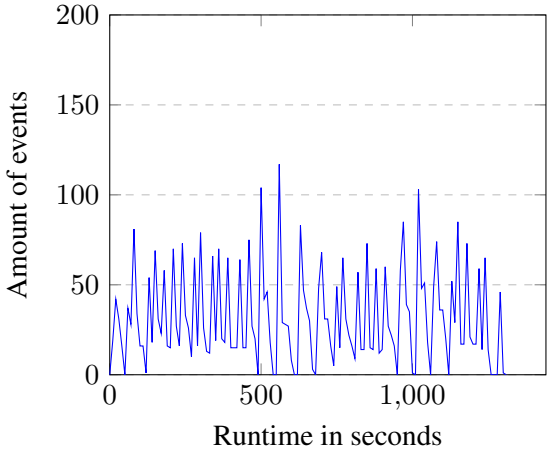
(c) Memory usage of the message broker.

Figure 6.36: The memory usage for each of the uses hosts in our evaluation application over time.

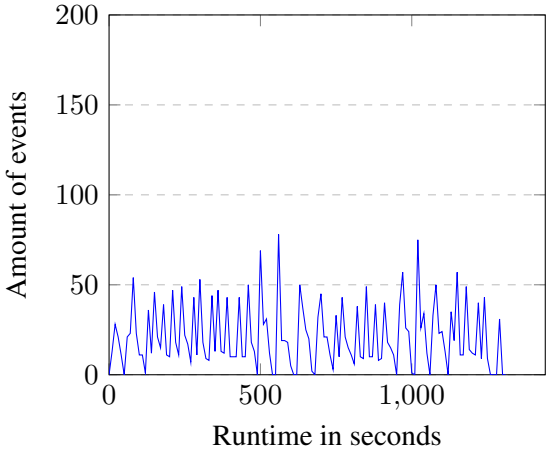


(a) Received *XADLEvent*s.

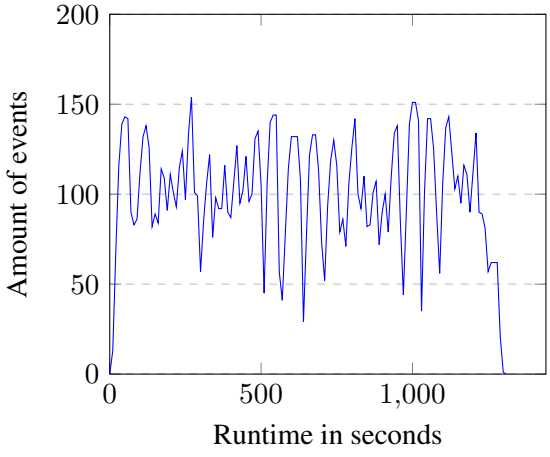
(b) Received *XADLRuntimeEvent*s.



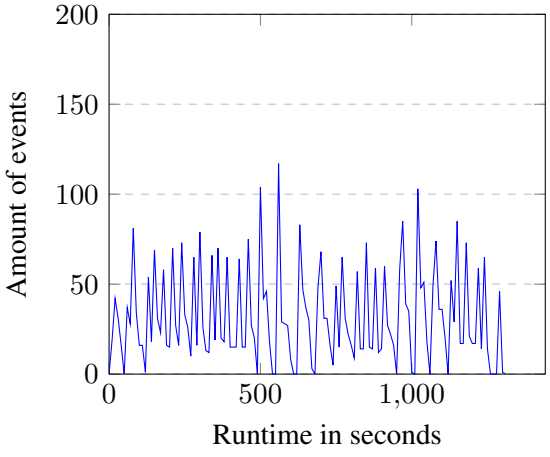
(c) Received *XADLLinkEvent*s.



(d) Received *XADLExternalLinkEvent*s.



(e) Received *XADLHostPropertyEvent*s.



(f) Received *XADLHostingEvent*s.

Figure 6.37: The different events received by the *Aggregator* over time.

6.6 Best Practices & Framework Limitations

We have seen that our approach is working as expected and can be used to monitor large distributed applications. Our evaluation scenarios provided us with this kind of information but they also revealed some limitations of our approach. Some of the limitation described above will be tackled in Chapter 7 by providing ways to overcome these limitations in future releases.

For our dynamic creation of bricks to work the developer needs to store all the information about the bricks to be created in the application itself. We need to know the blueprint identifier and all the types and names of interfaces the brick would have. The same applies to the linking of external interfaces. Here the developer needs to store the name and type of the interface and use these properties to dispatch the correct *XADLExternalLinkEvent*.

The best way to solve this would be to integrate the dynamically created bricks into the design-time architecture so our approach would be able to directly use this information.

The current implementation of the publish-subscribe pattern has some limitations due to it's current architecture. By handling each publisher in it's own thread and directly receiving and forwarding messages to the *MessageDistributor* without delay we loose the ability to reduce the bandwidth of the publisher. Once the transmission to the subscribers takes a little longer it may happen that many messages are held by the *MessageDistributor* which increases the memory usage of the application. A solution to this limitation would be the usage of non-blocking IO.

The current message transport implementation suffers from the size of the transported architectural events. Our evaluations have shown that these messages are around 1 Kilobytes (KB) each which leads to quite some network traffic while monitoring a distributed application. In case of our evaluation scenario we transferred a maximum of around 400 MB on event data. A way to overcome this would be to compress each message before it is transmitted.

Our evaluation results show that events of type *XADLEvent* and *XADLRuntimeEvent* are tightly coupled. The instantiation algorithm described in Section 5.2.1 shows that once all bricks are created their *begin* life-cycle method is called and thus for each brick both events are emitted. As our approach currently transmits both events, a possible solution for a subscriber to reduce the number of transfered events would be to subscribe only to events of type *XADLEvent*.

CHAPTER 7

Conclusion and Future Work

This final chapter concludes this thesis with a brief summary of the methods and results that have been presented and discussed. We will also recapitulate the results of this thesis that have been introduced in Section 1.2. Finally we will address possible future work that we were not able to cover.

7.1 Summary

The thesis starts with providing motivation for the creation of a decentralized architectural tracking framework including a motivated scenario. It follows by summarizing background information about the technologies that were used for the framework. Furthermore we presented research work related to the topics of this thesis.

We presented our approach to runtime architecture tracking by introducing our framework. At first the theoretical background of our approach was presented which completely describes how our framework works. We then presented our proof of concept implementation which addresses all the described theoretical topics. We have also shown how it is possible to integrate our framework into existing applications.

To verify that our approach works as expected we have presented a custom evaluation scenario, which was based on our motivated scenario. We have outlined the different problem instances that we have evaluated, presented and discussed the results for the feasibility and performance of our approach. Ultimately we provided best practices and the limitations of our approach.

We have published our work, that is the xADL extensions and the runtime architecture monitoring framework, at the following locations:

- `https://github.com/sideshowcecil/myx-monitor`
- `https://bitbucket.org/sideshow_bob/myx-monitor`

7.2 Results of the Master's Theses Revisited

We recapitulate the results of the thesis by returning to the requirements introduced in Section 1.2 and summarize the work that has been accomplished to address them.

As Chapter 4, 5 and 6 show we have created a distributed architecture tracking framework which satisfies the following requirements:

- **Local and decentralized (sub-)architecture management**
  Using the two applications of our approach, that is *Monitor* and *Aggregator*, we are able to extract architectural properties and aggregate them. The *Aggregator* may be deployed on the same machine as the monitored application for local architecture aggregation. By using a central *Aggregator* instance we are able to aggregate the complete architecture of a distributed application. The decentralized setting of the runtime architecture is defined by the *xArch HostProperty* xADL extension where we are able to assign a host to each component or connector and thus directly reflecting the decentralized setting in the architecture itself.

- **Architecture probes and sensors for cloud application, based on our evaluation scenario**
  Our approach introduces an event dispatcher which allows applications to install independent monitoring capabilities. By combining it with the *xArch HostProperty* xADL extensions gives us the ability to assign any kind of properties with a host, e.g. information about the host itself or statistical data. Our proof of concept implementation currently comes with two such dispatchers which can be used to monitor the load of a host.

- **A distribution mechanism for aggregating the overall architecture on specific granularity levels**
  By using a publish-subscribe system as a base for our *Aggregator* we are able to use the topic-based filtering to subscribe only to events about specific architectural properties. Thus we are able to create a hierarchy of *Aggregator*s with each operating on a different granularity level.

7.3 Future Work

Following the work presented in this theses, there are some improvements and future work we were not able to address sufficiently:

- **Dynamic creation of bricks** — Our current approach requires the manual specification of blueprint identifiers int the source code. An extended version of ArchStudio may automatically injecting them into the architectural description and our approach uses this information directly.

- **Hosts** — Information about the hosts are dynamically extracted by our approach. In a future version it might be possible to specify the host itself in the design-time architecture so the application may get launched on a specific host automatically.

- **Externalized links** — These kind of connections have to be monitored and propagated accordingly by the developer. By specifying information about the hosts in the design-time architecture we would able to distinguish between local- and external links thus it may be possible to directly specify them in the design-time architecture.
- **Decentralized consensus** — The current implementation of the aggregation of distributed architectures currently yields a single hierarchal instance that has overview of the complete system architecture. Whilst this approach can easily be extended to have multiple of such instances, the problem remains that those instances have no knowledge of each other. Future work may tackle to introduce a decentralized consensus between multiple aggregation components as described in [56] to overcome this problem.
- **Performance improvements** — There are different kinds of performance improvements that may be added to our framework:
  - *Using non-blocking IO* — Our message broker is currently excessively using threads to handle the different endpoints (publishers and subscribers). If many endpoints are connected to the message broker it makes sense to use Java NIO [34] to handle all connections.
  - *Batch event transfer* — The current event manager transfers each message directly to the configured aggregation component. To reduce the amount of transferred messages it would be applicable to transfer them in batches.
  - *Compression* — The current implementation of the event propagation simply serializes the events and sends them over the network. The reduce the amount of transferred data a simple compression algorithm may be used.
- **Supporting other technologies** — Our framework currently supports only a proprietary protocol for the propagation of architectural properties. Other technologies may be integrated:
  - *JMS* — It would be possible to include support for the JMS protocol for the transfer of messages. The only requirement would be a connector which allows to send and receive messages via JMS.
  - *RxJava* — By integrating the *Reactive Extensions for the JVM* into our message broker we might be able to increase it's overall performance.

APPENDIX A

Acronyms

ADL Architecture Description Language

ADT Abstract Data Type

AOP Aspect Oriented Programming

API Application Programming Interface

ASCII American Standard Code for Information Interchange

CDI Context and Dependency Injection

CPU Computational Processing Unit

CSV Comma Separated Values

DBL Data Binding Library

EJB Enterprise Java Beans

GRAF Graph-Based Runtime Adaptation Framework

IO Input/Output

IOC Inversion of Control

IP Internet Protocol

KB Kilobytes

JMS Java Message Service

JNDI Java Naming and Directory Interface

MADAM Mobility- and Adaptation-Enabling Middleware

MB Mega Bytes

OS Operating System

PC Personal Computer

PIM Platform Independent Model

POJO Plain Old Java Object

RMM Requirements Monitoring Model

PSM Platform Specific Model

SoS Systems of systems

SysML Systems Modeling Language

UML Unified Modeling Language

UUID Universal Unique Identifier

WAVE Waveform Audio File Format

XML eXtensible Markup Language

XSD XML Schema Definition

APPENDIX B

XADL Extensions

B.1 xArch Instance Mapping

```
<xsd:schema xmlns="http://www.ics.uci.edu/pub/arch/xArch/instancemapping.xsd"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:archinst="http://www.ics.uci.edu
        /pub/arch/xArch/instance.xsd"
    targetNamespace="http://www.ics.uci.edu/pub/arch/xArch/instancemapping.xsd"
    elementFormDefault="qualified" attributeFormDefault="qualified">

    <!-- Import namespaces used -->
    <xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/instance.xsd"
        schemaLocation="http://www.isr.uci.edu/projects/xarchuci/core/instance.xsd" />

    <xsd:annotation>
        <xsd:documentation>
            xArch IInstance Mapping XML Schema 1.0

            Change Log:
                2014-05-01: Bernd Rathmanner [bernd.rathmanner@student.tuwien.ac.at]:
                    Initial Development

        </xsd:documentation>
    </xsd:annotation>

    <!-- TYPE: MappedComponentInstance -->
    <xsd:complexType name="MappedComponentInstance">
        <xsd:complexContent>
            <xsd:extension base="archinst:ComponentInstance">
                <xsd:sequence>
                    <xsd:element name="blueprint" type="archinst:XMLLink" />
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <!-- TYPE: MappedConnectorInstance -->
    <xsd:complexType name="MappedConnectorInstance">
        <xsd:complexContent>
            <xsd:extension base="archinst:ConnectorInstance">
```

```
                <xsd:sequence>
                    <xsd:element name="blueprint" type="archinst:XMLLink" />
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <!-- TYPE: MappedInterfaceInstance -->
    <xsd:complexType name="MappedInterfaceInstance">
        <xsd:complexContent>
            <xsd:extension base="archinst:InterfaceInstance">
                <xsd:sequence>
                    <xsd:element name="type" type="archinst:XMLLink"
                        minOccurs="0" maxOccurs="1" />
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

</xsd:schema>
```

B.2 xArch External Identified Links

```
<xsd:schema xmlns="http://www.ics.uci.edu/pub/arch/xArch/extcon.xsd"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:archinst="http://www.ics.uci.edu
        /pub/arch/xArch/instance.xsd"
    targetNamespace="http://www.ics.uci.edu/pub/arch/xArch/extcon.xsd"
    elementFormDefault="qualified" attributeFormDefault="qualified">

    <!-- Import namespaces used -->
    <xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/instance.xsd"
        schemaLocation="http://www.isr.uci.edu/projects/xarchuci/core/instance.xsd" />

    <xsd:annotation>
        <xsd:documentation>
            xArch External Identified Links XML Schema1.0

            Change Log:
                2014-05-03: Bernd Rathmanner [bernd.rathmanner@student.tuwien.ac.at]:
                Initial Development

        </xsd:documentation>
    </xsd:annotation>

    <!-- TYPE: ExternalIdentifiedLinkInstance -->
    <xsd:complexType name="ExternalIdentifiedLinkInstance">
        <xsd:complexContent>
            <xsd:extension base="archinst:LinkInstance">
                <xsd:attribute name="extId" type="archinst:Identifier" />
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

</xsd:schema>
```

B.3 xArch HostProperty

```
<xsd:schema xmlns="http://www.ics.uci.edu/pub/arch/xArch/hostproperty.xsd"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:archinst="http://www.ics.uci.edu
        /pub/arch/xArch/instance.xsd"
```

```
xmlns:archtypes="http://www.ics.uci.edu/pub/arch/xArch/types.xsd"
targetNamespace="http://www.ics.uci.edu/pub/arch/xArch/hostproperty.xsd"
elementFormDefault="qualified" attributeFormDefault="qualified">

<!-- Import namespaces used -->
<xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/instance.xsd"
    schemaLocation="http://www.isr.uci.edu/projects/xarchuci/core/instance.xsd" />
<xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/types.xsd"
    schemaLocation="http://www.isr.uci.edu/projects/xarchuci/ext/types.xsd" />

<xsd:annotation>
    <xsd:documentation>
        xArch HostProperty XML Schema 1.0

        Change Log:
            2014-03-29: Bernd Rathmanner [bernd.rathmanner@student.tuwien.ac.at]:
                Extending for instance elmenents
            2012-07-24: Christoph Dorn [dorn@uci.edu]:
                Initial Development

    </xsd:documentation>
</xsd:annotation>

<!-- TYPE: HostedArchStructure -->
<xsd:complexType name="HostedArchStructure">
    <xsd:complexContent>
        <xsd:extension base="archtypes:ArchStructure">
            <xsd:sequence>
                <xsd:element name="host" type="Host"
                    minOccurs="0" maxOccurs="unbounded" />
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<!-- TYPE: HostedArchInstance -->
<xsd:complexType name="HostedArchInstance">
    <xsd:complexContent>
        <xsd:extension base="archinst:ArchInstance">
            <xsd:sequence>
                <xsd:element name="host" type="Host"
                    minOccurs="0" maxOccurs="unbounded" />
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<!-- TYPE: Host -->
<xsd:complexType name="Host">
    <xsd:sequence>
        <xsd:element name="description" type="archinst:Description" />
        <xsd:element name="hostProperty" type="Property"
            minOccurs="0" maxOccurs="unbounded" />
        <xsd:element name="subhost" type="Host"
            minOccurs="0" maxOccurs="unbounded" />
        <xsd:element name="hostsComponent" type="ElementRef"
            minOccurs="0" maxOccurs="unbounded" />
        <xsd:element name="hostsConnector" type="ElementRef"
            minOccurs="0" maxOccurs="unbounded" />
        <xsd:element name="hostsGroup" type="ElementRef"
            minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
```

```
        <xsd:attribute name="id" type="archinst:Identifier" />
    </xsd:complexType>

    <!-- TYPE: ElementRef -->
    <xsd:complexType name="ElementRef">
        <xsd:sequence>
            <xsd:element name="ref" type="archinst:XMLLink"
                minOccurs="1" maxOccurs="1" />
        </xsd:sequence>
    </xsd:complexType>

    <!-- TYPE: Property -->
    <xsd:complexType name="Property">
        <xsd:sequence>
            <xsd:element name="name" type="archinst:Description" />
            <xsd:element name="value" type="archinst:Description"
                minOccurs="1" maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>

</xsd:schema>
```

APPENDIX C

Event System

C.1 Base Event Class

```
package at.ac.tuwien.dsg.myx.monitor.em.events;

import java.io.Serializable;
import java.util.UUID;

public abstract class Event implements Serializable {

    private static final long serialVersionUID = -7750911233567472330L;

    private final String id;
    private final long timestamp;

    private String architectureRuntimeId;
    private String eventSourceId;

    public Event() {
        id = UUID.randomUUID().toString();
        timestamp = System.currentTimeMillis();
    }

    public Event(Event copyFrom) {
        id = UUID.randomUUID().toString();
        architectureRuntimeId = copyFrom.getArchitectureRuntimeId();
        timestamp = System.currentTimeMillis();
        eventSourceId = copyFrom.getEventSourceId();
    }

    public String getId() {
        return id;
    }

    public String getArchitectureRuntimeId() {
        return architectureRuntimeId;
    }

    public void setArchitectureRuntimeId(String architectureRuntimeId) {
```

```
        this.architectureRuntimeId = architectureRuntimeId;
    }

    public long getTimestamp() {
        return timestamp;
    }

    public String getEventSourceId() {
        return eventSourceId;
    }

    public void setEventSourceId(String eventSourceId) {
        this.eventSourceId = eventSourceId;
    }

    @Override
    public String toString() {
        return "Event [id=" + getId() + ", architectureRuntimeId=" +
            getArchitectureRuntimeId() + ", timestamp="
                + getTimestamp() + ", eventSourceId=" + getEventSourceId() + "]";
    }
}
```

APPENDIX D

Publish-Subscribe

D.1 Base Message Class

```
package at.ac.tuwien.dsg.pubsub.message;

import java.io.Serializable;

/**
 * This class contains the data for messages sent over the PubSubMiddleware.
 *
 * @author bernd.rathmanner
 *
 * @param <E>
 *            resembles the message data.
 */
public class Message<E> implements Serializable {

    private static final long serialVersionUID = 1L;

    private final Type type;
    private final String topic;
    private final E data;

    /**
     * Basic constructor with predefined message type <code>DATA</code>.
     *
     * @param data
     */
    public Message(String topic, E data) {
        this(Type.DATA, topic, data);
    }

    /**
     * Constructor for both message data and message type.
     *
     * @param data
     * @param type
     */
    public Message(Type type, String topic, E data) {
```

```
        this.topic = topic;
        this.type = type;
        this.data = data;
    }

    /**
     * Get the message type.
     *
     * @return
     */
    public Type getType() {
        return type;
    }

    /**
     * Get the message topic.
     *
     * @return
     */
    public String getTopic() {
        return topic;
    }

    /**
     * Get the message data.
     *
     * @return
     */
    public E getData() {
        return data;
    }

    @Override
    public String toString() {
        return "[" + getType() + "] [" + getTopic() + "] " + getData();
    }

    /**
     * All message types available.
     *
     * @author bernd.rathmanner
     *
     */
    public enum Type {
        TOPIC, INIT, DATA, CLOSE, ERROR
    }
}
```

D.2 Topic Implementations

D.2.1 String Topic

```
package at.ac.tuwien.dsg.pubsub.message.topic;

public class StringTopic implements Topic {

    protected final String topic;

    /**
     * Constructor.
```

```
     *
     * @param pattern
     */
    public StringTopic(String topic) {
        this.topic = topic;
    }

    @Override
    public boolean matches(String topic) {
        return this.topic.equals(topic);
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null && !(obj instanceof RegexTopic)) {
            return false;
        }
        return topic.equals(obj);
    }

    @Override
    public int hashCode() {
        return getClass().hashCode() + topic.hashCode();
    }

    @Override
    public String toString() {
        return "[" + getClass().getName() + "] " + topic;
    }
}
```

D.2.2 Regular Expression Topic

```
package at.ac.tuwien.dsg.pubsub.message.topic;

import java.util.regex.Pattern;

/**
 * Implements {@link Topic} based on regular expressions.
 *
 * @author bernd.rathmanner
 *
 */
public class RegexTopic implements Topic {

    protected Pattern pattern;

    /**
     * Constructor.
     *
     * @param pattern
     */
    public RegexTopic(String pattern) {
        this.pattern = Pattern.compile(pattern);
    }

    @Override
    public boolean matches(String topic) {
        return pattern.matcher(topic).matches();
    }
```

```
    @Override
    public boolean equals(Object obj) {
        if (obj == null && !(obj instanceof RegexTopic)) {
            return false;
        }
        return pattern.toString().equals(((RegexTopic) obj).pattern.toString());
    }

    @Override
    public int hashCode() {
        return getClass().hashCode() + pattern.hashCode();
    }

    @Override
    public String toString() {
        return "[" + getClass().getName() + "] " + pattern;
    }

}
```

D.2.3 Glob Pattern Topic

```
package at.ac.tuwien.dsg.pubsub.message.topic;

import java.util.ArrayList;
import java.util.List;

/**
 * Implements {@link Topic} based on the glob pattern.
 *
 * @author bernd.rathmanner
 *
 */
public final class GlobTopic extends RegexTopic {

    /**
     * Constructor.
     *
     * @param pattern
     */
    public GlobTopic(String pattern) {
        super(globToRegexp(pattern));
    }

    /**
     * Convert a glob pattern into a regular expression.
     *
     * @param globPattern
     * @return
     */
    private static final String globToRegexp(String globPattern) {
        StringBuilder buffer = new StringBuilder();

        int length = globPattern.length();
        for (int i = 0; i < length; ++i) {
            char c = globPattern.charAt(i);

            switch (c) {
            case '*':
                buffer.append(".*");
                break;
```

```
case '?':
    buffer.append('.');
    break;

case '[': {
    int j = findClosingSquareBracket(globPattern, i);
    if (j <= i + 1) {
        // Something like "[" or "[]" has no special meaning.
        buffer.append("\\[");
    } else {
        buffer.append(globPattern.substring(i, j + 1));
        i = j;
    }
}
    break;

case '{': {
    List<String> parts = new ArrayList<String>();
    int j = splitCurlyBraceGroup(globPattern, i, parts);
    int partCount = parts.size();
    if (j <= i + 1 || partCount == 0) {
        // Something like "{", "{}", "{,}", or "{,,}" has no
        // special meaning.
        buffer.append("\\{");
    } else {
        if (partCount == 1) {
            // Not very useful but why not?
            buffer.append('(');
            buffer.append(globToRegexp(parts.get(0)));
            buffer.append(')');
        } else {
            buffer.append('(');
            for (int k = 0; k < partCount; ++k) {
                if (k > 0) {
                    buffer.append('|');
                }
                buffer.append('(');
                buffer.append(globToRegexp(parts.get(k)));
                buffer.append(')');
            }
            buffer.append(')');
        }
        i = j;
    }
}
    break;

case '\\':
    // Escaped char: add as is (that is, escaped).
    buffer.append(c);
    if (i + 1 < length) {
        buffer.append(globPattern.charAt(++i));
    }
    break;

default:
    if (!Character.isLetterOrDigit(c)) {
        // Escape special chars such as '(' or '|'.
        buffer.append('\\');
    }
    buffer.append(c);
```

```
            }
        }

        return buffer.toString();
    }

    /**
     * Find the closing bracket in a the given string.
     *
     * @param s
     * @param offset
     * @return
     */
    private static int findClosingSquareBracket(String s, int offset) {
        int nesting = 0;
        char prevC = '\0';
        int length = s.length();

        for (int i = offset; i < length; ++i) {
            char c = s.charAt(i);

            switch (c) {
            case '[':
                if (prevC != '\\') {
                    ++nesting;
                }
                break;
            case ']':
                if (prevC != '\\' &&
                // Something like "[]a-b]" is equivalent to "[\]a-b]".
                        i != offset + 1) {
                    --nesting;
                }

                if (nesting == 0) {
                    return i;
                }
                break;
            }

            prevC = c;
        }

        return -1;
    }

    /**
     * Split a curly braced group.
     *
     * @param s
     * @param offset
     * @param parts
     * @return
     */
    private static int splitCurlyBraceGroup(String s, int offset, List<String> parts) {
        int groupOffset = offset;
        int nesting = 0;
        char prevC = '\0';
        int length = s.length();

        for (int i = offset; i < length; ++i) {
            char c = s.charAt(i);
```

```
            switch (c) {
            case '{':
                if (prevC != '\\') {
                    ++nesting;
                }
                break;
            case '}':
                if (prevC != '\\') {
                    --nesting;
                }

                if (nesting == 0) {
                    String part = s.substring(groupOffset + 1, i);
                    if (part.length() > 0) {
                        parts.add(part);
                    }

                    return i;
                }
                break;
            case ',':
                if (nesting == 1) {
                    String part = s.substring(groupOffset + 1, i);
                    if (part.length() > 0) {
                        parts.add(part);
                    }

                    groupOffset = i;
                }
                break;
            }

            prevC = c;
        }

        return -1;
    }

}
```

D.3 ExecutorService Extension

```
package at.ac.tuwien.dsg.concurrent;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.RejectedExecutionException;

/**
 * An extension to the {@link ExecutorService} interface that allow to
 * execute tasks on specific {@link Thread}s using identifiers.
 */
public interface IdentifiableExecutorService extends ExecutorService {
    /**
     * Executes the given command at some time in the future. The command may
     * execute in a new thread, in a pooled thread, or in the calling thread,
     * at the discretion of the {@code Executor} implementation. This method
     * guarantees that a {@link Runnable} with the same identifier is always
     * executed by the same {@link Thread}.
     *
     * @param command
```

```
     *            the runnable task
     * @param identifier
     *            the identifier
     * @throws RejectedExecutionException
     *             if this task cannot be accepted for execution
     * @throws NullPointerException
     *             if command is null
     */
    void execute(Runnable command, int identifier);
}
```

Bibliography

[1] Mehdi Amoui, Mahdi Derakhshanmanesh, JüRgen Ebert, and Ladan Tahvildari. Achieving dynamic adaptation via management and interpretation of runtime models. *J. Syst. Softw.*, 85(12):2720–2737, December 2012.

[2] Hazel Asuncion. Myx tutorial. `http://www.ics.uci.edu/~taylor/classes/221/MyxTutorial.pdf`, 2009.

[3] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, November 1987.

[4] Shang-Wen Cheng. *Rainbow: cost-effective software architecture-based self-adaptation*. ProQuest, 2008.

[5] Shang-Wen Cheng and David Garlan. Handling uncertainty in autonomic systems. In *Proceedings of the International Workshop on Living with Uncertainties (IWLU'07), co-located with the 22nd International Conference on Automated Software Engineering (ASE'07)*, Atlanta, GA, USA, 5 November 2007. http://godzilla.cs.toronto.edu/IWLU/program.html.

[6] Shang-Wen Cheng and David Garlan. Stitch: A language for architecture-based self-adaptation. *J. Syst. Softw.*, 85(12):2860–2875, December 2012.

[7] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Making self-adaptation an engineering reality. In Ozalp Babaoglu, Mark Jelasity, Alberto Montresor, Christof Fetzer, Stefano Leonardi, Aad Moorsel, and Maarten Steen, editors, *Self-star Properties in Complex Information Systems*, volume 3460 of *Lecture Notes in Computer Science*, pages 158–173. Springer Berlin Heidelberg, 2005.

[8] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.

[9] Paul Clements, David Garlan, Reed Little, Robert Nord, and Judith Stafford. Documenting software architectures: Views and beyond. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 740–741, Washington, DC, USA, 2003. IEEE Computer Society.

[10] E Dashofy. The myx architectural style. http://isr.uci.edu/projects/archstudio/resources/myx-whitepaper.pdf, 2006.

[11] E Dashofy, D Garlan, A Koek, and B Schmerl. xarch: an xml standard for representing software architectures. http://isr.uci.edu/projects/xarch/. Accessed: 2014-05-27.

[12] Eric Dashofy, Hazel Asuncion, Scott Hendrickson, Girish Suryanarayana, John Georgas, and Richard Taylor. Archstudio 4: An architecture-based meta-modeling environment. In *Companion to the Proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION '07, pages 67–68, Washington, DC, USA, 2007. IEEE Computer Society.

[13] Eric M Dashofy. Issues in generating data bindings for an xml schema-based language. In *Proceedings of the Workshop on XML Technologies and Software Engineering (XSE2001), Toronto, ONT, Canada*, 2001.

[14] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Trans. Softw. Eng. Methodol.*, 14(2):199–245, April 2005.

[15] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A highly-extensible, xml-based architecture description language. In *WICSA*, pages 103–112, 2001.

[16] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 21–26, New York, NY, USA, 2002. ACM.

[17] Giovanna Di Marzo Serugendo, John Fitzgerald, Alexander Romanovsky, and Nicolas Guelfi. A metadata-based architectural model for dynamically resilient systems. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, pages 566–572, New York, NY, USA, 2007. ACM.

[18] Christoph Dorn and Richard N. Taylor. Coupling software architecture and human architecture for collaboration-aware system adaptation. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 53–62, Piscataway, NJ, USA, 2013. IEEE Press.

[19] Jürgen Ebert, Volker Riediger, and Andreas Winter. Graph technology in reverse engineering – the tgraph approach. In *PROC. 10TH WORKSHOP SOFTWARE REENGINEERING. GI LECTURE NOTES IN INFORMATICS*, pages 67–81, 2008.

[20] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[21] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *IEEE Softw.*, 23(2):62–70, March 2006.

[22] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, October 2004.

[23] David Garlan, Robert T Monroe, and David Wile. Acme: Architectural description of component-based systems. *Foundations of component-based systems*, 68:47–68, 2000.

[24] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *Proceedings of the First Workshop on Self-healing Systems*, WOSS '02, pages 27–32, New York, NY, USA, 2002. ACM.

[25] David Garlan, Bradley Schmerl, and Shang-Wen Cheng. Software architecture-based self-adaptation. In Yan Zhang, Laurence Tianruo Yang, and Mieso K. Denko, editors, *Autonomic Computing and Networking*, pages 31–55. Springer US, 2009.

[26] Holger Giese and Stephan Hildebrandt. Incremental model synchronization for multiple updates. In *Proceedings of the Third International Workshop on Graph and Model Transformations*, GRaMoT '08, pages 1–8, New York, NY, USA, 2008. ACM.

[27] Holger Giese and Robert Wagner. From model transformation to incremental bidirectional model synchronization. *Software & Systems Modeling*, 8(1):21–43, 2009.

[28] Google. Guice - bindings. `https://github.com/google/guice/wiki/Bindings`. Accessed: 2015-07-30.

[29] Google. Guice - getting started. `https://github.com/google/guice/wiki/GettingStarted`. Accessed: 2015-07-30.

[30] Google. Guice - jsr330 integration. `https://github.com/google/guice/wiki/JSR330`. Accessed: 2015-07-30.

[31] Google. Guice - wiki. `https://github.com/google/guice/wiki`. Accessed: 2015-07-30.

[32] M.M. Gorlick and R.R. Razouk. Using weaves for software construction and analysis. In *Software Engineering, 1991. Proceedings., 13th International Conference on*, pages 23–34, May 1991.

[33] Svein Hallsteinsen, Jacqueline Floch, and Erlend Stav. A middleware centric approach to building self-adapting systems. In Thomas Gschwind and Cecilia Mascolo, editors, *Software Engineering and Middleware*, volume 3437 of *Lecture Notes in Computer Science*, pages 107–122. Springer Berlin Heidelberg, 2005.

[34] Ron Hitchens. *Java Nio*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[35] Paul Horn. Autonomic computing: IBM's Perspective on the State of Information Technology, 2001.

[36] Irvine Institute for Software Research at University of California. Myx and myx.fw. http://isr.uci.edu/projects/archstudio-4/www/archstudio/myx.html. Accessed: 2014-05-18.

[37] Eric Jendrock, Ricardo Cervera-Navarro, Devika Gollapudi Ian Evans, Kim Haase, William Markito, and Chinmayee Srivathsa. The java ee 6 tutorial. https://docs.oracle.com/javaee/6/tutorial/doc/javaeetutorial6.pdf, 2013.

[38] Rod Johnson, Juergen Hoeller, Keith Donald, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Alef Arendsen, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet, Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, Mark Fisher, Sam Brannen, Ramnivas Laddad, Arjen Poutsma, Chris Beams, Tareq Abedrabbo, Andy Clement, Dave Syer, Oliver Gierke, Rossen Stoyanchev, Phillip Webb, Rob Winch, Brian Clozel, Stephane Nicoll, and Sebastien Deleuze. Spring framework reference documentation - 5. the ioc container. http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html. Accessed: 2015-07-30.

[39] Andrew Josey, DW Cragun, N Stoughton, M Brown, C Hughes, et al. The open group base specifications issue 6 ieee std 1003.1. *The IEEE and The Open Group*, 20(6), 2004.

[40] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[41] Paul J Leach, Michael Mealling, and Rich Salz. A universally unique identifier (uuid) urn namespace. RFC 4122, The Internet Engineering Task Force, July 2005.

[42] Markus Luckey, Benjamin Nagel, Christian Gerth, and Gregor Engels. Adapt cases: Extending use cases for adaptive systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 30–39, New York, NY, USA, 2011. ACM.

[43] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In Wilhelm Schäfer and Pere Botella, editors, *Software Engineering - ESEC '95*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153. Springer Berlin Heidelberg, 1995.

[44] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and their Applications, IEEE*, 14(3):54–62, 1999.

[45] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[46] Bradley Schmerl and David Garlan. Exploiting architectural design knowledge to support self-repairing systems. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, SEKE '02, pages 241–248, New York, NY, USA, 2002. ACM.

[47] B. Solomon, D. Ionescu, M. Litoiu, and M. Mihaescu. Model-driven engineering for autonomic provisioned systems. In *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International*, pages 1110–1115, July 2008.

[48] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[49] R.N. Taylor, N. Medvidovic, Kenneth M. Anderson, E.James Whitehead Jr., Jason E. Robbins, Kari A. Nies, P. Oreizy, and D.L. Dubrow. A component- and message-based architectural style for gui software. *Software Engineering, IEEE Transactions on*, 22(6):390–406, Jun 1996.

[50] M. Vierhauser, R. Rabiser, P. Grunbacher, C. Danner, S. Wallner, and H. Zeisel. A flexible framework for runtime monitoring of system-of-systems architectures. In *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, pages 57–66, April 2014.

[51] Michael Vierhauser, Rick Rabiser, Paul Grünbacher, Klaus Seyerlehner, Stefan Wallner, and Helmut Zeisel. Reminds : A flexible runtime monitoring framework for systems of systems. *Journal of Systems and Software*, pages –, 2015.

[52] Thomas Vogel and Holger Giese. Adaptation and abstract runtime models. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 39–48, New York, NY, USA, 2010. ACM.

[53] Thomas Vogel, Stefan Neumann, Stephan Hildebrandt, Holger Giese, and Basil Becker. Model-driven architectural monitoring and adaptation for autonomic systems. In *ICAC*, pages 67–68, 2009.

[54] Thomas Vogel, Stefan Neumann, Stephan Hildebrandt, Holger Giese, and Basil Becker. Incremental model synchronization for efficient run-time monitoring. In Sudipto Ghosh, editor, *Models in Software Engineering*, volume 6002 of *Lecture Notes in Computer Science*, pages 124–139. Springer Berlin Heidelberg, 2010.

[55] Thomas Vogel, Andreas Seibel, and Holger Giese. The role of models and megamodels at runtime. In *MoDELS Workshops*, pages 224–238, 2010.

[56] Danny Weyns, Sam Malek, and Jesper Andersson. On decentralized self-adaptation: Lessons from the trenches and challenges for the future. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 84–93, New York, NY, USA, 2010. ACM.

[57] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 371–380, New York, NY, USA, 2006. ACM.