The INAETICS architecture

Introducing INAETICS

Authors:

Hans Bossenbroek

René van Hees

inaetics.org

Table of Contents

1 The INAETICS project

The goal of the INAETICS project aims to benefit from the changed economic opportunities of open innovation approaches in the field of software systems that are defined by a set of high-availability requirements. Within INAETICS, these requirements are translated into an architecture for geographically dispersed systems with a time-critical (often real-time) aspect and dynamic security demands. Because of the extensive character of the architecture and the open innovation strategy, the success of the INAETICS project is not only defined by a fit-for-purpose architecture and implementation, but much more in the form of a pre-competitive community that is motivated by a common Open Innovation strategy.

The main purpose of any Open Innovation strategy is to share knowledge and experiences in a pre-competitive setup with the purpose to gain a strategic advantage in terms of agility and time to market. Any successful Open Innovation strategy starts with a shared vision and shared principles. For the INAETICS architecture, the vision is based on the principles that evolution can be designed into a system and that it is possible to drive the development of INAETICS-based systems using one overall architecture style.

This architecture style, which is based on a dynamic services architecture, requires both a modular development strategy as well as a rich and robust infrastructure to support a cost-effective development process. The INAETICS project not only aims to deliver a fit-for-purpose architecture, but also an initial implementation of the infrastructure and demonstrate the usability using a number of domain-related examples. The core of this infrastructure will be made available as Open Source under the Amdatu project[1].

1.1 About this document

This document is written as part of the INAETICS research project. As such, it is part of a larger collection of documents. Essentially, this document builds on the whitepaper “Software architecture in an open world”[2]. In combination with this document, which introduces the INAETICS architecture, these two documents offer a complete introduction into the goals and intended architecture of the INAETICS project. By itself, this document aims to describe the INAETICS architecture by high-lighting the core mechanisms and patterns; the actual, more detailed designs are described in specific documents that are either mechanism or subsystem specific.

[1] Amdatu is an Open Source project aimed at fostering Open Innovation initiatives. See also: http://www.amdatu.org/philosophy.html.
[2] The whitepaper is available for download at http://www.inaetics.org.

2 Dealing with change

One of the biggest challenges driving the INAETICS architecture, apart from the collaborative aspect that is dictated by the Open Innovation strategy, is how to design evolvability into complex, distributed systems. The INAETICS project intends to address this challenge by rigorously applying compositional techniques with a design that is ultimately controlled by a dynamic coordination strategy.
Within INAETICS, the compositional techniques are realized using a dynamic component-services architecture-style. This is elaborated further in paragraphs 3.1 and 3.2. The actual realization is directly driven by the high-availability, deterministic and geographically dispersed character of the domains that are targeted by the INAETICS project.

Conceptually, the INAETICS architecture acknowledges 2 levels of behavior:

- Behavior and demands of the problem space: i.e. In order to be able to coordinate intelligently how a solution reacts to changing constraints or requirements, it is imperative to specify the desired behavior in terms that are not contaminated by elements of the solution space. This approach is called 'intent-based[3] modeling', a strategy that finds its roots in the design of large, complex networks where it is impossible to oversee all hardware and functional invariants.
- Behavior and qualities of the solution space: i.e. given a set of designed and implemented software services, configurations and hardware platforms, a solution space has been defined that is limited by a number of constraints or qualities. In general, the dynamics of this solution space is rooted in the granularity and dynamic capabilities of the contained applications, modules and components and their configurations.

These 2 levels of behavior are addressed separately to achieve the desired level of controlled evolvability. In the problem space the INAETICS architecture uses a coordination-based approach that is described in paragraph 4.2.3 (The INAETICS coordination model) of the architecture description. The evolvability aspect of the solution space is an integral part of the software architecture, but is largely based on a mechanism where the dynamics of a solution are made explicit in terms of capabilities, requirements and context of individual parts of a system.

2.1 Changing security

The evolvability aspect of the INAETICS architecture is not only aimed handling changed functional requirements or resource constraints, but also the ability handle changed security parameters. Currently, the majority of IT systems feature a rigid and centralized approach to security. Their security design, if any, is based on the notion of a single and usually rigid 'security perimeter' that is guarded using security policies that are managed from a centralized location. This approach has 2 major drawbacks. In the first place, recent history has shown that the security that systems have to deal with are changing rapidly, both in frequency as well as intelligence and intent. Furthermore, with the

[3] A more in-depth description of Intent-based design can be found in Appendix B.

increased number of connected devices of IoT deployments[4] it is becoming obvious that centralized approaches to security-management have proven not to scale sufficiently.

For these reasons, the INAETICS architecture also includes the evolution of security models and related mechanisms. Using a combination of attribute-based encryption, short-lived trust relations and non-centralized security-management, it is possible to dynamically reestablish trust-relations and security perimeters with a minimum of human intervention.

[4] According to Gartner it will scale to 26 billion devices in 2020 (see http://www.gartner.com/newsroom/id/2636073).

3 Architectural Principles

This chapter aims to capture the design philosophy of the INAETICS Architecture in the form of a set of principles. The principles of the INAETICS architecture extend beyond the principles of a software architecture that tend to have a qualitative character. Examples of architecture principles with an accepted, yet qualitative character are:

- The system should be built to change instead of building to last.
- The architecture should be modeled in order to analyze and reduce risk.
- Models and their visualizations should be used as communication and collaboration tooling.
- The key engineering decisions should be identified and acted upon upfront.

For INAETICS, these core principles aim to define a specific design and implementation space. This space dictates that it is possible to base any solution in the INAETICS target domains on a collection of one or more components or systems that can only inter-operate through defined services. The principles required to define that space as well as the inherent dynamics are described in the next paragraphs.

3.1 Software Modularity

The most fundamental principle of the INAETICS architecture is to apply the software modularity paradigm on every level of abstraction or development phase. The concept of software modularity is described by Edsger Dijkstra in the early 70's[5] and is to date one of the few patterns to conceptualize complex systems. The software modularity paradigm dictates that every system can be seen as a cohesive set of software parts[6] during every lifecycle of the development. By identifying and relating these parts during every phase of the lifecycle of a system, it is possible to achieve new levels of evolvability.

Essential in designing and implementing software using a modular approach is to rigorously apply the principle of "Low Coupling and High Cohesion". High Cohesion means that parts that can be grouped together into modules whenever their functionality is strongly related, whereas "Low Coupling" means that these modules should be dependent on each other to the least extent practically possible. These two principles are best implemented using a dynamic component-services architecture style.

3.2 Dynamic Component-Services Architecture

An architectural style that inherently supports the modularity paradigm is a dynamic component-service architecture. According to this style, any system can be partitioned into components that publish their functionality through services. These components can only interact with other components using these services. And because collaborations of services are defined to be inherently dynamic, it is possible to facilitate run-time evolution.

[5] This was described in EWD300 (for a transcript in Dutch see http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD300.html).
[6] For the sake of clarity, we use the term 'software part' and not modules or components to refer to any well-defined part of a system. These terms are part of a defined nomenclature which is described in paragraph 4.2.1.

The INAETICS architecture provides systems with an infrastructure that provides a level of indirection whereby systems can control the visibility and accessibility of services and, if applicable, extend services with remote capabilities. As a result, INAETICS systems can be partitioned into a set of components and services in a controlled way, without cluttering the implementation space with all kinds of partitioning details. Beyond the boundaries of a single system, larger solutions can be based on so called systems of systems[7], which are largely based on the same partitioning and coordination mechanisms.

3.3 Dynamic application assembly and deployment

In order to be able to handle large sets of components and the dynamics of services the INAETICS architecture supports the capability to dynamically assemble and deploy systems. Assembly and deployment of systems is based on compositions. The structure and dynamics of these compositions can be expressed through abstract capabilities and requirements. This makes it possible to specify how systems should be deployed, allocated and wired optimally given the current state, capabilities and availability of resources.

For this purpose, the INAETICS Architecture features a Domain Specific Language (or DSL) that enables architects to express the level of dynamism in the service dependency resolution process. This service-dependency DSL is largely based on a Requirements-Capabilities approach that can express how services relate[8]. Essentially, it assumes any required service can be matched against a collection of providers with capabilities, which may in turn have transitive requirements, and defines a resolution mechanism.

3.4 Risk-adaptive security architecture

The systems that are developed using the INAETICS architecture are expected to use (parts of) the public Internet. The continued growth and ubiquitous character of the Internet have made people aware of the need for increased security. Protection of valuable information assets and digitally controlled physical assets is a critical issue for everyone. The job of securing services, platforms, and networks from malicious users and software is a difficult one. The security dimension of the INAETICS architecture manifests itself on various levels. In the first place, all software conforms to the 'secure by design' philosophy[9], meaning that security is taken into account at all times, based on the assumption that malicious attempts will take place. But above all in order to facilitate the dynamic runtime behavior, the underlying infrastructure must support facilities to establish trust between services and enforce various levels of encryption whilst maintaining an acceptable level of overhead and pervasiveness.

[7] See also https://en.wikipedia.org/wiki/System_of_systems.
[8] A comparable mechanism is the Declarative Services specification that was introduced in release 5 of the OSGi standard. See also http://wiki.osgi.org/wiki/Declarative_Services.
[9] Secure by design means that the software has been designed from the ground up to be secure. Malicious practices are taken for granted and care is taken to minimize impact when a security vulnerability is discovered. See also: http://en.wikipedia.org/wiki/Securebydesign

To this end, it is assumed that the dynamic aspect behavior of INAETICS systems also stretches across the boundary of security by enabling the use of variable security perimeters and threat-dependent security levels.

4 The INAETICS architecture

The INAETICS architecture is geared towards organizations that have a software-intensive strategy and have access to dedicated software R&D departments. For this reason, the INAETICS architecture is designed to strike a balance between team-productivity and versatility. This balance is implemented by offering a consistent design and implementation space in combination with a rich infrastructure that hides complexity and distracting technical details.

4.1 Overview

The design and implementation space of INAETICS systems is based on a dynamic component-services architecture (see also paragraph 3.2). This means that any INAETICS solution can be composed of a dynamic set of systems and applications (see also paragraph 4.2.1). Each of these applications:

- Can be modeled as a dynamic set of services.
- Which are implemented by a cohesive set of modules and components, where:
    - Any component can only be accessed through one of its defined services.
    - Each component is self-contained, and does not depend on the context or state of other services.
    - The assembly of the modules and components, the application, can vary at runtime.

Because of the challenges that are inherent to the target domains of the INAETICS project, it is complicated to oversee, implement or deploy this architecture without subdividing the architecture. Therefore, in order to retain a manageable architecture, the INAETICS architecture is split-up into a number of layers:

- **The Coordination Layer** - The top layer contains the services and mechanisms capable of controlling the dynamic capabilities of INAETICS solutions. Ultimately, one of the goals of the INAETICS architecture is to dynamically resolve changes in the context of a solution. This capability is based on a coordination strategy, which is contained in this layer.
- **The Container Layer** - The middle layer of the INAETICS architecture implements an environment that isolates applications from each other and the underlying infrastructure while providing an added layer of protection for the application itself. In services architectures, this environment is usually referred to as a container. Containers control the lifecycle, context and isolation of applications and its components and services.
- **The Fabric Layer** - The bottom layer contains the services and mechanisms that abstract from the underlying networking, storage and computing infrastructure. Effectively these services and mechanisms are based on the notion of fabric computing[10].

These logical layers are illustrated in the following diagram:

[10] See also: http://en.wikipedia.org/wiki/Fabriccomputing.

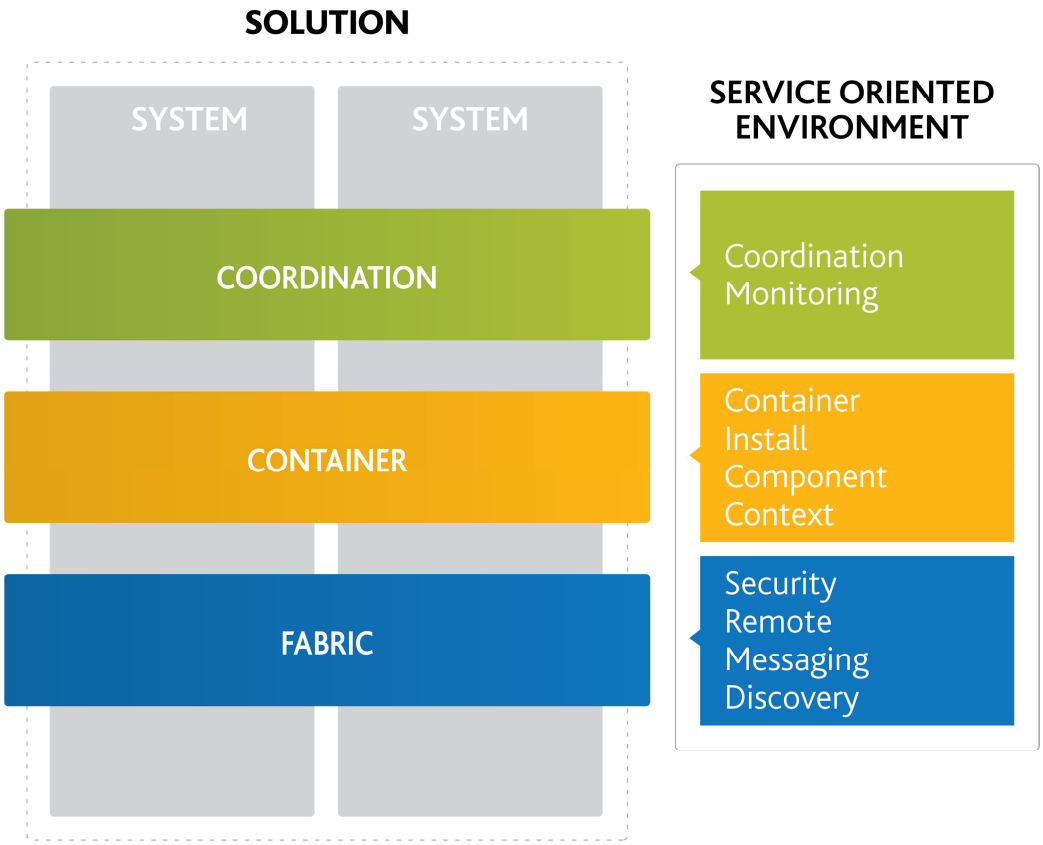

Figure 1: An overview of the architectural layers.

The figure above shows a graphical representation of the layers in the INAETICS architecture as well as a number of typical supporting features that can be found in a Service Oriented programming environment. A number of these typical features are described in Appendix A. Any INAETICS solution lives on top of computer resources which are represented using a secure fabric computing abstraction. On top of the Fabric layer, the application lifecycle is managed in the container layer. The container layer leverages the capabilities of the fabric layer and implements context services, install services and the underlying fabric layer services in order to support the lifecycle of applications and related components and services. The actual evolvability of a solution is controlled by services in the coordination layer. These services use the information that is exposed by the other layers to define a dynamic set of coordination algorithms to handle complex changes in the environment.
Using the three layers introduced above, the actual architecture will be described by highlighting the core mechanisms, followed by the architectural use cases. Each of the core mechanisms as well as a number of prototypical deployments of the INAETICS architecture are described in more detail in separate, yet related documents.

4.2 Core architectural mechanisms

The INAETICS architecture is based on a limited number of core mechanisms: an applied form of a dynamic component/services architecture, a dynamic security architecture and a specific approach to coordination. Each of these mechanisms will be described in the next paragraphs.

4.2.1 The INAETICS component – services model

INAETICS is defined as a architecture for software intensive solutions. As such, it defines any INAETICS solution as a collection of software parts with a specific nomenclature and ordering that is defined in the INAETICS component – services model. The software parts identified in this component – services model are designed to support solutions in the target domains using a compositional approach. Each of these types of software parts is illustrated in the next figure and will be described in the rest of this paragraph.

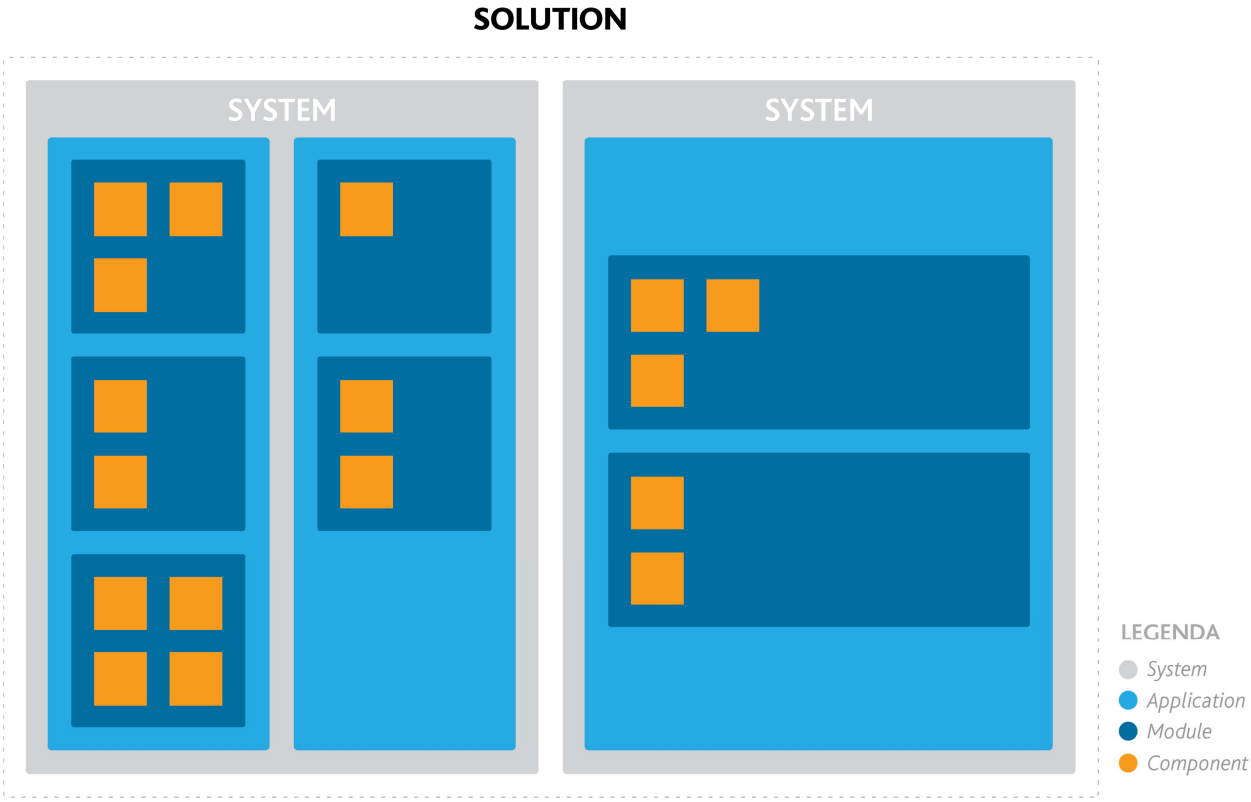

Figure 2: An overview of the INAETICS component nomenclature.

Conceptually, solutions Solutions, or solution definitions to be more specifically, are at the top of the component-services nomenclature of the INAETICS architecture. In the INAETICS architecture, solutions are defined using an intent-based model that is described using a INAETICS specific DSL[11]. This part of the INAETICS DSL contains a domain-specific aspect as well as behavior oriented aspect in the form of intent definitions.

Solutions in turn are composed out of one or more systems. Systems are defined using a constraint-based model that is described using another part of the INAETICS specific DSL. This part of the INAETICS DSL can express a variability or performance and other qualitative aspects to describe the boundaries of operation and an assembly aspect.

[11] DSL is an abbreviation for Domain Specific Language. (see also 3.3 Dynamic application assembly and deployment).

The third type of software part is the application. Applications are defined as one or more software programs that are related because of their identical scope or specifications. This scope is usually characterized by either a defined set of end-user roles or features that are related because of their strongly related place-time-feature aspect. Any INAETICS system can consist of one or more applications, where an application is based on a declarative[12] model that describes the logic and boundaries of an application, without describing the control flow over the applications. This declarative model is also part of the INAETICS specific DSL.

The declarative specification of an INAETICS application needs to be mapped on the underlying infrastructure using the models and capabilities of the Fabric layer. This is done, in decreasing order of size and complexity, using Modules and Components. Modules are software parts that are security and network aware and can communicate over zone-boundaries. As such they are based on a functional and logical viewpoint on security and units of networking offered by the services in the fabric layer. Components are the basic building blocks of the INAETICS architecture and contain all the functionality. As an overview and reference the following table, which lists the nomenclature of the software parts in the INAETICS architecture, is provided.

Software part	Coherence	Network	Dynamism
Solution	Intent definition	Infrastructure neutral	Intent-based coordination
System	Constraint/deployment definition	Cross infrastructure	Constraint-based coordination
Application	Declarative definition	Zone-remote	Runtime (QoS) based resolution
Module	Functional / Security	Node-Remote and zone-local	Runtime, semantic versioning
Component	Physical	Node-local	Code-time

Table 1: The nomenclature of the software parts of the INAETICS architecture.

4.2.2 The INAETICS security model

The security model of the INAETICS architecture is based on a number of core mechanisms. These mechanisms are integrated in such a way, that it is possible to design a security strategy that meet the demands of the INAETICS target domains as well as the security risks that are inherently related to distributed and geographically dispersed systems.

Although security definitions in INAETICS are dynamic, they are built up from individual and discrete elements. Depending on the granularity of the implementation of these elements, it is possible to design

[12] More on declarative programming can be found at: http://en.wikipedia.org/wiki/Declarative_programming

a security model that is sufficiently fine-grained to counter the types of threats that have been identified.

The elements of the INAETICS security model are based on the following starting points:

1. The design does not introduce new structural parts to the INAETICS fabric layer. Much more it aims to decorate the software architecture in such a way that dynamic security enforcement can be achieved without cluttering the actual implementation space with undesired details.
2. A zone is defined as the basic security concern[13]. That means that any software part that is considered to be a member within that zone can participate in secure communications within that zone.
3. Trusted connections and encrypted communication across multiple zones can only be attained at module level and not by individual components. For this purpose, each module features one or more components that are decorated with the appropriate security features. Any module can operate as a bridge between 2 disparate security zones as well as within a zone between 2 applications.
4. There is always a Security Manager, or set of Security Managers, that is responsible for defining and enforcing policies that can be used to establish trusted relations and encryption schemes.

Consequently, this can be illustrated using the following diagram:

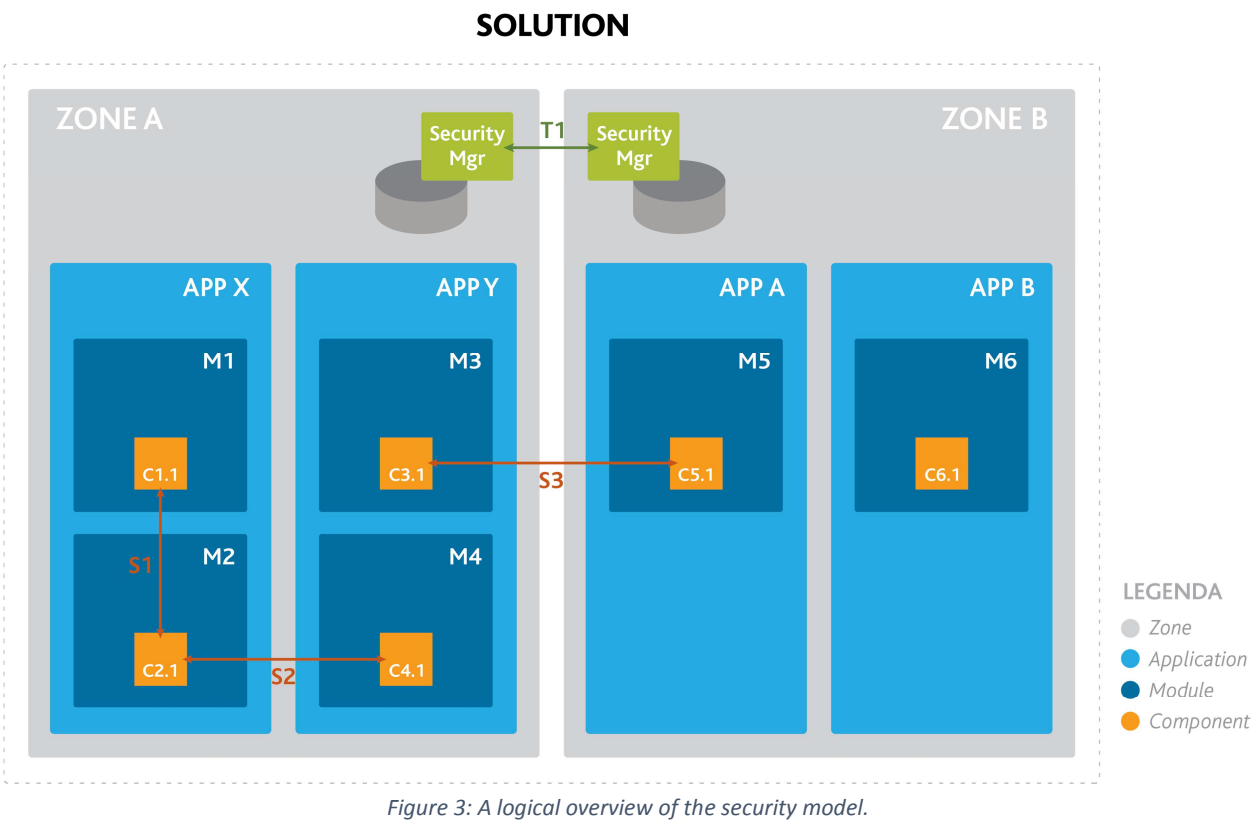

Figure 3: A logical overview of the security model.

[13] Therefore, there is always at least one zone: a default zone with a default security model.

In this example there are 2 zones, each containing two disparate applications with multiple modules; each module containing a number of components. Trusted and secure communication is only allowed at module level and is depicted using the arrows. In general, security and trust of communication is controlled by the INAETICS Security Manager. Suppose that in this example the Security Managers have the following policies:

1. Zone A and Zone B have a trusted relation;
2. Component C1.1 is able to securely communicate with Component C2.1;
3. Component C2.1 is able to securely communicate with Component C4.1;
4. Component C3.1 is able to securely communicate with Component C5.1.

Components C6.1 is neither accessible from the outside nor able to communicate beyond its module boundaries.

Every zone has a Security Manager that is local for that zone. This Security Manager has knowledge about the modules that exist in a zone and that are capable of maintaining a trust relation in terms of authorized access and encryption with other modules. Note that in this approach in order to be able to establish a communication channel between modules, applications or systems they need to use the Security Manager.
Whenever communication between multiple zones is required, a related number of security managers will be involved. However, because the Security Model was designed to be one security space divided into multiple zones, there is no designated single master in the security space. A mechanism of leader election and automatic discovery are the basic mechanisms for establishing this security space with multiple Security Managers.

4.2.3 The INAETICS coordination model

Given the dynamic capabilities of the INAETICS architecture, it is possible to design the runtime behavior of any INAETICS based solution. This is done using a number of coordination algorithms that can be adapted depending on the target domain. The INAETICS architecture aims to facilitate evolvability to support changed functional requirements, altered hardware capabilities or updated security threats or perimeters. Therefore, the most important lifecycle states of any INAETICS solution can be illustrated using the following diagram:

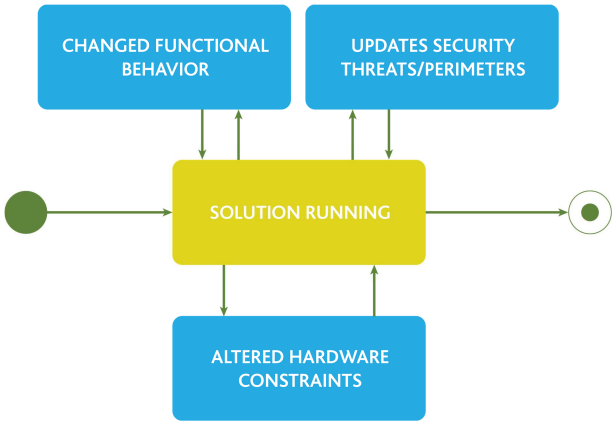

Figure 4: The various states of the lifecycle of an INAETICS solution..

As described in chapter 2 “Dealing with change”, change in the INAETICS architecture is split across to different spaces: a Problem Space and a Solution Space. Each of these spaces supports a specific form of control over variability. The Problem Space features ‘Intent-based coordination algorithms” and the Solution Space features “constraint-based coordination algorithms”. Conceptually, these types of control over variability differ from each other in the type and timing of the control they have over evolvability. They are, however, implemented and deployed as distributed processes without centralized coordination or state, which ensures they are as scalable and reliable as the coordination-space they control.

Intent-based coordination algorithms are designed using one or more intent models of a solution, that can consist of a system or even a systems of systems. These algorithms integrate these intent models into one coordination space, enabling them to optimize the behavior of the related solution space to best meet the given demands. Thereby, these algorithms directly control the degrees of freedom of the solution space. Ultimately, these algorithms are controlled by either human users or decision-oriented actors.

Constraint-based coordination algorithms on the other hand control the evolution of the solution space. They are implemented to optimize a defined set of constraints given information they receive and collect from their environment. These algorithms are implemented in the INAETICS architecture as a single coordination space per system or application. Architecturally this type of coordination algorithms implements all mechanisms that are required to handle changed constraints such as resource parameters of a system. These changes all manifest themselves through software services. Therefore, the algorithms are based on the information that is published by these software services, independent of the type of abstraction they represent.

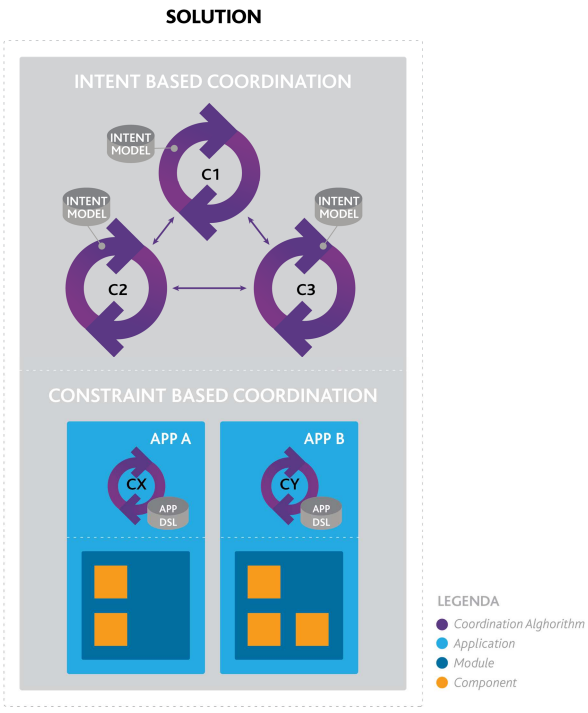

Figure 5: An overview of the coordination model.

4.3 Architectural Use Cases

The functionality that is typical of the INAETICS architecture can be grouped using the main architectural layers that were introduced earlier in this document: Coordination layer use cases, Container layer use cases and Fabric layer use cases.

4.3.1 Coordination layer use cases

The use cases of the the coordination layer that are relevant for the architecture are:

1. Coordinate changes in a system and related aspects that are caused by changes in constraints. This use case denotes the functionality that is specific to adaptability in the solution domain.
2. Coordinate changes in functional behavior that are caused by events that are not aligned with the governing Intent-model. This use case identifies the behavior that is relevant to (re)align a solution with the governing intent-model.
3. Coordinate changes in security policies or attributes (i.e. the security model). For example, after a security threat was detected somewhere in the system the security model can be adapted to prevent the threat from harming the system and simultaneously isolate it with minimum degraded performance.

4.3.2 Container layer use cases

The Container Layer of the INAETICS architecture implements the mechanisms required to control the lifecycle, context and isolation of applications and the associated components and services[14]. From that perspective, the container layer use cases that are relevant for the architecture are:

1. Deploy and manage a software part (see paragraph 4.2.1 for a typology of software parts). Given an appropriate security environment, it is possible to dynamically add or otherwise manage the lifecycle of (parts of) software. This can either be done manually or by one of the automated mechanisms such as dependency management[15] or executed deployment descriptors.
2. Manage characteristics of software parts.
   The capability of INAETICS solutions to dynamically adapt to changing environments and demands is based on coordination algorithms (this is based on a Qualities of Service mechanism –QoS-). Each software part has characteristics that describe something about its behavior or operation. Given the correct security definition, it is possible to change the behavior of any software part. This can be done by external actors as well as the software part itself.
3. Register and provide service.
   Services, which expose specific functionality that is implemented by a component, must be registered before they can be consumed. In this use case a service abstracts from

[14] Because of the composition approach in INAETICS, a strict hierarchy of nomenclature is defined to identify parts of a system. This terminology is further described in paragraph 4.2.1 The INAETICS component – services model.
[15] For a more detailed description refer to: http://www.osgi.org/wiki/uploads/Links/AutoManageServiceDependencies_byMOffermans.pdf

implementation details like networked versus local or synchronous versus asynchronous invocation.

4. Consume service.
   After a service has been retrieved, it must be possible to use (or consume) it. Like the previous use case, this use case is independent on the underlying implementation details.
5. Control service visibility.
   Finally, in order to control the use and visibility of services, it is possible to decorate any service registration with extra or changed attributes or security information. Using this mechanism, it is possible to actively control the evolvability of any service.

4.3.3 Fabric layer use cases

The Fabric Layer is a system of loosely coupled processing nodes which collaborate to provide a robust and highly dynamic distributed services infrastructure that can span organizational, technical and geographical boundaries.

The fabric layer is composed of:

1. Nodes, offering computing power and elementary actors in the fabric. Nodes are the first level of abstraction away from the physical device. It should be noted though, that one physical device can host multiple nodes.
2. Wires, implementing communication channels between nodes. Wires can express both the type of communication (e.g. synchronous and asynchronous) as well as addressing and other topology details.
3. Clusters, a number of processing nodes that are grouped in a bounded context. This context is based on various forms of computing affinity, e.g. in order to minimize networking or a shared security definition.
4. Zones, a group of nodes, either or not organized in clusters, with a certain scope. For example, services running with one zone are all within the same security zone.

So, ultimately the fabric consists out of a set of one or more zones which are able to interact in a secure fashion and respond dynamically to either functional, physical and security constraints.

The architecturally relevant use cases for the fabric of INAETICS solutions deal with managing nodes, clusters and zones. The basic capability that is implemented at this level of the architecture is to be able dynamically adapt to changing environments in a controlled way. The use cases are:

1. Manage nodes. It is possible to dynamically add, remove or otherwise manage nodes in the INAETICS architecture.
2. Manage wires. Managing interconnections between nodes is done using wires. Apart from administering the lifecycle of wires, it is also possible to manage their configuration.
3. Manage Clusters. It is possible to dynamically add, remove or otherwise manage clusters or nodes of clusters in the INAETICS architecture.
4. Manage Zones;

a. It is possible to dynamically add, remove or otherwise manage zones in the INAETICS architecture.
b. Clusters can be dynamically added and removed to zones. Either manually or automated.
c. Control security definition; security definitions in the INAETICS architecture can be changed dynamically (see also paragraph 3.4).

Appendix A: On service-oriented computing

The INAETICS architecture is designed for developing solutions using the Service-Oriented programming paradigm. To use this paradigm efficiently, a rich programming and runtime environment is required. This appendix elaborates on this environment by high-lighting a number of the most important services. The following figure shows high-level, hierarchical view of these services and their relations:

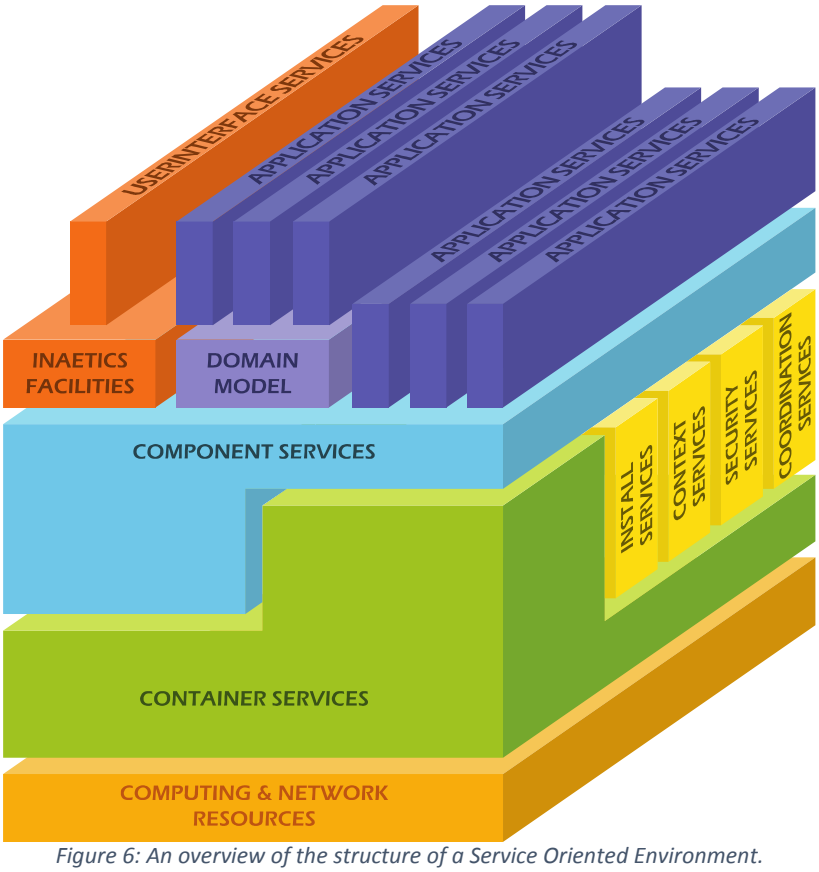

Figure 6: An overview of the structure of a Service Oriented Environment.

Container Services

One of the key components of a service-oriented programming environment are the Container Services. Within the INAETICS architecture, the container Services have access to processing and networking resources using a fabric computing based abstraction. A Container Service provides for isolation of applications and components as well as lifecycle management: starting / stopping component execution and restarting failed components. Other lifecycle features include starting components on system boot, live component updates, persisting component state, and moving components between containers. Component restarts or updates can be hot, warm, or cold. Starting the new component and switching all resources associated with the old component to the new component without interruption accomplish hot updates. Warm updates are accomplished by notifying the old component's resources to associate with the new component that has started. Cold updates are accomplished by stopping the old component, then starting a new one. Component mobility is the key enabler for balancing the processing load over a cluster of Container Services on different machines. It also facilitates agent

frameworks where a component can initiate a move to a different platform. Pausing its execution, capturing its state, transmitting the state to another container and restarting it makes it possible to move a component. Capturing the state of a component can be a complex task, involving internal state such as threads, or external state such as services being used and open files. Connections to other components must be terminated and renegotiated; files must be closed and reopened.

Clustering of Container Services facilitates load balancing and fault recovery. Container Services in a cluster cooperate to optimize the loading and performance of each, moving components as needed or even starting new component instances to ease the load on a heavily used component.
Furthermore, a Container Service can add or remove processing resources from a task or set of tasks. All of these load balancing and fault recovery features are based on system status and performance information that is provided by the underlying infrastructure.

Install services

In order to be able to support the runtime deployment of components from the ground up, a service-oriented environment should feature an Install Service with that capability. The concept of an Install Service inverts the normal installation paradigm. Typical install programs are bundled with the program being installed. As a result, these installers have to ask a series of questions about the platform they are installing the application on. This also poses a security risk in that unauthenticated software is allowed to run on the system unchecked.
The INAETICS Install Service or a dedicated part of it runs all the time. The Install Service interacts with the underlying infrastructure and can detect installation files or other events to initialize installation sequences. In this approach, the user does not need to be queried for platform details and the security signature can be validated before anything ever run on the platform. The Install Service can also resolve context-related attributes through a process of policy resolution. This resolution process allows for customization of the deployment environment of a component. The Install Service can act as a proxy to make components available to other platforms. The Install Service allows the registration of listeners for installation events. This allows the Container Services to instantly run applications after they are installed, if desired.

Component Services

Another aspect of a service-oriented programming environment is the family of Component Services. Component Services provide a Service-Oriented Programming (SOP) abstraction for service discovery and lookup. Within the INAETICS architecture, each self-contained unit of functionality can be treated separately. Component Services abstract from the details of discovery and lookup so that these components can easily communicate, whether they are in the same Local Area Network (LAN), connected by a Wide Area Network (WAN) or even running in the same process. Component developers are only concerned with a small set of standardized operations. For synchronous services this means providing and using services, removing a provided service, and discarding a used service. For

asynchronous services this means publishing and subscribing to services, unsubscribing a service and unpublishing a service.

Context Services

The INAETICS architecture is designed for the development of systems of systems. Therefore, it must be possible to dynamically (re)deploy components and services of the system to other parts of the system. This notion is captured in the term of a "deployment context". A context is designed to remove environmental details from components and thereby make it possible to deploy them in any environment.
As such, the use of context and a Context Service to manage them is an illustration of the metaphor of the wormhole, which was introduced as part of the underlying meta-model. Take for example a system that consists of 2 disparate technology environments: a deterministic and an on-line environment. Given the mechanism of the contexts it is possible to (re)deploy components of the system between environments that historically have a different design and implementation space. In the metaphor this is illustrated by the wormhole in the model: one doesn't have to re-implement components in order to be able to be used in a different environment.

Context Services are designed to provide an environment for Service-Oriented programming. Effectively, context services make it possible to control the lifecycle of contexts, configure their basic behavior; in terms of for example security and discovery mechanisms and link them by defining trusted relations between contexts.

When a context has not yet been created, components will reside in a default context. The default context provides a Service-Oriented environment for an isolated platform. Every system that is based on the INAETICS architecture will have a default context that contains a minimal number of services: a runtime platform with the core services mentioned earlier and an install service (or at least a local agent that can connect to an install service).

Coordination Services

The INAETICS architecture is designed to encompass different computing environments. Fundamentally, these environments differ in the way they view time and their strategies to work within the limits of the timing requirements. The INAETICS architecture was designed to be agnostic of time related strategies by introducing coordination services that enable dynamic change in scheduling and coordination policies.

An important part of coordination is based on Scheduling Services. Scheduling services define the level of granularity for the arbitration of computing resources. Scheduling policies dictate how much CPU time is allocated to tasks. The goal of any scheduling policy is to fulfill a number of criteria:
No task must be starved of resources - all tasks must get their chance at CPU time;
If using priorities, a low-priority task must not hold up a high-priority task;

The scheduler must scale well with a growing number of tasks, ideally being O(1)[16]. This has been done, for example, in the Linux kernel.

Security Services

The security aspect of the INAETICS architecture is based on two major mechanisms: A security and policy manager on one hand and various levels of attribute-based access control on the other.
The INAETICS security manager is a trust policy-based service that can create and maintain "fuzzy" trust relations between users and systems using a control mechanism that is based on policies and a number of (external) inputs. In this, inputs can be anything that can be used to determine the authenticity and trustworthiness of a system or user. Examples of such inputs are: credentials (username/password), personal tokens, certificates, and so on.
If we combine this service and the way it enforces trust with additional information in the form of attributes, such as, GPS locations, (cyber-security) threat-levels, and so on, it is possible to enforce more fine-grained decisions whether or not a system or user can be trusted and therefore can perform certain actions or access certain parts of the system.

[16] An O(1) scheduler is a kernel scheduling design that schedules using constant amounts of time, regardless of how many processes are running. See also: http://en.wikipedia.org/wiki/O(1)scheduler

Appendix B: On Intent-based design

Applying compositional technologies to develop large-scale systems brings all kinds of new levels of freedom and agility to these systems. However, these levels of freedom need to be designed and manageable. In order to solve this problem, numerous organizations have begun to experiment with a new type of solution definition that is based on capturing the intent of a solution as opposed to defining the exact implementation details of the solution itself. These intents can than be used as a basis to infer coordination strategies which in turn control the dynamics of a solution[17].

A successful application of Intent-based models finds its use in Software Defined Networks (–or SDN–). Here SDN Controllers determine how to translate the Intent the of network design into an infrastructure-specific "prescription" that causes the network to behave in the desired manner[18].

Metaphorically, an intent-based model can be explained as follows:
When you hire somebody to cut your lawn, you don't give them a list of all the blades of grass in your yard and the length to cut each one to (prescription), you tell them to make it look nice (intent) and they figure out the rest. Intent-based networking emphasizes the "cut my lawn" interface and is designed to move away from the "industry-standard CLI" model which is the "each blade of grass", traditional approach. Once the description of what is needed is separated from the details of how it's implemented, there are many benefits as described below.

Intent is invariant

Intent doesn't change as a result if (part of) a solution fails because of a crashing server, changing cloud provider, switch vendors, upgrading firmware or any other change to the underlying infrastructure. This invariant description frees solutions definitions from the underlying implementation details, simplifying overall development, testing, and deployment. If the expression has to change based on the state of the infrastructure, you have not yet captured the essential intent, which is infrastructure agnostic by design.

Intent is portable

Intent (describing the needs of the solution) is not specific to protocols, vendors, media-types or other implementation details. Because it is abstracted from changes to these details, intent-based modelling eliminates the impact of such changes.
With respect to the design of SDN's, intent-based modeling allows what enterprises, service providers and telecom carriers have been seeking; portability across a range of dissimilar solutions including the SDN controllers, eliminating lengthy applications integration changes and run-time complexity as a result of inevitable changes to the infrastructure.

[17] An example of applying intent models in network virtualization can be found at: http://researcher.ibm.com/researcher/files/zurich-DCR/An Intent-based Approach for Network Virtualization.pdf
[18] The text in this appendix is an adaptation of an article by SdxCentral about Intent-based network design. It can be found at: https://www.sdxcentral.com/articles/contributed/network-intent-summit-perspective-david-lenrow/2015/02/

Intent is compose-able

Intent-based approaches can be seen as a logical extension for the compositional strategy described in chapter 2. The extensible character of an intent-based model is designed to allow disparate services, developed independently, to express their resource requirements in a common language. As a result, all of the services accessible via the intent-driven interfaces share a common coordination strategy.

By sharing this coordination strategy, it is possible to design a solution for "split brain" and "multiple writer" challenges in distributed systems design. Any combination of intent-driven services can be used concurrently. In current SDN systems, one can run a third-party QoS service or a third-party security service, but not both. Building intent-driven systems provides a way to alleviate this problem by requiring that new services be built using the intent interface. Operators will be able to choose a la carte services to offer from multiple independent software developers with minimal risk to system integrity.

Intent scales out, not up

The intent of a solution doesn't change when you go from one infrastructure provider to another one. You can take a single description of intent and hand it to all of them. This enables a scale-out approach to designing, for example, solutions that span multiple domains simultaneously supporting small failure and maintenance domains concurrent with massive overall scale. Currently, many architectures assume building a scale-up system using a single, massive, clustered domain, which creates several operational and deployment challenges. By effectively sharding the collective intent across an arbitrary number of independent domains, we get end-to-end intent fulfillment with the superior survivability and fault handling of locally autonomous controllers.

Intent provides context

When different services of a solution expose low-level dynamics, there is always a risk of conflicting changes to the system state. Attempts to examine these dynamics and resolve such issues can prove to be unsuccessful because, at the level of abstraction of the individual service, it is impossible to decode the overall intent of the services exposing the dynamics. Because any intent-based model is designed to convey the why, rather than the how, it is possible to determine actual or apparent conflicts and seek ways to fulfill cumulative intents.
Promising research in the area of intent-based conflict resolution in multi-service SDN systems is being considered in standardization activities in the ONF[19].

We expect to see significant progress and easier-to-use, more capable solutions as a result of the intent-based work. Many groups working independently are now beginning to work together in standards organizations and open-source communities to create common intent-based approaches.

[19] ONF Stands for "The Open Network Foundation". See also https://www.opennetworking.org/.

About the authors

Hans Bossenbroek (1962), CEO of Luminis.
Hans has worked for various IT consulting organizations before he co-founded Luminis in 2002. He has a rich technology background and is widely acknowledged as a thought leader in the area of IT architecture; he is a regular speaker at international conferences and has been a member of the Dutch Java User's Group board of directors for several years. Prior to founding Luminis, Hans worked for various companies amongst others ATOS.

René van Hees (1965), Chief Software Architect at Thales Netherlands.
René has worked for several software companies in both the Netherlands and Germany before he started at Thales Netherlands in 2002. In his role of Chief Software Architect, he is responsible for all technological, process, methodology, architectural and innovation related aspects concerning the development of (real time embedded) radar sensor software.

About INAETICS

For the INAETICS architecture, the vision is based on the principles that evolution can be designed into a system and that it is possible to drive the development of INAETICS-based systems using one overall architecture style. This architecture style, which is based on a dynamic services architecture, requires both a modular development strategy as well as a rich and robust infrastructure to support a cost-effective development process. The INAETICS project not only aims to deliver a fit-for-purpose architecture, but also an initial implementation of the infrastructure and demonstrate the usability using a number of domain-related examples.

The INAETICS project is a collaboration between a number of organizations, based on the principles of Open Innovation. The INAETICS partners collaborate and share their expertise in order to realize a shared vision of an open and robust architecture for the next generation of time critical systems. INAETICS is publicly funded by the European Union and the government of The Netherlands.

More information can be found on www.inaetics.org

investing in your future
European Regional Development Fund
European Union

Gelderland & Overijssel
Gebundelde Innovatiekracht