Software Architecture evolution in an Open World

Introducing the INAETICS project

inaetics.org

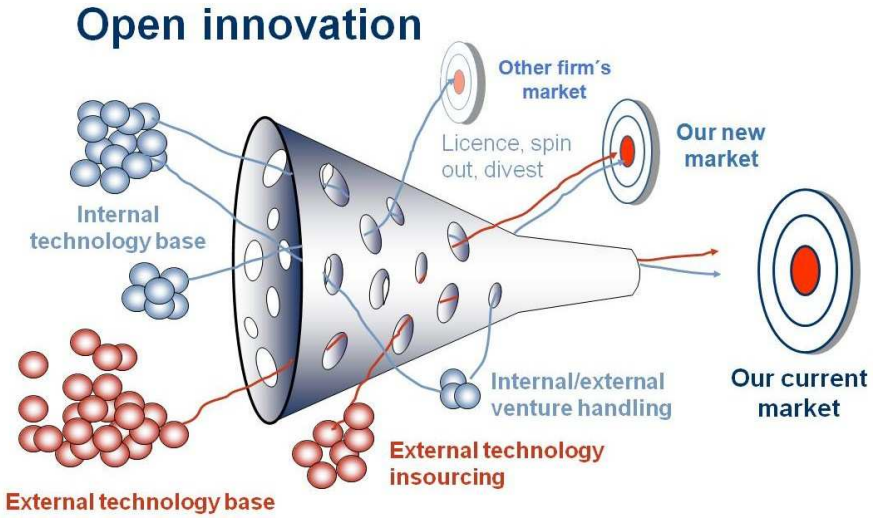
Auteurs
Hans Bossenbroek
René van Hees

Inhoudsopgave

1 Introduction

In recent years, the speed of innovation has increased considerably due to a number of catalysts. Among these, the Internet and more specifically the strategic impact of information technology play a key role. The combination of the increased speed of innovation and the evolution in information technology has confronted companies with a software intensive strategy with a new challenge: how to compete in a market where the external speed of technological evolution is considerably higher than can be attained by the own organization.

In 2003 Henry Chesbrough coined the term 'Open Innovation'[1] that suggests that the solution to this problem lies in cooperation and co-creation instead of smarter, closed R&D. Since this innovation strategy was conceived, a number of best practices have emerged. The latest of these best practices is to acknowledge the fact that the competitive advantage of software intensive strategies is not to have an excellent architecture but much more to be able to deliver domain-specific and competition enhancing features as fast and agile as possible.



Yet changing the innovation capability of organizations to have more collaborative culture is not the solution in itself. Over the past years, the structure of the IT landscape has undergone a deep change. Over the past years, Moore's Law defined how computing technology evolved. Yet, with the ubiquitous character of networks and the Internet as the ultimate backbone, Metcalfe's Law has become the relevant driver. Metcalfe's law states that the value of a network is equal to the number of nodes of the network squared[2]. This is true for the same reason that people can achieve amazing feats when they work together; witness the pyramids, the Space Shuttle, and even the shopping mall. When computers work together, they enrich the interaction of people with their environment and other people. The time is coming when all kinds of hardware and software will integrate seamlessly and become a natural part of a person's environment. Today's "born to surf" generation will give way to the "born in the web" generation.

Current software architectures do not provide the consumer with the most benefit from this connectivity; it is very difficult for people to set up, manage, and interact with networked devices around them. Networked systems today are static and brittle; they usually consist of a collection of stovepipe client-

[1] See also: http://openinnovation.berkeley.edu/what_is_oi.html
[2] Robert Metcalfe is one of the driving forces behind the Ethernet standard. See also: http://en.wikipedia.org/wiki/Metcalfe's_law

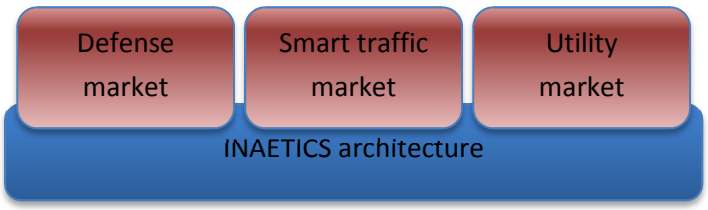
server applications where the clients only communicate with particular servers, with particular protocols, in very specific environments.

1.1 The INAETICS project

In late 2012, a number of companies with a software intensive strategy started project the INAETICS[3] project. This project is aimed at enabling the participants to harness the changed speed of evolution in information technology using an open, shared architecture. The INAETICS project is a publicly funded collaboration where a number of industry-parties collaborate to design an open and robust architecture. The project elaborates on a number of previous and existing initiatives like the STARS[4] project and Amdatu[5].
This whitepaper highlights the vision and underlying design of this shared architecture, whereby the individual participants of the project can excel at their strategy by applying it in their strategic domains and related technologies.

In order to show the possibilities of open innovation as an enabler for the development of competitive features with an open and shared architecture and technologies the project aims to produce a number of demonstrators that are linked to domains of interest. These different domains of interest are related directly to the different participants of the project. Each of these domains is served with different products that require a high level of time-to-market that can be attained by sharing an open and enabling architecture. The INAETICS project aims to prove this by:

- *Using this architecture to implement a number of illustrative demonstrators that are relevant in the domains of the various participants*
- *And use it as a basis for the development strategy for each of the projects participants.*



[3] See also: http://www.inaetics.org and http://www.inaetics.org/partners for more information about this project.
[4] See also: http://www.starsproject.nl
[5] See also: http://www.amdatu.org

2 The INAETICS architecture

Acknowledging the relevance of agile, open innovation and speed as extra requirements to survive in the current markets, poses a new challenge for people that work within software intensive organizations: how to design or help emerge[6] an architecture that supports the existing requirements, that is inherently linked to a business domain, and meets the increased evolvability requirements.
The INAETICS project works on the assumption that it is possible to design an architecture that meets these requirements for a specific type of systems. These systems have the following characteristics:

- **Multiple control strategies** – First and foremost, each of these systems features multiple control strategies. An illustrative example is the combination of an on-line or human interaction perspective with a deterministic behavior.
- **Functionally adaptive behavior** – Another characteristic of these systems is that their functional behavior is context dependent and must be able to evolve at runtime without a structural need for reconfiguration or re-installation.
- **Technologically heterogeneous** – Furthermore, in order to optimize the versatility of these systems they are expected to be infrastructure agnostic.
- **Geographically dispersed** – The business domain of these systems not only dictates that they run in geographically dispersed locations, they should also be dynamically relocatable.

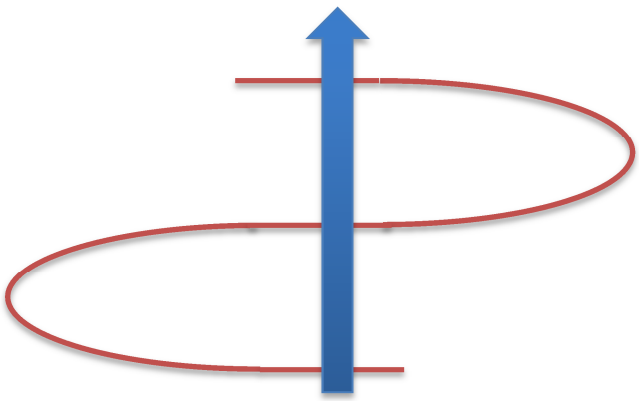
The INAETICS architecture is based on the assumption that it is possible to identify one architecture style that abstracts from all underlying computing-paradigms. This results in a single design and implementation space with the a number of architecturally relevant patterns and mechanisms.

Designing an architecture that is fit for purpose for the INAETICS participants is a matter of balance. The INAETICS architecture must balance between extensibility towards domains and yet have a sufficient technological richness to deliver an acceptable level of out-of-the-box functionality. This document is not intended to be an in-depth and complete architecture design. Much more, it is intended to describe the overall coherence of the project by means of a shared architecture. This is achieved by outlining those aspects of the architecture that constitute the most important design and technology choices. For this purpose we will use a qualitative way of describing the INAETICS architecture. This description consists of a number of different perspectives, each high-lighting a specific characteristic of the project.

In order for the INAETICS architecture to be successful, it needs to be scalable for software development organizations (ease of adoption) and have a sufficient external adoption. Based on these criteria and the challenges outlined earlier, we identified 2 major principles underpinning the INAETICS architecture:

- **Consistency is the overall guiding principle of the architecture.** Over the last years it has become clear that consistency and continuity throughout the development process is one of the most important factors of developers' performance. Whether with the use of continuous integration approaches or a successful YAGNI[7] strategy that is based on a shared understanding of the system and its architecture, it is vital to prevent translation and interpretation in the development process as much as possible.
- **No architectural impedance intermediaries.** Typically, the INAETICS architecture shares a number of disparate types of systems: for instance deterministic, on-line user interaction and machine-to-machine. Historically, these system-types translate directly into bespoke architectures. Within INAETICS, the architecture is designed to be agnostic of this technology-push and defining a continuous design and implementation space (or architectural style).

[6] The discussion how to arrive at an architecture, by design or through an emerging approach, is outside of the scope of this whitepaper or the INAETICS project.
[7] See also: http://en.wikipedia.org/wiki/You_aren't_gonna_need_it



We decided to use a metaphor to express these principles of the architecture that is loosely based on the notion of "wormholes"[8]. In this illustration, the red line represents the various types of architecture that can be found in a large system; an analogy with the warped space, and the blue line is a specific feature that is to be implemented; similar to a wormhole.
The INAETICS architecture states that we aim to take the wormhole to travel from place to place instead of the long way, the latter route in analogy with the practice of developing architecture impedance intermediaries. A familiar example that illustrates the use of wormholes as a guiding metaphor for software construction is the problem of the 'impedance mismatch' of object-relational mapping software[9]. The underlying problem of storage strategy that does not match with the implementation technology in one application is only partly solved by introducing an intermediate layer: the so-called object-relational mapping layer. The problem with object-relational mapping layers is that they oftentimes evolve into system-wide data layers that contain unwanted levels of domain-related knowledge. This results in unsolicited forms of inflexibility. The INAETICS architecture is based, amongst others, on the 'stewardship-of-information' principle. This means that each individual component in the system is responsible for its own information, so that accessing this information does not require an intermediate mapping layer but just a call to the correct service that represents the information.

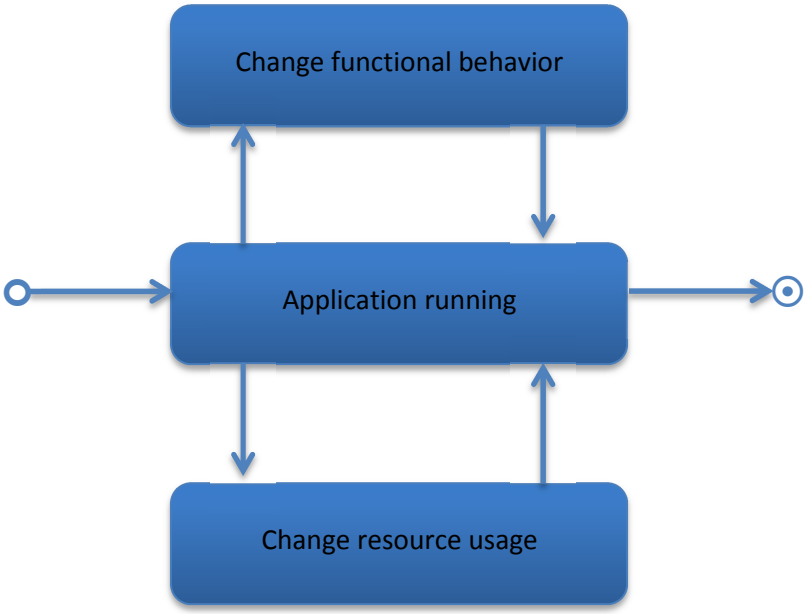
[8] See also: http://en.wikipedia.org/wiki/Wormhole
[9] See also: http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch

3 A dynamic services architecture

Assuming the consistency principle of the architecture drives all system development, it is mandatory to base this on one overall architecture style that enables runtime evolution. This architecture style is referred to as a ”dynamic services architecture[10]”.
Effectively, a dynamic services architecture dictates that any system:

- Must be divided into small, independent modules or components, where:
  - Any module can only be accessed through one of its defined services.
  - Each module is self-contained, and does not depend on the context or state of other services.
  - The assembly of the modules and services, the system, can vary at runtime.
- Services are a core concept in these architectures, because:
  - They represent the only exposed interface of a unit of functionality (module or component) that can be accessed by other services.
  - They have a lifecycle that is differs from the encompassing system. Because services can show-up and disappear at runtime it is possible to support runtime evolution of the system that is different from the deployment lifecycle of the system.

Given this architecture, where everything is represented by services, it is possible to design the types of runtime evolution of any system. Within INAETICS, services are used to abstract from software modules and hardware resources. The evolutionary states of any INAETICS system, therefore, can be simplified as follows:



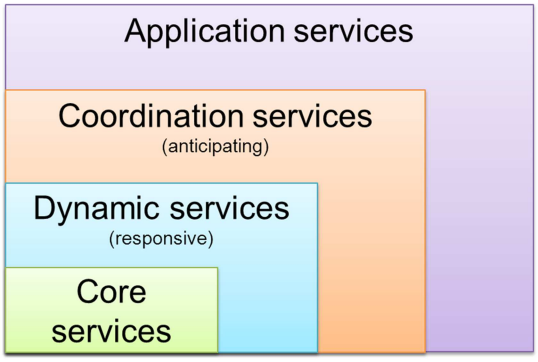
Assuming that any system can be represented by a dynamic set of services, it is important to categorize the different levels of evolution in a system. These levels of evolution are grouped into the following categories of services:

- **Core services** - Core services are considered to be the minimal set of services that are required for any independent part of a system to control its lifecycle. This means that these services

[10] As a reference to his type of architectures, one can refer to OSGi. The OSGi standard is maintained by the OSGi consortium. It defines an implementation of a Java-based dynamic services architecture. See also: http://www.osgi.org

represent the first set of software-modules that are to be deployed and must remain present on any part of the system.

- **Dynamic services** - In order to respond to changes in the context or control the behavior of the system for other reasons a category of dynamic services exist. These services are designed to detect changes in the context and can be used to force changes in the services that are deployed at any given moment.
- **Coordination services** – Based on first 2 categories of the system, it is possible to implement systems that consist of networks of subsystems. In order to distribute behavior over these subsystems, it is important to support various coordination strategies. Within INAETICS, coordination strategies consist of scheduling policies (e.g. process priority or cost-based economic policies) and concurrency policies (e.g. agent coordination, massive or symmetric multi-processing). The services enable the use of various coordination strategies as well as the associated management and control algorithms.
- **Application services** – Finally, the architecture acknowledges application services that implement the actual functionality of the system.



Irrespective of the design strategy of this architecture, where we aim to attain a high level of runtime dynamism by minimizing the footprint of implementation technologies, there are still a number of techniques or assemblies of techniques that define core capabilities of a system. Within the INAETICS architecture, these assemblies of techniques are referred to as 'Architecture Mechanisms'. The key mechanisms of the INAETICS architecture enable designers and developers to divide a system into a number of smaller (sub)systems[11], to define a coordination strategy for each individual (sub)system as well the overall system and enforce specific security structures and strategies on the system. Each of these mechanisms is introduced in the next paragraphs.

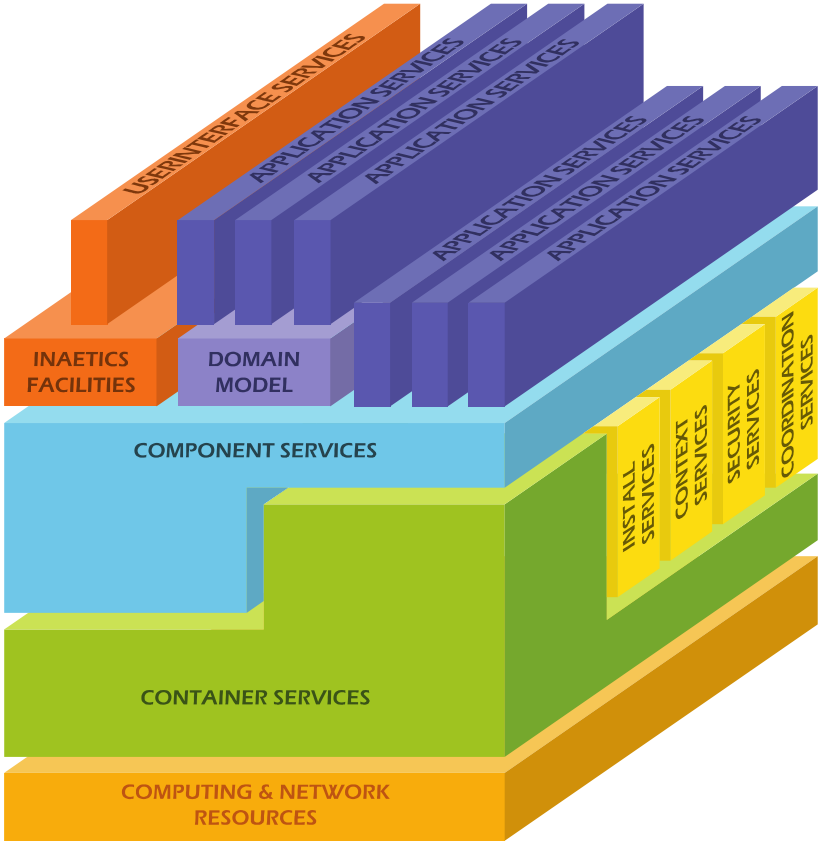
[11] See also: http://en.wikipedia.org/wiki/System_of_systems

4 Architecture overview

The goal of the INAETICS architecture is to provide a Service-Oriented Programming framework for highly dynamic networked systems of software and hardware components. The focus of the architecture is components, which use and provide discoverable services. Every component is a manageable, reusable encapsulation of functionality that can be dropped into any context because they are independent of platform, protocol, environment and database. Systems are self-forming, self-healing and highly available.
INAETICS is designed to minimize the influence of execution technologies, by providing plug-ins and abstractions of these technologies. The architecture will be used in a variety of ways during the development of a product or system:

- System engineers use the architecture to design and prototype component-based systems
- Software developers use the architecture to develop reusable components
- System managers use the architecture to control system deployment
- Users will experience true hardware/software plug and play and new levels of interoperability between the devices they use.

The INAETICS architecture provides for an infrastructure for Service-Oriented Programming with a number of mechanisms. The most important services of these mechanisms are: Container Services, Component Services, Context Services, Security Services and Install Services. The following figure shows a high-level view of the INAETICS architecture:



4.1 Container services

One of the key components is the Core Services of INAETICS are the Container Services. Container Services provide processing resources through a virtualization abstraction. Every INAETICS platform contains a Container Service that controls access to the platform's processing resources. The Container Service provides control of 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.
An additional feature of the Container Service is that it provides the ability to add or remove processing resources from a task or set of tasks. Finally, the Container Service is a component that provides system status information, such as CPU loading and free memory. This is the same information used to perform load balancing in a cluster of Container
Services.

4.2 Install services

In order to be able to support the runtime deployment of components from the ground up, there is an install service designed in the INAETICS. The designs of these Install Services invert 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 on the system all the time. The Install Service is aware of settings on the platform and can detect installation files on new media inserted on the platform. 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 parameters through a process of policy resolution. Policy resolution allows deployment environment information to be customized for the 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.

4.3 Component services

One of the key components is the Core Services of INAETICS are the Component Services. Component Services provide a Service-Oriented Programming (SOP) abstraction of service discovery and lookup. Within the INAETICS architecture, every self-contained unit of functionality is treated as a component, whether it is hardware or software based. Hardware components are always wrapped in software. Component Services abstract from the details of discovery and lookup so that 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 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.

4.4 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, this is an illustration of the metaphor of the wormhole, which was introduced as part of the underlying metamodel. 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 concurs with the INAETICS architecture will have a default context that contains a minimal number 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).

4.5 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.

Scheduling services is the first group of coordination services. Scheduling services define the most fine-grained policies for the arbitration of computing resources. A scheduling policy dictates how much CPU time is allocated to tasks. The goal of any scheduling policy is to fulfil 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)[12]. This has been done, for example, in the Linux kernel.

Concurrency services constitute the second group of coordination services. They are aimed at coordinating the synchronization between groups of tasks or (sub)systems. Within the INAETICS architecture, this is implemented using different concurrency policies. With concurrency policies, it is possible to control the way resources are utilized by complete (sub)systems. Concurrency policies enable developers to express the way tasks process work. Using different concurrency policies, it is possible to fine-tune the system's behaviour under stress of for example large volumes of data or other external incidents.

4.6 Security services

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[13], meaning that security is taken into account at all times, based on the assumption that malicious attempts will take place. Furthermore, the security services support focus on transport, service and mobile code security.

There are two key services in the INAETICS Security architecture: the Authentication Service and the Authorization Service. The Authentication Service handles validation of certificates. The Authorization Service allows the user to dynamically manage access control to the system by simply defining roles, assigning permissions, and establishing relationships with other authentication services. The INAETICS architecture enforces three types of security: code security, transport security, and service security. Code security is first enforced by the installation service, which validates installation components. Next the underlying infrastructure environment controls the access the newly installed component has to the local platform. Transport security between services is enforced using secure connectors. Secure connectors provide confidentiality and integrity. The underlying runtime infrastructure and the component services jointly enforce service level security. This allows access to distributed services to be controlled.

[12] 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

[13] 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/Secure_by_design

5 Conclusion

The goal of the INAETICS project is to increase the economic opportunities of systems that have an inherent time-critical (often real-time) and safety dimension by designing a fit for purpose and cost-effective architecture. We believe that the success of the INAETICS project lies beyond the specific architecture and is rooted in an active Open Innovation strategy.

The main purpose of the 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 time to market as well as the increased knowledge and experience that is gained by combining scarce resources. 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, 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. This core of this infrastructure will be made available as Open Source under the Amdatu project[14].

[14] Amdatu is an Open Source project aimed at fostering Open Innovation projects
See also: http://www.amdatu.org/philosophy.html.

About the authors

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.

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 as principle technology consultant at ATOS Origin.

INAETICS

investing in your future
European Regional Development Fund
European Union

Gelderland & Overijssel
Gebundelde Innovatiekracht

INAETICS is funded by