# ANA Project

Autonomic Network Architecture


Sixth Framework Programme
Priority FP6-2004-IST-4
Situated and Autonomic Communications (SAC)

**Project Number: FP6-IST-27489**

Deliverable D.2.4

# Initial Design and Prototype Implementation of the Functional Composition Framework (includes software)

Version 1.0

# ANA Project

## Autonomic Network Architecture


| | |
|---|---|
| **Project Number** | FP6-IST-27489 |
| **Project Name** | ANA - Autonomic Network Architecture |
| **Document Number** | FP6-IST-27489/WP2/D2.4 |
| **Document Title** | First Draft of the Functional Composition Framework |
| **Workpackage** | WP2 |
| **Editor** | Manolis Sifalakis (ULancs) |
| **Authors** | Manolis Sifalakis (ULancs)<br>David Hutchison (ULancs)<br>Andreas Louca (ULancs)<br>Lorenzo Peluso (Fokus) |
| **Reviewers** | Stefan Schmid (NEC) |
| **Dissemination level** | Public |
| **Contractual delivery date** | 31st December 2007 |
| **Delivery Date** | 15th February 2008 |
| **Version** | 1.0 |

**Abstract:**

This document describes the design initial design of a functional composition framework for the network subsystem of the ANA Node. Two independently developed, yet orthogonal and complementary components are detailed. They may be used in combination to provide flexible runtime adaptation, which is a fundamental requirement for achieving the autonomic goals of the project. The first is the composition mechanism which enables the flexibility to modify or extend the functionality of the ANA Node. The second it the information sensing and sharing framework, which by enables system awareness and can be used to drive/trigger adaptation. The combination of the two may be used to carry out autonomic optimisations and reconfiguration tasks. An analysis of the building blocks is presented and wherever possible we provide implementation details and evaluation results.

**Keywords:**

Functional Composition, Information Sharing, Event-based systems, Information collection, System Awareness

# Executive Summary

The goal of the ANA project is to explore novel ways of organising and using networks beyond legacy Internet technology. The ultimate goal is to design and develop a novel network architecture that can demonstrate the feasibility and properties of autonomic networking. As specified in the description of work, it is the intension of the project to address the self-* features of autonomic networks such as auto-configuration, self-organisation, self-optimisation, self-monitoring, self-management, self-repair, and self-protection.

In order to meet the self-* goals set, it is important to enable flexible adaptation of the ANA node operation in response to the application requirements (mission) and imposed operational environment constraints (administrative policies, network conditions, system limitations, etc). At the same time collaborative functionality should be easily achievable among a compartment's members so as to provide both redundant and/or collective servicing.

To this end the aim of this deliverable is the description of the design and the prototype implementation of an architecture that can *enable* and *drive* adaptation. The composition framework and its comprising building blocks is the enabling technology that permits the inclusion or exclusion of functionality, or the re-configuration of the existing one. The information sensing and sharing framework is the means for igniting and triggering adaptation by enabling event-based awareness in a system (of its state, the external operational environment, and its mission). The current prototype abides and complies with the ANA Node Blueprint (D1.4/5/6v1) and the previously investigated requirements of this component of the project (D2.2).

# Table of Contents

# 1 INTRODUCTION

The main aim of this task is the design and development of the framework that will constitute the basic network subsystem in the ANA node, by means of which it will be possible to dynamically and flexibly integrate new and evolving network functionality and carry out autonomicity-driven reconfiguration. During the course of the past 18 months of the project this task has focused on the following areas:

- A methodical analysis of the lessons learned from traditional strict-layered architectures in order to identify the cases where layering provided an acceptable solution, and where this model is deemed insufficient for autonomic architectures such as ANA. From the results of this analysis, it has been decided which functions, abstractions and other aspects of current models, need to be incorporated in the design of the ANA communications system. This has led to a design of a prototype framework that enables functional composition in ANA and which is presented in this document.
- A study of various research proposals from the literature on cross-layering and information sharing, in order to understand how to best exploit their properties and usage patterns in our design. The results of this study are reflected in the prototype design of an event-driven architecture that provides system and network awareness and which can be used to drive or trigger the functions of (re-) composition in autonomic architectures.
- We have developed independent implementations of both systems and carried out initial evaluations of their operation, feasibility and flexibility in enabling the functionality required for the purposes of ANA. The composition framework has so far been integrated in the ANA software providing a very basic functionality and is currently being tested. Its functionality will be enriched in the near future through a more sophisticated classification engine from the literature [-11]. The information sensing and sharing architecture having more complex requirements, although implemented in its core functionality, it has not yet been integrated in the ANA software, but has been comprehensively tested through simulations for its conformance to the requirements. Its integration with the ANA system is also eminent in the course of the project.

In accordance with conceptual work carried out in tasks 1.5 and 3.1, the work carried out in this task can interface and use (albeit not rely on) the components developed in these tasks to drive composition and enhance the ANA system's ability for autonomic behavior.

Furthermore the designs presented in this document have taken into account requirements from tasks 2.1, 2.5, 3.2 and 3.3 to enable the expected degree of flexibility for introducing functions, mechanism or strategies for facilitating optimization tasks, servicing resilience needs, and customizing the communication transport between nodes.

## 1.1 Scope of Deliverable

The aim of this deliverable is has been the analysis, design, prototype development and evaluation, of a functional composition framework that will leverage and enable the flexibility and functionality in the ANA architecture. The prototype abides to the specification of the ANA Node Blueprint (D1.4/5/6v1).

To serve the autonomic purpose advocated in the ANA proposal it is important that functional composition and cross-layer control loops reflect the need for context-awareness, dynamic adaptation, re-configurability and autonomic optimisation of the systems operation with regard to its mission, application, administrative policy above, as well as to the communication environment an traffic state below.

To this end it has been within the scope of this deliverable to explore the need and propose prototype mechanisms by which the ANA node can sense its operational environment, and facilitate optimizations in its basic operation, as well as investigate the type of information needed to drive these optimizations and consider mechanisms for collecting and using the required information.

## 1.2 Structure of the document

This document is organized in the 5 sections and two main parts of work (delivered in section 3 and 4). Section 1 provides an introduction to the objectives and aims of this deliverable, defines the research scope of task 2.2 and describes how it relates to other tasks of the ANA project.

Section 2 introduces the terminology used in this deliverable, abbreviations and definitions.

Section 3 introduces the design and discusses in more detail the prototype implementation of the composition framework, which enables functional configuration and composition in ANA.

Section 4 introduces and explains the design and prototype implementation of the information sensing and sharing architecture for providing awareness and drive the functional composition process.

There has been an effort to preserve a relatively high level goal-driven description supported wherever required by technical examples, formal descriptions and only wherever needed implementation details so as to avoid flooding the reader with technical details that may hinder the understanding of the intensions and goals of this work. At the same time as we don't intend to prescribe a "final" solution at this stage –the project is only half way through- we consider inappropriate to detail a technical annex.

Finally section 5 concludes this deliverable, by summarizing how the operation of the two components couples to enable autonomic behavior in the ANA node architecture.

# 2 TERMINOLOGY

## 2.1 Abbreviations

| | |
|---|---|
| ANA | Autonomic Network Architecture |
| API | Application Programming Interface |
| BP | Bootstrap Protocol |
| FB | Functional Block |
| IC | Information Channel |
| IDP | Information Dispatch Point |
| IDT | Information Dispatch Table |
| MC | MINMEX Controller |
| FC | Functional Composition |

## 2.2 Definitions

**Identifiers:**

- An Identifier consists of a finite sequence of symbols of a given alphabet.
- Identifiers are used for identification of an entity within a set of entities (e.g. to single out one or more objects from a set of objects).
- Identifiers are typically persistent and globally unique within a given identifier space.
- Within ANA, identifiers are used to identify objects, resources, etc.

***Names:***

- A Name is a globally unique, persistent identifier used for recognition of an entity (e.g. object, resource, and host). For example, a name can be used to resolve the address/locator of the entity.
- Within ANA, names are used to identify an entity (e.g. object, resource, and host). Names can thus be used to resolve the identifier or address/locator of an entity, which is necessary to gain access or communicate with the entity.

***Addresses:***

- An Address is an identifier that is used for routing and forwarding.

- Addresses entail the information that is needed to transport data/information from a source to a destination.
- In case of structured addresses, which define the location of an entity within a topology, addresses are also called Locators.

**Functional Blocks (FBs):**

- FBs are the information processing functions in ANA.
- They generate, consume, process, forward information.
- FBs run on one node only, which means a node has full control over the functional blocks it hosts.
- They can have zero or more input and output.

**Information Channels (ICs):**

- ICs are the channel/medium over which communication between functional blocks takes place.

  Note: ICs can be logical, i.e. they can span across multiple nodes: it is a kind of "distributed object" (made of FBs and other "lower" ICs).

**Information Dispatch Points (IDPs):**

- IDPs are the entities that provide decoupling for ICs and FBs. That is, access to an FB or IC is done via the IDP it is bound to.
- IDPs allow dynamic (re-)binding of FBs and ICs in a way that is transparent to the "client" of the FB or IC. That is, the entity bound to an IDP can be changed but the IDP used to interface with the entity remains.
- IDPs perform no processing except data forwarding (actual processing of data is performed by FBs).

**Compartments:**

- Compartments are non empty sets of Functional Blocks (FBs), or non empty sets of composed FBs (where a composed FB is two or more FBs hosted by the same ANA node), agreeing on some common set of operational and policy rules (the "recipe").
- The common recipes are typically the communication principles, protocol(s) and policies to be used.
    - Examples of common communication principles: how naming, addressing, routing, etc. is handled
    - Examples of protocols: communication protocols between peer FBs on different nodes, etc.

    - Examples of common policies: membership (join/leave) procedures trust, etc.

- FBs forming a compartment communicate through Information Channels (ICs). ICs can be seen as abstractions of "connections" or "communication channels" through underlying compartments. In case there are no underlying compartments, ICs represent the underlying communication channels (outside the ANA world) that are used for transferring information between FBs (e.g., an inter-process communication channel, a L2 channel).
- Some compartments may require a minimal set of (composed) FBs in each and every participating ANA node. This is considered again as part of the compartment policies. Those composed FBs are closely related FBs interconnected through IDPs.

# 3 ENABLING AUTONOMIC BEHAVIOR AND FUNCTIONAL EVOLUTION

## 3.1 General

In summary, the motivation for the work presented in this deliverable lies on the need for adaptation as a fundamental enabling capability for autonomic operation as argued in [3]. The feedback loop in the lifetime of an autonomous system (be that a network or a node) follows the pattern:

*Sense* → *Infer/Decide/Select* → *Adapt* → *Sense*

Or

*Sense* → *Adapt* → *Sense*

The top control loop refers to autonomicity that lies on intelligent or evolutionary processes that lead to what we have called in D2.2 "long term" adaptation. If we were to a draw an analogy to the physical world, this would correspond to the conscious thinking-acting process of a human brain, or the way that evolution is driven in nature. It has to do more with the ability of a system (physical or artificial) to modify its internal structure, and add or remove traits/features

The bottom control loop on the other hand, refers to autonomic behavior that corresponds to hard-wired processes, or what we have called in D2.2 "short term" or "fast" adaptation. In the analogy we did, this would correspond to instinctive behavior of a human or an animal in response to instincts and intuition. Therefore it mostly refers to autonomously adapting *behavior* by optimizing the existing structure.

As part of the work carried out in this task, in the preceding chapters of this deliverable we have tried to model and capture the semantics and operation of the *Sensing* and *Adapting* processes and enable them in ANA.

## 3.2 Sense and Adapt Capabilities in ANA

**Error! Reference source not found.**, captures the essence of the work in this task. It illustrates the envisioned capability that the two main activities in this task, opt to enable, and illustrate how the functional composition framework and the information sensing and sharing architecture integrate and interact with the remaining of the ANA architecture.

Both the composition framework and the information sensing and sharing architecture reside somewhere between the data plane and the control plane. They both operate as functional blocks (abiding to the broader paradigm of ANA) in the ANA playground, and interact with other functional blocks in the playground through the MINMEX message broker.

Starting from bottom to top in **Error! Reference source not found.**, functional blocks that collect information in the control plane (out-of-band) or in the data plane (in-band), provide input to the information sensing and sharing architecture. In return the ISS system is able to generate various different notifications (that correspond to different ways of correlating that low level information into high level -composite- event patterns). These notifications are delivered to subscriber functional blocks either in the data plane or the control plane.

Functional blocks in the data plane are typically participants in the data path processing (function composite) of data traffic and which are able to carry out optimizations e.g. by updating configuration parameters in protocol functions.

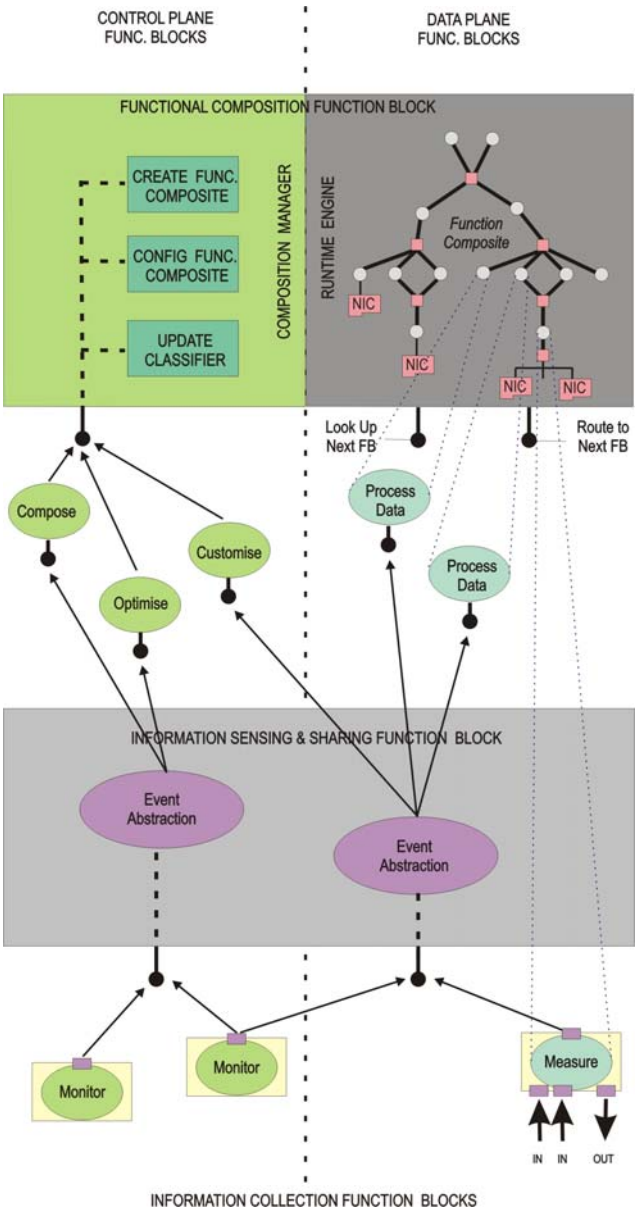



**Figure 1. Task 2.2 inside ANA**

Functional blocks in the control plane correspond to higher level inference logic that upon reception of appropriate input information, they may choose to enable/disable a function, re-compose or modify the data path processing (function composite), or create a new functional context (functional composition instance) in response to an ANA compartment instantiation.

Finally all these dynamic operations are made possible through the functional composition framework functional block, which hosts different collaboration schemes for data plane functional blocks, and exposes a control plane API for the management of these collaboration schemes in the context of different compartments.

Note that such a collaboration scheme, which captured in the design and implementation of function composite classifier does not include the data plane functional blocks themselves (it does not provide any sort of execution environment), instead the implementation of a classifier implements the rules and principles that govern their collaboration.

# 4 FUNCTIONAL COMPOSITION FRAMEWORK

## 4.1 Overview - Current Architectural Limitations

Fixed network architectures such as the TCP/IP have thus far served technological advances in the Internet. However, and although robust and more controllable, the static nature of current network subsystems has been also problematic. First of all in facilitating fast evolution to serve emerging application requirements and second for facilitating adaptation/customization to different or changing operational conditions, which has been a requirement of many modern application domains (sensor networks, mobile networks, etc). This imposes a fundamental restriction in achieving convergence of different network technologies as well as on fulfilling emerging goals (protection, resilience, QoS, etc). On the other hand this the exact essence and long term objective of autonomically behaving systems.

The degree of self-management expected in an autonomic system imposes strong demands for a capacity to modify the functionality of the data plane or control plane; in occasions fast and effectively at runtime in response to emerging events, or mission changes. Although this is often achieved to some extend through a reconfiguration capability (cross-layering) or the capacity for incremental addition/removal of functions (active networks), we advocate a more holistic approach by offering the ability to restructure a data/control plane to satisfy different requirements criteria or serve different semantics.

For many projects that take a "*clean-slate*" architectural approach, e.g. [1] to enable autonomicity, this leads to a radical shift in design philosophy. More specifically, at present applications are flexible and the systems are inflexible. Applications are developed in a best-effort approach to trade requirements compliance, while respecting the restrictions and limits imposed by the rigid infrastructure. The result is that things like QoS and resilience are rather difficult and cumbersome to achieve.

The proposed alternative on the other hand advocates a sustainable degree of flexibility on both sides (application-system), whereby the system itself makes also a best effort to adapt to the application and user expectations for a given operational environment. In such a philosophy, the enforcement of strict rules (i.e. reluctance to adapt) towards the servicees (application), give place to a more "*willing to optimize*" attitude from on the system's point of view.

The most fundamental component for moving towards this goal is a holistic capacity for change. The decision of what suggests an optimal change towards adaptation as well as the assessment of its outcome (with regard to performance, redundancy, etc) is left for the moment out of the scope of this work and it is seen as an algorithmic problem. The achievement of the ultimate possible flexibility and ability to carry out a change of

functionality on the other hand, is the exact goal of this work.

In the remainder of this section we introduce a framework that enables dynamic composition of functionality to leverage autonomic behavior. Before that however we give some background information on the goals and principles of the ANA project [1, 2] which motivate the herein presented work. We introduce the ANA system design approach that precipitates some of the design decisions made, after introducing the design of the functional composition framework we proceed to an early evaluation. We finish by explaining how the scope of this work differs from previous work on active networks and outline the future directions.

## 4.2 Background, Motivation

ANA [1] aims to design a *clean-slate* architecture for enabling autonomic system behavior that targets customization of a network to the user and application requirements and operational environment conditions [3].

The principle standpoint on ANA is a bottom-up approach where architecture-determining criteria such as address handling and network formation that were static beforehand, shall be replaced by an autonomic management system that actively composes and customises the network provided functionality. The ultimate goal is the design of a dynamic and integrated network architecture where the network structure is not dictated in protocol standards, but handled by the network system itself.

In ANA one of the fundamental primitives is that of an operational context (called *compartment*) within which a set of basic functionality providing elements (called *functional blocks*) cooperate to provide a service. Functional blocks may be co-located on the same *ANA node* or distributed across many. An ANA node provides simply a broker entity for functional blocks. An ANA node may map to a physical host but does not have to. It may also spawn across physical nodes or more than one ANA nodes may co-exist on a physical host. Adaptation is enabled through the coupling process among functional blocks, and indirection (through *information dispatch points* – i.e. label identified attachment points between functional blocks) is a crucial primitive for enabling this transparently and dynamically.

A compartment dictates how functional blocks cooperate to enable a data, control, or management plane functionality or service. In the ANA world multiple compartments may be interfaced in a layered, peering or orthogonal way to combine operational contexts and promote a virtualization of an ANA network over the physical infrastructure (where the legacy internet is only one of them).

The scientific objective of ANA is to challenge and prove the feasibility of this experiment (and not the performance at the moment). The aim is a network model that scales in time and in a functional way; that is, the network can extend both horizontally (more functionality) as well as vertically (different ways of integrating abundant functionality) and change over time. The main premise of this work is that a functionally scaling network is the basis for an evolving network which includes the various self-* attributes such as self-management, self-optimization, self-monitoring, self-repair, and self-protection.

In this "sea of functional blocks" the here in presented composition framework opts to provide the basic mechanisms for enabling their cooperation. It should be flexible enough to accommodate existing and even future type of cooperation schemes and not act as an impediment for future network developments.

## 4.2.2 Project-specific objectives

Figure 2 shows a block diagram of an ANA node. At the heart of and ANA node resides the *MINMEX*. It is a simple message passing microkernel that provides a message switching broker service to functional blocks. It may reside in the kernel space or the user space of the hosting platform.

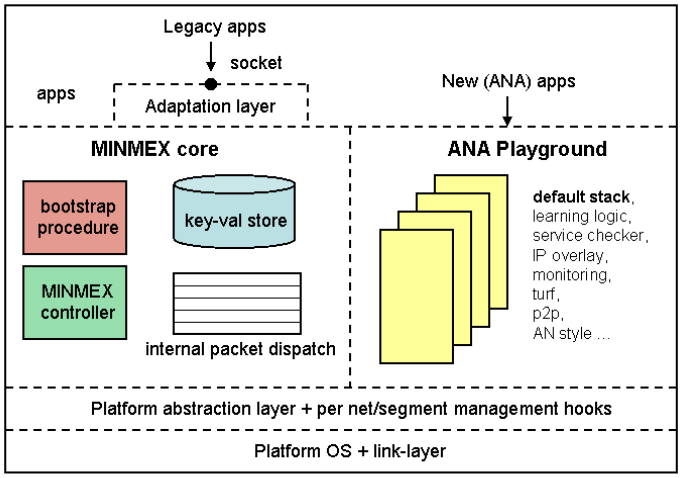



**Figure 2. ANA Node block diagram**

Any other functionality is provided by means of functional blocks which reside in the ANA *playground*. The playground is like an execution environment vertical to the kernel user space bordering, which means that functional blocks may run in user space and or kernel space, exchanging messages through the MINMEX.

All that is exposed to the MINMEX from the playground is a set of *information dispatch points* (IDP). As we mentioned earlier an IDP is the endpoints of message communication through which functional blocks exchange messages. An IDP of a functional block may be dynamically discovered through a well-known resolution process (and that is the only well-known thing in ANA).

In the most simple form of communication (which is also the paradigm followed in today's legacy network subsystems), any functional block already knows the IDP of any other functional block it wants to communicate with and is able to exchange directly messages with it. In this paradigm a function chain, i.e. the sequence of all subsequent functional blocks that will process a message, is rather explicit. In this model indirection is possible by explicitly "hiding" a functional block behind the already known IDP.

One objective of the functional composition framework is to make this process more implicit and often transparent to the sender functional block, in other words more

dynamic. The benefits advocated are twofold. First, in this way a functional block will not need to have a-priori knowledge of a subsequent functional block in a function chain (possibly –but not necessarily- only of the functionality expected next). Note that this also allows a subsequent functional block in the function chain, to be remotely residing without having advertised its IDP locally. Second, decisions of the next receiving functional block can be more dynamic, and optionally based on run-time conditions and decisions which are beyond the semantics of the message sender (external conditions, system state, etc). At first glance in such a paradigm, opacity between functional blocks seems to dominate. However, in reality it is more transparency rather than opacity, since the sender functional block may express (through some predicate information) its wishes/expectations/requirements for the message delivery, but does not have explicit control of whom the message is delivered to.

The dynamicity achieved through this first objective, opens the way for new routing potentials. By enabling more dynamic decisions, it is possible that the implementation of message routing in a hosted function chain takes into account different semantics without the introduction of *hacks* (cross-layering):

- message state (as in legacy systems)
- network occurring events (e.g. congestion, topology changes)
- environment occurring events (e.g. radio noise levels)
- node local state (e.g. power level, cpu load)
- flow information
- administrative policies
- application profiling information
- … etc

This immediately allows more space for autonomic behavior as decisions can be based on multiple inputs and are external to the message path. They may be made at a more abstract (higher) level within the node or within a compartment.

The second objective of the composition framework is to leverage virtualisation. As the main virtualization primitive in ANA is founded on the compartment to provide a collaborative context for functional blocks, the functional composition framework needs to be able to support these collaborative contexts by accommodating multiple independent function chains. In this aspect a functional block may appear in many function chains. A requirement for this is, of course, that the operation of the functional block internally should be as atomic as possible, and their sole dependencies in terms of other functional blocks to be expressed in an abstract way. Two independent function-chains may as well interface/cascade or be otherwise coupled to provide the virtual combination of contexts advocated in the previous section. To imagine this in terms of an example from the legacy internet, a function chain servicing an IPv6 routing compartment may be using the services of an IPSec servicing function-chain, and therefore the two of them may be coupled for the needs of security IPv6 extension headers.

A final requirement which is satisfied by the composition framework is for loose enforcement. As the existence of the composition framework is not internal to the MINMEX message exchange, but rather an ANA playground component (a functional block itself), if for some reason a set of functional blocks wish to communicate in a more

explicit way (as in TCP/IP today), then they are free to not use the composition framework service.

## 4.3 General Design

We now introduce the general design of the ANA functional composition framework and detail how this design achieves the aforementioned goals. There are two main parts encapsulating the functionality entailed in the composition framework: the *runtime engine* and the *composition manager*.

### *4.3.2 Run-Time Engine*

The runtime engine (Figure 3) is responsible for the main operation of the functional composition framework. It integrates all the data structures and functions needed for processing incoming messages determining their recipient functional block and dispatching them. Incoming messages are en-queued scheduled for classification (decision of the next functional block in the processing chain) and dispatched. It is implemented as a functional block itself running in the ANA playground and providing services to other functional blocks that may want to use it. In this way it abides to the design principles of the ANA node.

There are two types of requests that the framework services:

- *Lookup* requests: An entity needs to know where it should send a message (or bulk of messages)
- *Routing* requests: An entity passes a message to the framework, the framework determines (by means of the classifier) the recipient of the message and propagates the message to the recipient.

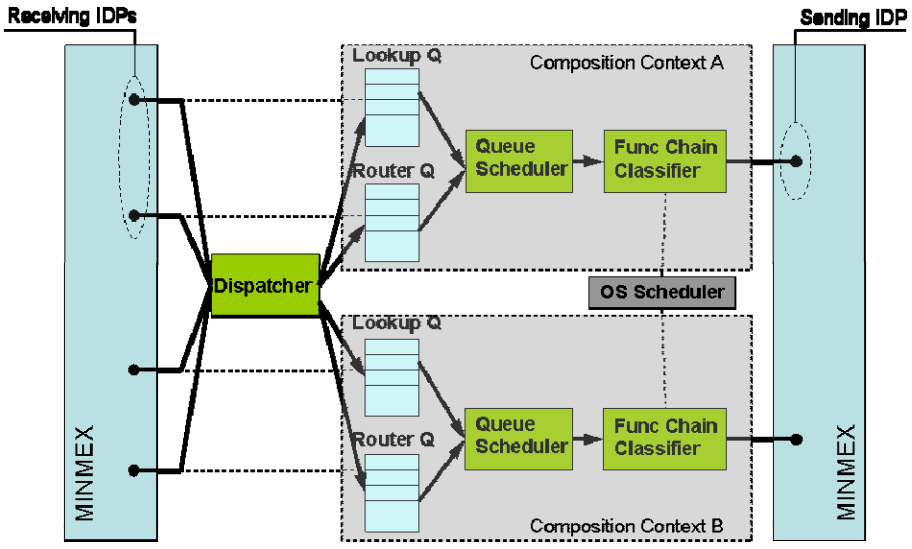



**Figure 3. Composition Framework Runtime Engine**

The *scheduler* is responsible for de-queuing requests from the two queues respectively and invoking the function chain classifier to determine the next recipient of the message. Maintaining a scheduler as opposed to a simple dispatcher allows us to implement different scheduling strategies and processing priorities for the two queues, so as to achieve the best possible performance.

A *function chain classifier* is at the heart of the composition framework and essentially represents a certain collaboration scheme among its participating functions, that can be static (in its simplest form as in legacy systems) or dynamically extensible encompassing whatever routing semantics one wants to accommodate in the routing process between functional blocks. Once a classification decision has been made and the next functional block in the processing chain has been determined, a response is formed and en-queued at the *send* queue. This response is either a response to a lookup request (routed back to the requested) or a propagation of a message directly to the recipient in the case of a route request.

One may be tempted to think that in a sense the composition framework duplicates the message passing functionality of the MINMEX in a slightly more sophisticated way. However such a parallelization would be more than an oversimplification. Basically the functionality MINMEX provides and that of the functional composition framework differ in semantics. MINMEX does switching among functional blocks, while the composition framework provides the means for routing decisions (!) in a more or less sophisticated way (depending on the context instantiated in the classification engine).

An important feature captured in Figure 3, is that the framework can accommodate multiple functional composition instances, which is aligned with the requirements for multiple collaborative functional contexts. Each of them owning its own queues, scheduling strategy, and classification engine is able to match the requirements, and capture the semantics and incentives of the service a compartment aims to provide. One may wonder why not run multiple instantiations of the functional composition functional block (each serving the purposes of a different compartment). Although this of course possible, however by enabling this feature it is possible to achieve more fine grained runtime control through the composition manager: It is possible to couple/cascade functional contexts, replace classification engines or scheduling strategies, etc by simple redirecting the queues of one composition instance to another or from one scheduler to another as a single step (of editing some data structured); and it all happens transparently to the servicees.

### 4.3.3 Composition Manager

The composition manager operates out of band to the runtime engine's functionality. It implements a set of control functions that give access to the runtime component configuration tasks at runtime or at the instantiation time of a composition context instance.

The functions it performs can be summarized in the following groups:

- *Instantiation of a new functional composition context*: These functions create the data structures (queues, scheduler, classification engine, etc) and start the servicing threads for a new functional composition context.

- *Access to a classifier engine for re-configuration tasks*: These functions enable access to the classifier API of a functional block composite so that new functions can be added, or existing ones can be reconfigured or removed, and new policies may be established.
- *Customisation/Re-configuration of a composition instance of the framework*: This set of functions allow the modification of various runtime re-configurable aspects of the framework's structures such as scheduling quantum, queuing strategy, queue sizes, queue redirection, classifier replacement, etc.
- *MINMEX interfacing operations*: These are function used to perform operations related to the interface with the MINMEX, such as IDP registration, key-val repository updates, etc.
- *Other housekeeping operations*: This set includes some house keeping operations for the internal integrity of the functional composition framework.

## 4.4 Integration with ANA

In this section we describe the interfacing and operation of the composition framework within the ANA node.

### *4.4.2 MINMEX Interfacing*

The runtime component registers with MINMEX two IDPs for each framework instantiation. One of them is for listening for message routing requests while the other is for servicing look-up requests. The dispatcher practically knows which queue to deliver a message to by observing which IDP is was received from.

At the sending end a thread practically for each message at the send queue looks up the recipient IDP set by the classifier and sends to it the message. There is currently one sending queue for all framework instances, although if need appears it would not be difficult to extend it to facilitate multiple sending queues.

The manager component exposes to the MINMEX one permanent IDP for listening to incoming (re-)configuration and framework management messages.

One additional special IDP is registered with the MINMEX for the manager module which facilitates exclusive communication between the MINMEX and the runtime component. Through this IDP is possible for the MINMEX to inform the manager module about changes or updates in the state of the running functional blocks. This is needed so that the manger module can safeguard the correct operation of the instantiated function chains.

The current prototype implementation of the composition framework has already been integrated with the ANA software.

### 4.4.3 Operation of Functional Composition Framework

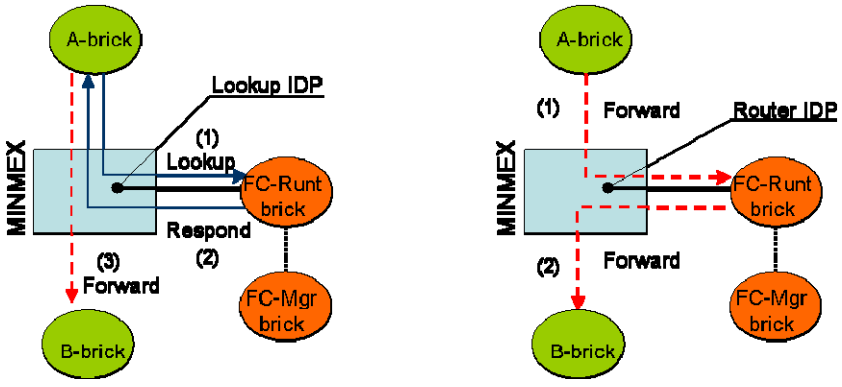
Let us revisit the dual functionality that the framework provides as its purpose probably needs a bit more explaining. In the most primitive datagram communication a sender sends an individual message to a recipient. All the sender needs to do is carry out its processing on the message payload, and dispatch the message to the next entity that will process it. This is a somewhat simple and agnostic message transmission. At best the sender may provide some predicate information that the framework can use to make a better decision in determining the recipient (in legacy stacks this would be the next header field in the datagram).

In a somewhat more complex (less agnostic) case the sender might need to establish a "connection" or "session" by carrying out some form of negotiation or exchange of state information with the recipient before exchanging messages.

Moreover the first case requires a per-message routing decision and dispatching throughout a message exchange (datagram switching), while the latter only takes one decision step (for the request) and then the whole message exchange may happen in batch between the involved functional blocks without further intervention from the functional composition framework (like circuit establishment). It is interesting to point out that a hybrid combination of the two is possible for separate hops across a processing chain. A more static version of it happens today in the legacy internet as datagrams are "circuit switched" inside the stack, and "packet switched" across in the network. In accordance with the aims of ANA, we generalize this process here by making it more dynamic and less hard-bounded.

Figure 4 illustrates these two types of servicing and operation of the functional composition framework within the ANA. Note that a pair (or set) of functional blocks may freely choose to avoid using the services of the framework altogether as it is not enforced in any way, as this lack of strict enforcement is the exact essence of ANA.

It should be obvious now that different incentives and performance requirements can be serviced by the composition framework through these two types of requests; and this has necessitated the two different queues in the block diagram above. When a request arrives the *dispatcher* examines its type and en-queues it to the respective queue.




**(a) lookup service** **(b) routing service**

**Figure 4. Composition framework operation.**

## 4.5 Basic Evaluation

### *4.5.2 Flexibility*

From an implementation point of view several decisions have been made to further assist the extensibility of the composition framework.

First of all the two queues in each framework instance maintain their own en/de-queuing primitives which are responsible for implementing different queuing strategies while maintaining the same interface towards the remaining of the frameworks building blocks. The *dispatcher* can then make use of different queuing strategies as instructed by the *composition manager*, while preserving the same API.

Similar approach has been followed for the implementation of the scheduler. There an additional configuration capability has been added so that the operating system's scheduling priorities can be set differently for each framework instance scheduler.

The selection (at instantiation time or during re-configuration at run-time) or scheduling algorithm, queuing strategy, and queue service slices, at each framework instance enable the amortization of QoS performance and overhead introduced by the framework in specific operational conditions.

Finally as has already been mentioned the implementation of a function chain classifier has an impact on the trade-off between flexibility and performance. Inevitably a more flexible and software implemented classifier will be expected to involve a high computational delay. However, an implementation that we consider porting in the near future from the literature [11] has been reported to achieve sustainable performance (near line speeds) at high flexibility, for edge networks.

### *4.5.3 Performance*

We wanted to understand if the overhead introduced by the framework is at least constant and insignificant and therefore we conducted a very early delay overhead measurement with a user space implementation. The results are shown in **Error! Reference source not found.**, by means of two delay distribution plots (time and pdf).

We used a dummy pass-through classifier as we did not want to encounter classification overhead but instead measure how the framework "behaves". For a number of runs we generated large numbers of messages (sized 100B - although this is of no important as they are copied by reference). The graph is indicative of the results and shows the propagation delay as messages traverse the framework, which remains on average constant in the range of 100-107μsec (for reference a 100Mbit net interface can generate one 1.5KB packet every 120μsec). Table 1 shows the mean and standard deviation values for 3 of the runs. In the first run the existence of some outlier values led to a rather overvalued standard deviation. However as the mean shows the average delay is consistent with the that of the other runs.

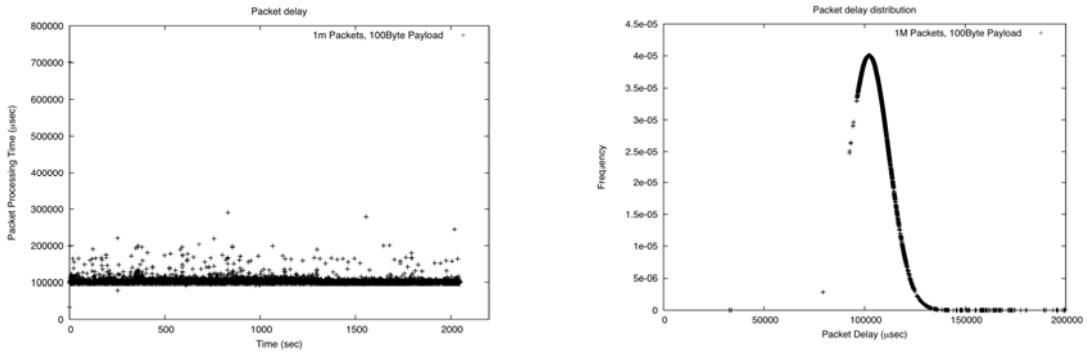



**Figure 5. Propagation delay distribution**

| 21845 messages / run | | |
|---|---|---|
| Run | Mean (µsec) | Std. Deviation (µsec) |
| 1 | 105.6 | 42.2 |
| 2 | 101.6 | 6.1 |
| 3 | 102.5 | 9.9 |

**Table 1. Mean and std deviation for 3 runs**

## 4.6 Related Work

The idea of developing extensible network subsystems, building composable protocols, and on-demand selecting data paths within a system or across a network is not new. Early research on the subject is as old as the Internet as this is the fundamental principle of routing and therefore already captured in most typical packet classifiers [4, 5, 6] and many of the ideas and the developments already appear in systems. Setting the border line between a rigid, inflexible system, which enforces a certain protocol function set, and a flexible one that allows dynamic selection of processing paths, is hard to define, as it depends on the viewer‘s standpoint. For example in today’s (considered fixed) TCP/IP network stack an application still may (in most cases) select to use a certain transport protocol (UDP over TCP), or not at all (raw sockets), IPSec or not, and implicitly select among multiple data processing paths (which is the essence of a classifier after all).

In the past many systems have been developed that enable a certain (often fine grained) degree of extensibility of the classification engine and the resulting (set of possible) processing path(s). Some of the most prominent ones include [7, 8, 9, 10]. However what we are targeting in this project is somewhat different than this, as we opt for run-time re-configurability and extensibility instead of compile time.

Relatively more recent work on active and programmable networks has had similar but run-time objectives, namely on facilitating adaptive and extensible functionality either in-band or out-of-band and in this respect our work here has borrowed and reused some of the ideas in the existing literature [11, 12, 13, 14]. However once again our scope is a bit

broader as we want not only to add/extend data path functionality, but also add context and semantics to this process it. An application does not need to explicitly instruct the deployment of one function of the other, and especially functions which are semantically unrelated to its operation, it is the system that we expect to adapt and customize. For example an application may need to express its QoS requirements as real-time, the system then given measurable metrics such as delay might choose to deploy a trans-coding function increase the transmission rate, etc.

Moreover only few active network solutions have considered the magnitude of the service composition model for real-life network environments and hence provide only a limited flexibility for the composition of network services. A common objective of most active network approaches was to expedite network evolution through solutions that enable extensibility of network functionality by way of dynamically loaded code. And most active network approaches, [13, 15, 16, 17, 18] accomplish this through software plug-ins or a similar form of active code integration. However a study [19] has revealed that even most extensible active router platforms lack sufficient flexibility in order to allow for true evolution. Such modular or plug-in based architectures typically limit the scope of future changes through pre-defined interfaces. Instead, true extensibility should not be limited to a fixed set of modules or plug-ins, but should rather allow modification and replacement of all components contributing to a service composite.

Having argued the need for tailoring a system to different environments, virtualisation as a context-related capability becomes of a fundamental significance. ANA inherently permits and promotes this principle at different scales through the compartment primitive, which imposes the requirement on the composition framework the simultaneous, independent (and often orthogonal) coexistence of multiple processing function-chains.

## 4.7 Conclusion & Future Work

In this paper we have presented the design of a framework for managing the creation of functional service composites in ANA to leverage autonomic operation, detailing the aspects that render it a flexible and extensible solution. We have briefly evaluated a prototype user space implementation to assess its overhead. The results show that the processing latency imposed by it is relatively insignificant and constant.

In this framework a *classifier* implements a function composite which captures the routing and collaboration semantics among processing elements. For this reason the continuation of this work will be in developing a number of classifiers able to provide processing paths based on different context incentives. A starting point will be a classifier engine from an active router architecture in the literature [11], which was shown to be flexible and extensible enough to facilitate new functions or compose evolvable protocols.

# 5 INFORMATION SENSING AND SHARING ARCHITECTURE

## 5.1 Overview - Current Architectural Limitations

As already highlighted in the previous sections the current fixed structure of network subsystems (such as the TCP/IP-based architecture of the Internet), although they have served very well for the last twenty years, they can no longer serve the needs of emerging networks in the face of adaptation to application requirements and network convergence. The proof for this insufficiency in the current internet architecture can be witnessed in the proliferation of NATs, firewalls and proxies, and other "middleboxes", specifically engineered to patch the network in a rather inelegant way, in an attempt to serve different classes of applications, or bridge the gap between different networks (e.g. the Internet with the mobile telephony network).

In ANA and other autonomy-advocating architectures, we have advocated and exemplified and need for capacity to adapt, both and equally to application needs and the operational environment. This, being the fundamental requirement for enabling autonomic behaviour, led us in the previous section to engineer a solution for embodying this capability in ANA. In this section we go one step further in our exploration and we try to tackle the issue of how to trigger and guide adaptation.

As a starting point we anticipate that information collected from intra-protocol state and from the wider network context, can be used to determine the choice of protocol functions, mechanisms and parameters for optimising performance. Our approach is to fuse ideas of cross layering and of active and passive measurement into a generic framework, so as to provide universal views of a network context. At different levels of abstraction these can be used to help choose protocol functions, and dynamically compose them into a communication system for the particular application, stream and operational context.

## 5.2 Background - Motivation

A key research area currently investigating the effects and performance benefits of sharing state information across a network subsystem is cross-layering, which has been shown to be more appropriate in newer, non-traditional environments, e.g. [1]. Cross layering breaks down layer boundaries, sharing information about network and application state, which allows performance to be optimised.

However cross layering has to date been used in a problem and/or network specific manner, focusing on one problem at a time and independently or what is happening in the global state of the system (holistic view). It was explained in deliverable D2.1 and

exemplified with cases from the literature that to ensure stability of operation and ongoing interoperability in the wider network environment, a generic and universal, engineered approach is required, which will provide the means to safeguard against possible side-effects of cross-layer optimisations.

At the same time there is need for a solution that will enable better feedback from across the network, in terms of currently available bandwidth, delay, jitter and other metrics, as well as the physical environment possibly, providing richer context for adaptivity decisions such as for example codec choice and content placement.

The objective of an information sensing and sharing system is to drive adaptation and optimisation of what lies between the user (application) and the network environment (i.e. the network-subsystem, be that a layered one, a monolithic one, or component-based one), so as to improve performance and stability. In that respect a solution for sensing and sharing information must support and provide input for run-time functional composition and (re-)configuration.

In this work we have taken the approach of event-based systems due to the fact that they can provide low-cost time-sensitive knowledge-driven (interrupt-based) service and a re-active paradigm to changes happening in the monitored environment. Another reason for this choice is the decoupling of system components, permitting them to operate independently until there is need to collaborate (event occurrence). This carries the potential for easy integration of heterogeneous and independently designed components in complex systems that are easy to evolve and scale. At the same time it overcomes the limitation of an *a-priori* decision for a centralized versus a distributed design, as they can be used interchangeably for either or a hybrid combination of the two by inherently promoting indirection. In view of these arguments, they are fundamentally superior to request-reply architectures.

In the literature there are numerous designs with applications in various time-critical applications in data bases, fault detection, environmental sensing, network management, etc. Some of the most prominent research prototypes include [**Error! Reference source not found.**, **Error! Reference source not found.**, **Error! Reference source not found.**, 5, 6]. Note that in this research we regard publish subscribe systems as event-systems as well.

In the remaining of this section we describe the design and explain the capabilities of the information sensing and sharing architecture. It differs from other initiatives in the literature (although integrates many of the interest principles of interest seen in them), and it is engineered to meet the requirements of ANA (explored further in subsequent sections).

## 5.3 General Design of an Information Sensing and Sharing Architecture

In this section we focus on the design and implementation of the information sensing and sharing framework (hereafter called the ISS framework). Figure 6 presents an abstract view of the building blocks that comprise the architecture (the illustrated

scenario/example is discussed in [18]). A closer examination of the main building blocks (those semantically related to its functionality) follows, along with a description of the basic operation.

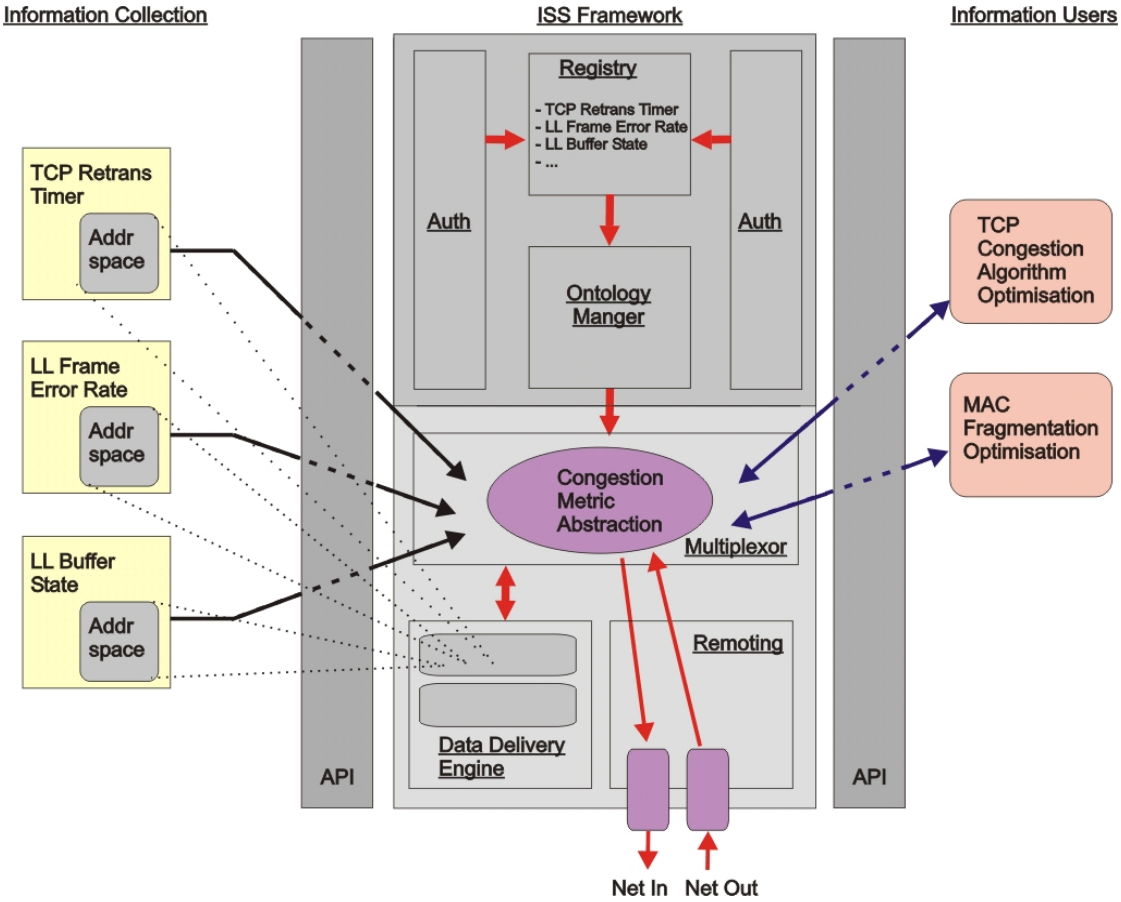



**Figure 6: ISS Framework.**

There are two sets of components in the ISS framework. The first set of components (on-line components), provide the runtime functionality of the ISS framework, namely system (node/network) awareness. They are responsible for collecting and disseminating information in an event driven fashion. These components are the *multiplexor*, the *remoting*, and the *data delivery manager*. The other set of components (off-line components) are related to management tasks within the framework or between the framework and its client entities. These components include the *authenticator*, the *registry* and the *ontology manager*. The key functionality is provided by the *multiplexor*, the *ontology manager*, the *data delivery engine*, and the *remote notification* components, which we describe next.

1. *The multiplexor* is the heart of the ISS framework as it is responsible for multiplexing events triggered by one or more event sources and demultiplexes them to one or more interested client entities. Any entity sharing information of any type acts as an event source, and any entity interested on information acts as an event sink. Any ISS client may act as either an event source (providing information), or event sink (consuming information) or both. The event multiplexing functionality is founded on Boolean algebra to describe the

processing that takes place and to guarantee that at from any canonical state, the system will progress to another also canonical state (closure property) and therefore retain stability and predictability. Boolean logic operators combining event sources, are therefore both extensible and easily combined to generate multiplexing capabilities (satisfying the requirements specification of a sink). Figure 6 exemplifies three event sources or monitors on the left, operating at different protocol layers. Two clients (or event sinks) on the right side have registered higher level abstractions of interest that specify how the event sources are to be multiplexed.

2. *The data delivery manager* is responsible for the sharing of the data and delivery to interested parties (event sinks) when a (combination) of event(s) is triggered. An analysis of the recent literature in cross layering [7] regarding different types of information that is typically shared across the network or across layers, leads us to classify information sources into three main categories: those that serve simple notifications (binary), those that provide single value information (scalar), and those that provide a larger volume of (spatially or temporally) collected data. In the first case, if an information consumer is only interested in the occurrence of an event then the multiplexor provides the complete functionality. However, in the latter two cases, where a larger volume of information is shared, the data delivery component will decide on an scheme for delivering the information as well as how the information may be arranged depending on semantic heuristics such timeliness and volume. For instance aggregation operations may be instructed and copy of the data to a location appointed by the information consumer or combine the information in dynamically generated data structures as prescribed by the information consumer and buffered in a queue.

3. *The remote notification facility*. is used to extend the ISS functionality beyond the node's scope across the network, in order to enable network driven context awareness. It augments the functionality of the mutliplexor and the data delivery component across the network in a uniform way, thus promoting a universal view of the information collection and dissemination process towards the clients of the ISS framework. As it simply extends the corresponding APIs, it is not bound to any specific network transport and may deploy any available transport protocol. A significant limitation of course is the existence of delays or errors during the propagation of information across the network and this has to be taken into account in the construction of a multiplex. However, in cases where this is an inevitable condition the ability to localize processing and aggregate information/events before transmitting them over the network is actually a benefit. The main additional flexibility offered through the ISS framework is the ability to combine separate or different event sources across the network. These issues are still an open issue in the proposed design.

4. *The ontology manager*. One of the main feasibility challenges in the ISS framework is the task of understanding the client/sink requirements and translating them to a multiplexing of event/information sources. The abstraction that ISS provides between information collection and information use relies on this capability. This requires some formal means for expression of requirements

(from the information users), and an inference process for associating them correctly with the capabilities and services provided by the entities that generate the events and provide the information. A domain ontology backed by a knowledge base of user provided "experience" is deployed to leverage this process. The knowledge base stores information of how combinations of events associate with high level abstractions that the information consumers use to express their requirements. Some simple examples are shown below.

A summary of the operation of the ISS framework is as follows. Entities, which are able to collect and share information, register with ISS as information providers, while entities that want to acquire information register as information consumers. Information providers are essentially event sources for the ISS framework, while information consumers are event sinks. Any single client module of the ISS framework may register as an information provider, consumer or both. During the registration process the client is first authenticated with the framework and acquires an ID token (hash key) which presents to the framework thereafter in all transactions. If the client is an information consumer module the ontology manager parses its requirements specification, consults its knowledge base for translating to the appropriate bindings of event sources, and generates the event multiplex description, which is passed to the multiplexor for instantiation. The multiplexor then instantiates the 'wiring' and registers the events with the event sources. Thereafter, operation begins, and whenever the appropriate combination of events occur the collected information is delivered to the event sink, in the form determined by the data delivery component. This resembles the illustration of Figure 7.

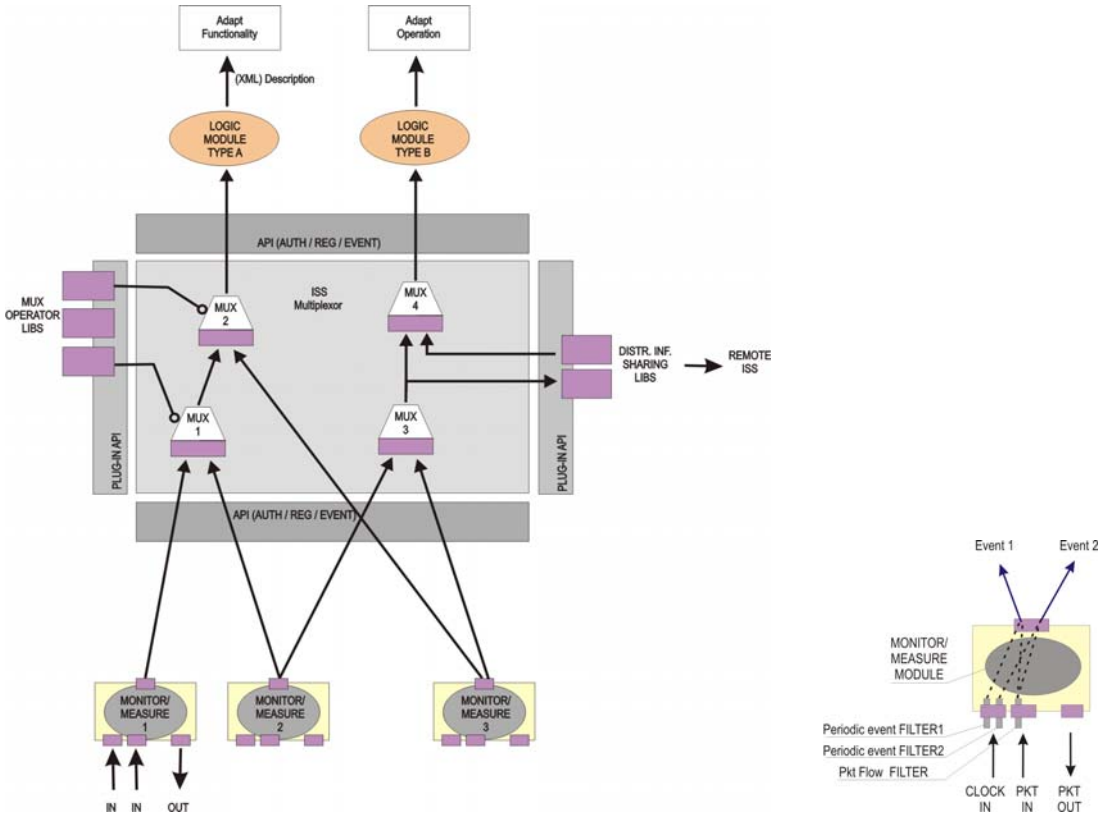



**Figure 7 (a) and (b): ISS Operation**

In Figure 7.a, *mux1* combines events from *monitor1* and *monitor2* and propagates an event to *mux2* which in turn combines with an event from *monitor3* and upon trigger fires a notification to *moduleA* (which presumably computes some logic). Similarly, *mux3* combines events from sources *monitor2* and *monitor3* and propagates an event to a *mux* at a remote ISS instance in the network as well as to *mux4*. Finally *mux4* combines the output from *mux3* and a remote event source from the network, and upon trigger sends a notification to logic *module2*. The plug-in interface on the left enables simple processing capabilities for the multiplexing elements so as to allow them to perform "aggregation" functions (used for internal processing of the combined input signals). Example of such operations might be simple data aggregation, averaging of values, min-max operations, etc.

Figure 7.b, at the right corner, illustrates the event API of an information provider modules by which they are able to deploy input filter corresponding to the events to be fired. For example when filter mask 1 and 3 are matched event 2 is fired, while when filter mask 2 and 3 are matched event 1 is fired.

One concern regarding the ISS framework is obviously the performance overhead that is introduced. This concern has led to a number of implementation strategies that aim to minimize overhead. First, the service provided by the ISS framework does not rely on a running server process but rather is implemented as a set of dynamically loaded shared libraries that maintain a shared memory allocation for the common parts of all user processes. A second improvement is that once the initial event multiplex has been produced for an information consumer, a number of algebraic optimizations may take place to reduce the operations required. These optimizations are quite straightforward most of the times since the construction of a multiplex is founded on an extensible algebra on event automata, which also complies to generic optimisation techniques in the literature (Mealy/Moore machines). The expected effect in practice is the reduction of the number of computations, the number of intermediate processing steps in the propagation of the event notifications, and the amount of memory allocations required for the data structures in the multiplex. Note that most test cases that we have considered, after simple optimizations the number of steps is reduced to 2.

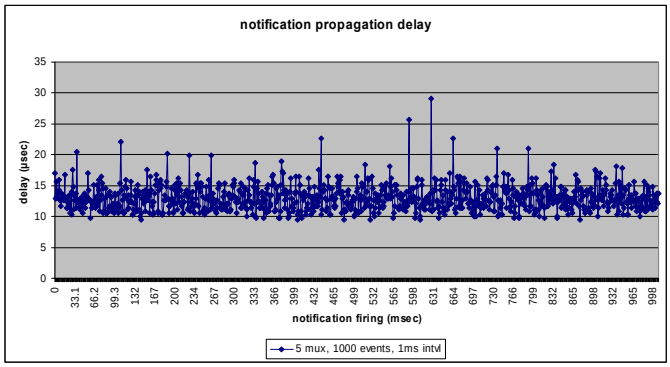
## 5.4 Targeting Performance

As it was briefly pointed out earlier timeliness of event delivery is critical in an event based framework that is tailored to network systems operation, as very often network or system occurring events have a short-lived temporal validity. At the same time the analysis of various metrics from the literature (presented in the respective section of deliverable D2.2), has shown that several of the measurable quantities that can potentially trigger events, often occur at very high rates (congestion window, flow state, packet arrivals, frame errors). Therefore it is important for the ISS framework to be able to follow the pace that events sources are triggered.

For this reason we were driven to characterize the performance of ISS in terms of two measurable metrics, notification propagation delay and response time of the system to the event occurrences.

As the current prototype is a user space software implementation, it is easily understood that there is plenty of space of performance improvement from an equivalent kernel space or in hardware implementation (FPGAs). The initial measurements we have carried out however are quite encouraging.

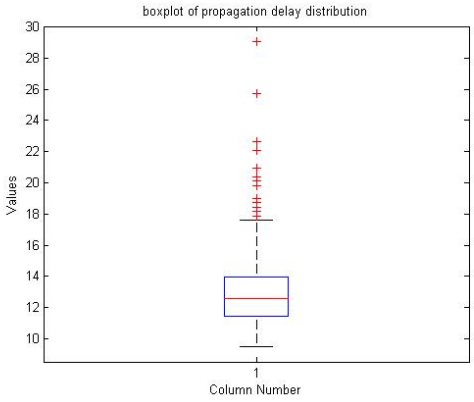
Figure 8 shows an analysis of the propagation delay for a single event that triggers 1000 notifications to a consumer in a multiplex that has a number of 5 intermediate multiplexing structures (*mux* elements) between the event producer and the event consumer (we consider this to be a representative number of intermediate data structures for the majority of cases we have encountered in the cross-layering literature).



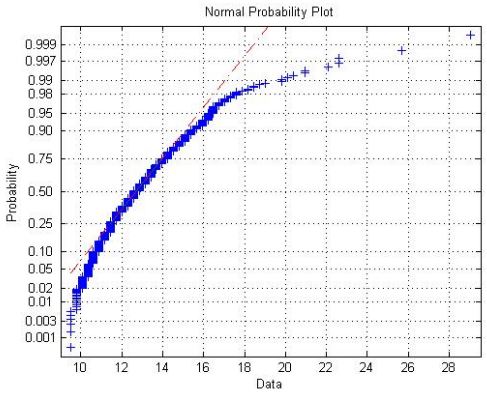

**(a) propagation delay**

| Percentiles | Delay (µsec) |
|---|---|
| 0 % | 9.4984 |
| 25 % | 11.4540 |
| 50 % | 12.5714 |
| 75 % | 13.9683 |
| 100 % | 29.0540 |

**(b) delay distribution percentiles**




**(c) Box-plot of propagation delay**




**(d) empirical versus normal delay distribution**

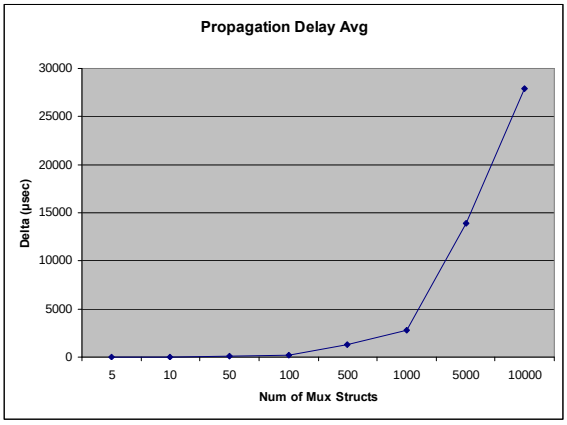
**Figure 8**

Figure 8.b (table) and Figure 8.c (box-plot) show the important percentiles of the distribution of the delay values with the 25% and 75% being in the range of 11-12 µsec (note that the granularity of the benchmarking tool was at 290nsec). Finally Figure 8.d shows the delay value distribution against a normal distribution which shows a small skew at the highest delay values (10% of them), which is encountered at the first few data samples and is most probably due to the adjustment of the benchmarking tool.

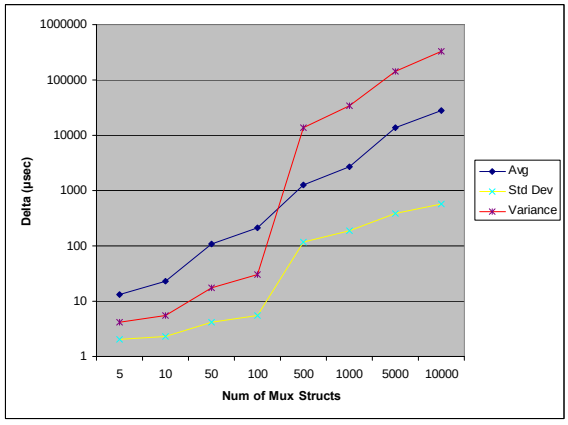
Figure 9, further examines the issue of propagation delay increase as the size of the multiplex increases. While we have stressed the test to unrealistically large sizes of multiplexes ($10^4$) the aim is to discover the saturation points of our implementation. It is appreciated that in no case a multiplex will contain more than a couple of 10ths of *mux* data structures in depth (all examples we have considered from the literature need less

than 10). Figure 9.a shows that the propagation delay does not exceed on average the 1ms limit before a size of 500 mux structures and it remains below 100µsec until a multiplex size of 50 mux structures. Figure 9.b shows more clearly in log scale how the variance and standard error follow the propagation delay increase as the multiplex size increases. We observe a step in the variance and standard deviation at the 500 mux sized multiplex (in depth). This is due to the fact that events were fired at 1 msec intervals in all cases and therefore since for the 500 *mux* sized multiplex the propagation delay exceeds the 1 msec, some of the events have to wait longer before the previous ones are delivered. This also suggests that events (encountering larger delays) are queued but not lost. Finally Figure 9.c shows the dispersion of our 8 averaged data sets about their means (25% and 75% quantiles).

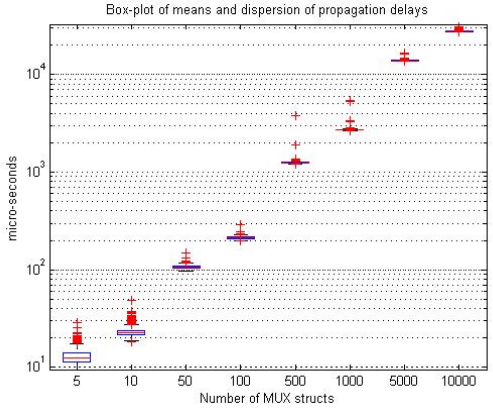
Regarding the response time it is very encouraging that up until a multiplex size of 100 *mux* structures in depth, our implementation can respond by triggering notifications in real time to network packet arrival times (120µsec is the nominal minimum packet inter-arrival delay for a 100Mbit network card).




**(a) Propagation delay increase with multiplex size**




**(b) Delay avg, variance and std error as multiplex size increases**




**(c) Dispersion (25-75 percentiles) of propagation delay for various multiplex sizes**

**Figure 9.**

To further measure the response time we arranged that notifications are fired at small intervals (50µsec 100µsec and 150µsec – 120µsec matches the minimum at line speed of

a 100Mbit network interface) from multiple simulated event sources. In Figure 10 we show how the propagation delay varies as a function of the number of event sources (mean/standard deviation). In all cases the experiment showed that our implementation is quite robust and scales really well as the "breadth" (number of sources) increases in a multiplex.

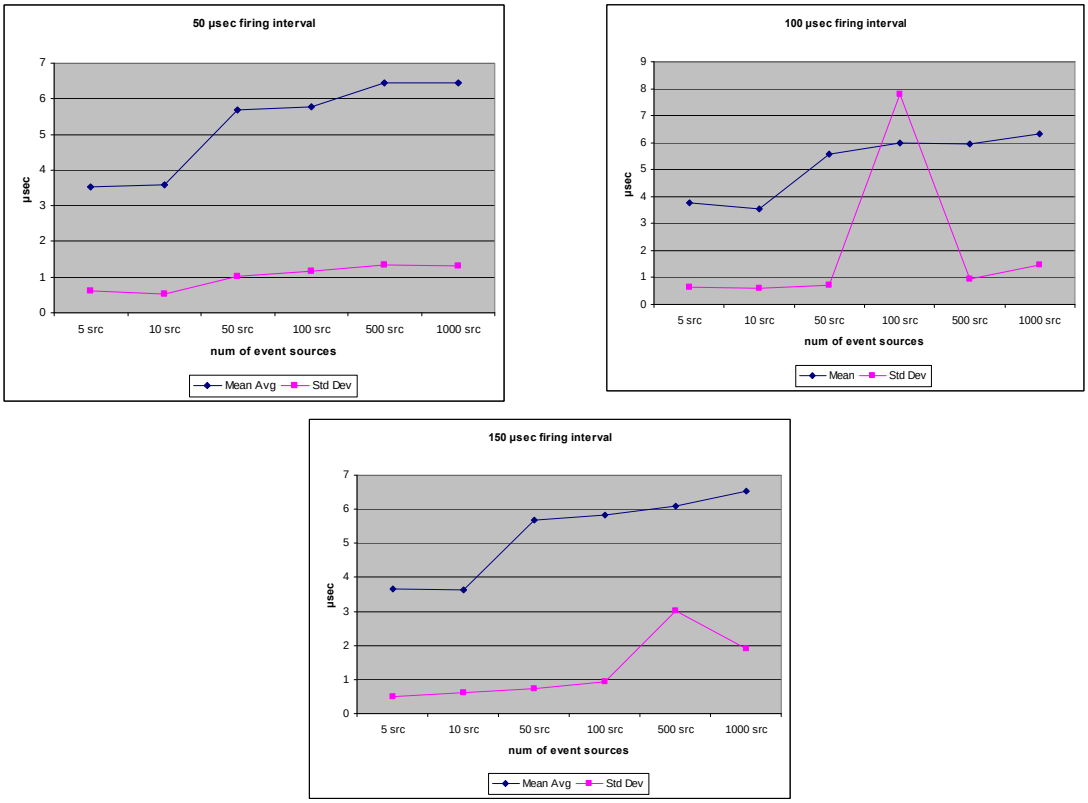



**Figure 10. Response times for multiple event sources**

Finally, despite that the most important quantification of performance so far has been in terms of temporal aspects (i.e. propagation delay and response time), we also considered important to examine the memory overhead imposed on the host system. This is shown in Figure 11 where we show how memory needs for the multiplex construction increase with the number of event sources. Once again the results are quite encouraging and justify our implementation decisions.

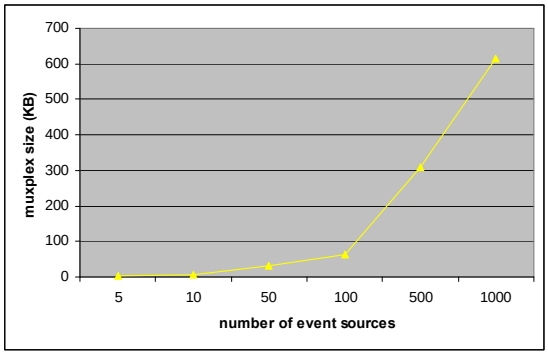



**Figure 11. Memory overhead**

## 5.5 Attaining Correctness

Performance as evaluated in the previous section is one fundamental requirement, correctness is the second one. Attaining correct operation in event systems is an interesting engineering challenge mainly due to the asynchronous nature. Furthermore what is expected to be the correct operation in one event-based system might not be the case for another. It is a case specific goal.

For this reason we tend to model the system as a black box, then, examine and specify its expected behavior, and finally proceed to engineer the system to implement this behavior. It is often pertinent in the literature, in order assist capturing and understanding the requirements, the use of a formal specification language to describe the externally perceived behavior of the system. We use a similar tactic here to first specify the external behavior of the ISS framework with regard to the requirements that ANA imposes, and then we proceed to explained how the internal engineering of the framework complies with this specification.

Most formal models (and as a result the one we consider here), define a system as a state machine, moving from one state to the next through actions. The black box view entails defining the correct behavior of the system at its interface. The system's evolution can be specified as a sequence of states describing a system's behavior, and it is called a *trace.* Trace semantics can be used to describe the behavior of a single process system as well as concurrent and distributed systems. At first this may sound counterintuitive as well as unrealistic, as it is not possible to enforce total ordering in distributed or concurrent systems. Yet, although it is possible to provide unrealistic and impossible specifications for distributed systems if they are defined in terms of total or global time, however if relative time and ordering is defined only for causally related states, it is possible to define implementable specifications, such as the ones we examine here.

Some definitions for our formal expressions in accordance with the literature [8] are shown in Table 2

| | |
|---|---|
| *Trace* | *A sequence of states* $\sigma = \sigma_0, \sigma_1, \sigma_2, \ldots$ |
| *Subtrace* | *Let* $\sigma = \sigma_0, \sigma_1, \sigma_2, \ldots$ *be a trace. Then if for* $i \geq 0$ *the subtrace* $\sigma_{\mid i}$ *is the trace* $\sigma = \sigma_i, \sigma_{i+1}, \sigma_{i+2}, \ldots$ |
| *Specification* | *A specification* $\Sigma$ *is a set of traces. A system satisfies a specification* $\Sigma$ *if it only exhibits traces which are in* $\Sigma$*. A specification is given as a set of combined predicates on traces.* |
| *Atomic Predicate* | *The atomic predicate P is true for every trace whose first state satisfies P* |

| | |
|---|---|
| Logical operators for predicates | $\varpi$ (OR),<br>$\omega$ (AND),<br>$\Psi$ (if-then),<br>$\neg$ (true if *atomic P* is not satisfied) |

| | |
|---|---|
| Quantifiers | □ (for each)<br>□ (for every) |

| | |
|---|---|
| Temporal operators | ± (next)<br>□ (eventually)<br>~ (always) |
| *If Ψ is an arbitrary temporal formula and* $\sigma = \sigma_0, \sigma_1, \sigma_2, \ldots$ *is an arbitrary trace* | |
| □ *Ψ is true for trace σ if there exists an* $i \geq 0$ *such that Ψ is true for the trace* $\sigma_{\mid i}$ | |
| ~ *Ψ is true for trace σ if for all* $i \geq 0$ *Ψ is true for the trace* $\sigma_{\mid i}$ | |
| ± *Ψ is true for trace σ if Ψ is true for the trace* $\sigma_{\mid 1}$ | |

**Table 2. Formal definitions and operators**

Using the formal expressions defined so far, we try in a step-wise fashion to introduce and explain the behavioral requirements for the ISS framework.

A basic event system consists of client components acting as *producers* or *consumers* of notifications (occasionally both as well), and an *event notification service* (that would be the ISS framework in our case). The producers and consumers interact with the event notification service via its interface that offers a set of input/output operations (Table 3). The operations *subscribe*, *unsubscribe*, and *publish* are input operations, whereas *notify* is an output operation.

| | |
|---|---|
| *Subscribe (X, F)* | *Consumer X subscribes to event notifications specified in filter F* |
| *Unsubscribe (X, F)* | *Consumer X unsubscribes from event notifications specified in filter F* |
| *Publish (Y, n)* | *Producer Y publishes event notification n* |
| *Notify (X, n)* | *Consumer X is notified about event n* |

**Table 3. Event notification service interface operations**

To explain a bit more these operations, let us introduce their semantics and their operation domains:

Let *C* the domain of all components in the system

Let $N \phi C$ the set of all notifications delivered to a consumer via the *notify* interface operation.

Let $F \phi C$ the set of all filters that is notification subscriptions a consumer is interested in. Subscriptions or Un-subscription to notifications are carried out through *subscribe* and *unsubscribe* interface operations. Note that in ISS these two operations happen implicitly at the registration to the ISS service and based on these the multiplex is constructed. This as will be discussed later on offers the advantage that the multiplex as a notification

service can be subjected to optimisation to the consumer and system's requirements, offering a significant flexibility advantage over typical publish subscribe systems.

Several implicit assumptions are expected to be enforced such as

- notifications are unique (i.e. each notification $n \ 0 \ N$ can be published at most once)
- every filter is uniquely identified to enable distinguishing between subscriptions

At the same time the state of the event notification service is characterised by three state variables, whose value is changed through the interface operations as shown in Table 4:

- $S_x$ , the set of active subscriptions
- $P_x$ , the set of published notifications
- $D_x$ , the set of delivered notifications

| | |
|---|---|
| *Subscribe (X, F)* | $S'_x = S_x \chi \{F\}$ |
| *Unsubscribe (X, F)* | $S'_x = S_x \setminus \{F\}$ |
| *Publish (Y, n)* | $P'_y = P_y \chi \{n\}$ |
| *Notify (X, n)* | $D'_x = D_x \chi \{n\}$ |

**Table 4. Updates of the state variables**

The behaviour of the event system is defined by its reaction (*notify* operation) to the actions of its external components: (a) producers (*publish* operation), and (b) consumers (*subscribe/unsubscribe* operations). Therefore the initial requirements for the basic event system are identified in terms of the correct operation of its interface [3, 9], and they can be informally stated as follows:

i) a component receives only notifications it has currently subscribed to (no false notifications)

ii) a component receives notifications that have been previously published (no lost notifications)

iii) a component receives each notification at most once (no false positives)

iv) a component receives all future notifications matching one of its active subscriptions

The first three requirements define a safety property [10] and can be formalised in the following postulate

*A simple event system is a system that exhibits only traces satisfying the following:*

$$\sim [\textit{notify (Y, n)} \ \Psi \ [n \ 0 \ N(S_Y)] \ \varpi \ [n \ 0 \ \chi_{X \ 0 \ C} \ P_X] \ \varpi \ [\Box \sim \neg \textit{notify (Y, n)}]]$$

**Equation 1. Safety-basic property**

The fourth requirement essentially describes the real operation of the system (liveness property [10]) and can be formally expressed in the following postulate

*A simple event system is a system that exhibits only traces satisfying the following:*

$$\sim[\sim(F \cap S_Y) \Psi [\Box \sim(publish(X, n) \varpi n \cap N(F) \Psi \Box notify(Y, n))]]$$

**Equation 2. Liveness property**

The liveness condition describes under what circumstances a notification is delivered to a consumer. It reads as: "If at some point in time a consumer *Y* registers (by subscribing to a filter *F*) to receive every notification matching filter *F,* then from some time onwards when a provider *X* publishes a notification that matches the filter *F* will be propagated to consumer *Y*".

These two properties, essentially guarantee that "nothing bad" will happen and that "something good" will eventually happen throughout the operation of the system. It is shown in the literature [11, 12, 13] that all properties of a system can by expressed by intersection of clauses in either of the two properties above.

As one may notice although these properties describe what happens in the basic operation of an event system, there are no temporal or ordering constraints. However, ordering is an important aspect for the operation of the ISS framework, as it allows inference about the relationships of the events happening in an environment, and characterisation of an environment (which may allow further inference on what functions or mechanisms to deploy – i.e. composition, one of our main goals for ANA). As not all events are necessarily correlated total ordering, which is hard to achieve especially in a concurrent or distributed environment and restricts scalability, is not necessary. Local FIFO and causal ordering on the other hand is what we are interested in: i.e. ordering should be respected temporally when events occur local and should be achievable globally among events that are correlated somehow with a causal relationship.

To include the FIFO-local and causal ordering requirement for an event system we need to extend the safety property to include the following clauses

*An event system respects FIFO-producer ordering if it only exhibits traces satisfying the following requirements:*

$$n_1 \Box n_2 \varpi \Box[publish(C_1, n_1) \varpi \Box publish(C_1, n_2)] \Psi \neg \Box[notify(C_2, n_2) \varpi \Box notify(C_1, n_1)]$$

**Equation 3. Safety-FIFO property**

*An event system respects causal ordering if it only exhibits traces satisfying the following requirements for every $k \geq 2$:*

$$\Box(1 \geq i, j \leq k),\ i \Box j\ \Psi\ n_i \Box n_j\ \varpi$$
$$\Box[publish\ (C_1, n_1)\ \varpi$$
$$\Box[notify\ (C_2, n_1)\ \varpi\ \Box[publish\ (C_2, n_2)\ \varpi$$
$$\ldots$$
$$\Box[notify\ (C_k, n_{k-1})\ \varpi\ \Box publish\ (C_k, n_k)\ ]\ldots\ ]\ ]\ ]\ \Psi$$
$$\neg\Box[notify\ (Y, n_k)\ \varpi\ \Box notify\ (Y, n_1)]$$

**Equation 4. Safety-causal property**

Equation 3 states that notifications which are published by a component $C_1$ should not be delivered to a component $C_2$ in order different from the order that they have been published. Equation 4 says, is that if there is a sequence of components (consumers and/or providers), and one of them say $C_i$ publishes a notification $n_i$, which is propagated to a component $C_{i+1}$, then if $i<k$, component $Y$ should not be notified about $n_1$ after it was notified about $n_k$.

Note that causal ordering practically implies *FIFO* ordering as well, which means that notifications are delivered in the same order that they occur. Therefore the second clause practically obsoletes the first one when events are related causally.

Now that we have managed to define the interface specification for the correct behavior of our event-system, we enter inside the black-box to explain how the engineering of the the ISS system satisfies its specification and wherever needed evaluate the truthfulness of out postulations.

### 5.5.1 Evaluation of the Safety-basic, Safety-causal and Liveness properties

Looking at the safety-basic property in ISS the first requirement (*a component receives only notifications it has currently subscribed*) it is guaranteed by design at the multiplex construction time. When the multiplex is constructed by processing the consumer requirements specification, a wiring is determined that defines a fixed circuitry for the lifetime of its operation only between the consumer and the events (producers) or its interest. Errors, if they appear will be due to erroneous wiring either from software bugs in the implementation of the processing of the requirements specification, or due to erroneous knowledge stored in the ontology; in any case these are neither design nor run-time errors. In the current prototype we have done rigorous testing with different scenarios to try and eliminate any such errors.

The second and third requirements (*component receives notifications that have been previously published* and *a component receives each notification at most once*) essentially demand that there are no lost notifications and that no false positive notifications are propagated, respectively.

To achieve the former we had two options. If the ISS framework was implemented as a service running on its own (daemon) process we would counter lost notifications by providing sufficiently large queues to hold the incoming notifications from the publish interface. However as the number of event producers increases and the rate at which notifications are triggered can vary, it is hard to assess what a "sufficiently large" queue can be. Therefore adaptive queue sizes would be needed. This imposes an additional computational overhead every time the queues are resized, which at the same time can impose performance degradation for the rate the system processes/propagates the notifications. A switch in thinking contributed in our choice to implement ISS as a shared library. What this means is that each process essentially processes (propagates) by using ISS its locally generated notifications within its own scheduling quantum! As the need for access competition to the critical parts of the shared memory is distributed among the *mux* data structures of interest, the waiting time for each process it minimised (and moreover is decreased as the number of intermediate *mux* allocations increases – which is an amortisable wiring decision). Furthermore, there is no need for queues maintenance within the ISS framework; instead in the worst case where events within an event source are generated at a higher rate than they are consumed, each event source is free to choose a best strategy for managing them, either postponing them, or aggregating them, waiting sleeping or queuing them internally or even traffic shaping them (this was also shown and argued in Figure 9b, when we explained the fluctuation in variance). In any case the problem is therefore distributed (mitigated) and localised for better management. And although this is a theoretical worst case handling strategy, our implementation decisions has been justified by experimental results which have shown that there is no real concern even when events occur near line speeds (Figure 10). In Figure 12, we show the triggering of 25 events at random times in a multiplex with 10 event sources. The strict monotonic increase in the excitation times proves that the notifications are delivered in FIFO-order and the existence of a red bar (delivery) adjacent to each yellow bar (event firing), shows there were no lost notifications (the experiment involved 300 event triggers in total with the same behavior).

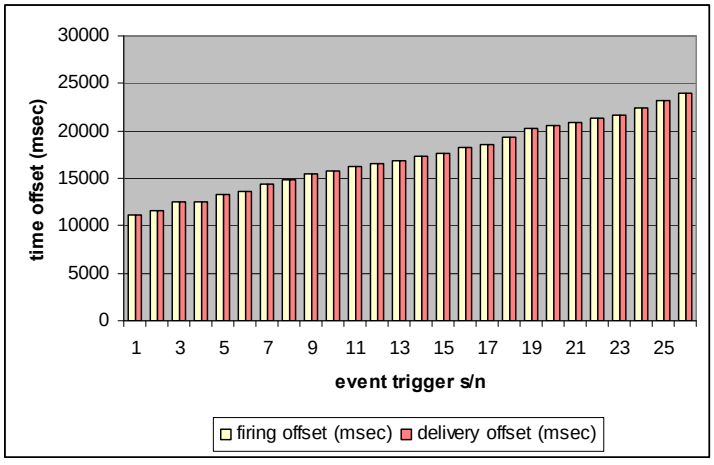



**Figure 12. Subset of 300 event notifications in a multiplex of 10 event sources (FIFO ordering, No lost notifications)**

To guarantee that each notification is delivered at most once (no false positives), there are a number of mechanisms used within the ISS framework to tackle different facets of this issue.

First of all every notification appearing in the multiplex is uniquely identified through the the *id* of the data structure of its origin. At the same time its current state within the intermediate structures in the multiplex is marked through a flag which is cleared right after it has been propagated to the next processing stage (*mux*) or consumed, thus making sure that no stale state is retained that may cause the re-evaluation of the event. In the cases of events that have temporal validity and need to be "remembered" for an amount of time, there exists a *temporal memory* mechanism to make sure that the current state of the event is preserved for as long as the event is valid. This however is guaranteed not to cause re-triggering the same notification by design, only contribute in another notification's propagation (in case the two are causally dependent). To elaborate on this a bit more, we remind the reader that due to the distributed operation of ISS (as a shared library), processing steps (i.e. notification propagations) happen only in response to single events occurring in the context of a producer's process, and not otherwise. As a result a notification may reach a consumer only if an event has actually occurred. Those notifications whose state has been temporally cached within the framework (by means of the *temporal memory* structures) can only *contribute* in the further propagation of a notification that has excited the framework and not in their own self-propagation.

Another issue concerns the false positives. There is the always the possibility that events similar to "events of interest" may occur at unexpected times for a number of reasons, and so their appearance signify nothing but "white noise" unless a number of other conditions are met. Such conditions are possible to prevent within ISS through reinforcing causal ordering, localising event composition, and realising sophisticated *mux* operations in the multiplex design.

An illustrating example of false-positive prevention is the following: Let's assume that a notification is wished to be delivered to a consumer when a TCP server connection request appears providing the input host interface through which the request appeared. A packet sniffer serves as an event source triggering a notification when a TCP SYN packet is detected delivering as notification data the source endpoint information as well as the interface info. The consumer may then use the information to carry out some "advanced" operation (interface accounting, load balancing, etc). However this alone (appearance of a TCP SYN packet) does not constitute evidence for the existence of a TCP connection establishment, and therefore if the consumer was notified for every TCP SYN packet appearance it would not operate as expected. For this reason we deploy a second event source is a monitor of the TCP protocol state, which triggers a notification whenever a connection with for certain client endpoint information is established. A multiplex wiring to serve our purpose is shown in Figure 13.

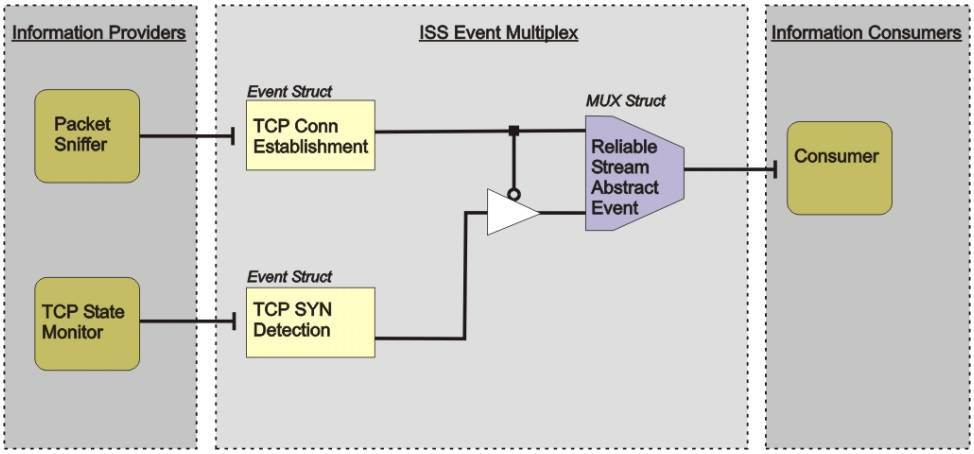



**Figure 13. Event multiplex for reliable stream abstract event detection**

Whenever a TCP SYN event occurs a notification is triggered. This notification is cached at the *mux* for some memory interval (which corresponds to the TCP connection timeout period). At the same time a *mux* operation that reads the source information of the packet "activates" the second event source with access trigger (pattern match) the packet source information. The TCP state monitor may try to fire a notification at every TCP connection establishment, but its event data structure is only activated (excites the multiplex) whenever the notification data match the access trigger. If these correspond to the TCP SYN source endpoint, the notification reaches to the *mux*. At the *mux* if the memory interval of the TCP SYN notification has not expired, the right event pattern has occurred and the notification is finally delivered to the consumer. This operation shows one way of how the property of causal ordering, and composition within the framework can leverage correctness and safety in the notification delivery mechanism. Figure 14 evaluates the instantiation of this operation in ISS, by showing the times that events occur and the times a notification is delivered to the consumer. That is, a TCP connection event establishment event is propagated only after a TCP SYN is detected before it (which permits its propagation). Moreover a notification is triggered and delivered to the consumer only when the TCP connection is establishment has occurred within the validity period that the TCP SYN detection is cached in memory.

| **Excited Element** | **firing offset (msec)** | **memory (msec)** | **notify consumer offset (msec)** |
|---|---|---|---|
| TCP Conn | 0 | | |
| TCP SYN | 2000.005029 | 30000 | |
| TCP Conn | 11000.00363 | | 11000.0109 |
| TCP SYN | 41000.00559 | 30000 | |
| TCP Conn | 50000.00447 | | 50000.0095 |
| TCP SYN | 90000.00531 | 30000 | |
| TCP Conn | 150000.0078 | | |
| TCP Conn | 152000.0045 | | |
| TCP SYN | 160000.0039 | 30000 | |
| TCP SYN | 162000.0059 | 30000 | |
| TCP SYN | 173000.0036 | 30000 | |
| TCP Conn | 180000.0045 | | 180000.014 |

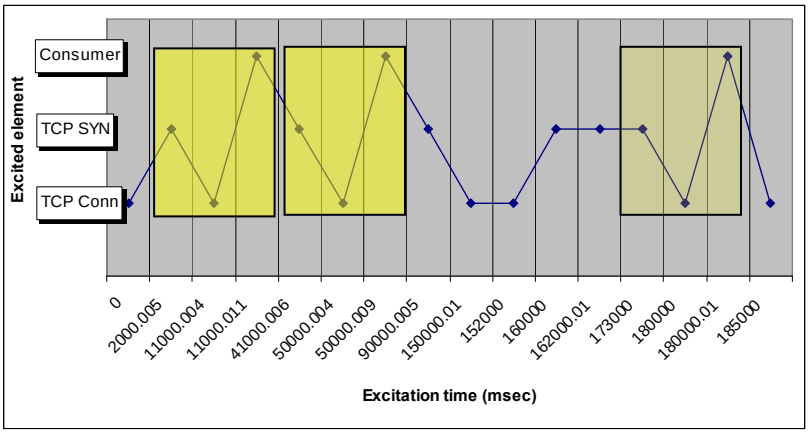



**Figure 14**

Regarding the liveness property (*a component receives all future notifications matching one of its active subscriptions*), its interpretation in the context of ISS suggest that an event trigger from a subscribed source does not deterministically result in the delivery of a notification to a consumer, but rather only after the multiplex construction is complete and thereafter if its processing within the multiplex *contributes* to the conditional generation of a notification. This is indeed guaranteed, for two reasons. First of all, the subscription to events results in the construction of a multiplex wiring, and operation starts only after the completion of the multiplex construction. The fixed wiring thereafter enforces the propagation of all event triggers that enter the multiplex. Second, the absence of queues from our design and implementation of the multiplex (discussed earlier), enforces that event propagation/processing is immediate (events are not likely to disappear by queue overflows or other queue handling issues), and that it either happens, once the trigger enters the multiplex, or it does not at all, if it does not enter the multiplex. Finally to the extend that an event *contributes* to the generation of a consumer notification, and that is subject to the event detection pattern encoded in the intermediate *mux* structures, the notification will actually reach the consumer.

This last statement is seen in the example of Figure 13. At first looking at the evaluation, it may seem that compliance to the liveness is not accomplished. However, one can see that this is a false perception when she realizes that in this example the consumer subscribes to some *event abstraction* (the abstract event “reliable stream”), instead of the explicit low level events of TCP connection detection and TCP SYN detection. The liveness property is seen to hold in Figure 14, as a notification is always delivered to the consumer, each time the correlation of the low-level events satisfies the detection pattern represented in the reliable stream event abstraction (yellow rectangles).

Note that it is possible for the consumer to subscribe to the actual event sources directly if it wishes but in this case the benefit of indirection and abstraction is lost and the correlation processing of low level events will be left to the consumer. This case was shown in the experiment of Figure 12. The operation/capability of event pattern detection and composition is properly introduced and explained in detail in the following section.

Finally as far as the ordering (safety-causal) is concerned if is seen in both earlier examples (Figure 12 and Figure 14) that it is respected in ISS. In Figure 12, all events are

temporally delivered in the exact order that they occur (FIFO), and in Figure 14 the notifications about the presence of a reliable stream are delivered in the order that they occur and also each consumer notification is generated only when a TCP SYN packet has caused the establishment of a TCP connection. The semantics and engineering of an event abstraction captures in this case the causal relationship of the two lower level events.

## 5.6 Facilitating Composition and Distribution

For certain applications and environments the expressiveness of event subscriptions by consumers needs to go beyond expressing requirements in terms of primitive low level events. In these cases a service for *event composition* facilitates the management of a set of low level events, enabling the consumer to specify its interest in a way that is semantically meaningful to it. Practically this service introduces one or more levels of indirection that enable the consumer to subscribe to an *event pattern* which in return is composed (transparently) through a correlation of lower level of events.

Looking a bit deeper in sophistication, one of the main design principles of ISS has been the use of what we call *event abstractions* to decouple semantic dependencies between consumers and providers, by means of event composition. Event abstractions are *filters* which aim to capture such event patterns; they enable indirection, composition and aggregation of lower level events. In this section we evaluate this service by means of an example that aims to demonstrate the purpose that this service facilitates. Before hands we set a formal context that specifies the required operation of this service and through which we can argue for the design decisions made.

### *5.6.1 Composite Event Definitions*

A composite event service is based on the notion of a *composite event abstraction* that allows subscribers to avoid being overwhelmed by a large number of primitive notifications. The composite event is published whenever a certain pattern of events occurs. Note that any of the so called primitive events may as well be another lower-level event abstraction (!), enabling multiple levels of abstraction/indirection. The subscriber to the composite event (consumer) can subscribe directly to a complex event pattern as opposed to having to subscribe to all the primitive events that make up the pattern. In this case the correctness properties examined in the previous section (safety, liveness) needs to hold separately between the subscriber and the composite event, and each composite event and its composing events, as opposed to between the consumer and the primitive events!

Often the subscriber of the composite event may be interested in the notification data of the primitive events that caused the composite event to occur and that is managed in ISS through the virtual mapping mechanism of the data delivery component (see general design section). Since a composite event is composed by primitive events following a well defined set of rules every composite event can be assigned an *(composite) event type* of its own, which allows the subscriber to perceive it as yet another primitive event.

Knowledge about the event composites can be fixed (hardcoded), however in ISS we propose the use of a knowledge base supported by a domain-ontology to facilitate extensibility in accommodating new composite events or allow any composite event to be composed/expressed by alternative primitive event sets (producers).

The following formalizations are in accordance with [14].

Definition 1: *Every composite event c has a composite type* $\tau_c$ *and belongs to the composite event space C,*

$$(c: \tau_c)\ 0\ C$$

*and consists of an interval timestamp* $t_c$ *and a set of composite subevents* $\{c_1, c_2, \ldots c_k\}$,

$$(c: \tau_c) = (t_c, \{c_1, c_2, \ldots c_k\})$$

The interval timestamp states when the event occurred and how long it is valid for. Since in distributed event systems there is no notion of global time [15], the temporal aspects of composite events can be captured better using *interval timestamps* [16], expressing the clock uncertainty at the event framework instance as well as its validity duration from the first composing event to the last.

The temporal relationships between composite events can be captured by the following two relational operators which will be used as seen later in the event detection formalization:

Definition 2: *An interval timestamp* $t_c$, *has start time* $t_c^l$, *and end time* $t_c^h$

$$t_c = [t_c^l; t_c^h],\ (h: high, l: low),\ t_c^l \leq t_c^h$$

*and they may be ordered with either of the following two ordering operators*

*(strong order: non overlapping intervals)*

$$t_{c1} < t_{c2} \cdot t_{c1}^h \leq t_{c2}^l$$

*(weak order: overlapping intervals)*

$$t_{c1} \sqsupset t_{c2} \cdot (t_{c1}^h \leq t_{c2}^h)\ \omega\ (t_{c1}^h = t_{c2}^h\ \varpi\ t_{c1}^l < t_{c2}^l)$$

## 5.6.2 Composite Event Detection

The semantics of the composite event detection process can be elegantly captured by means of event detection automata [17], which are extended finite state automata that incorporate temporal semantics. This formal representation has the advantages that (a) enables a well understood computational model that can easily map to an implementation, (b) their expressiveness allows predictable discrete resource usage in a system, and (c) any regular expression language is a mere 1-1 or 1-many mappings of the defined operators for the automata, which express the event detection patterns.

A detection automaton comprises of a finite number of states and transitions among them. Each state is described by an *input domain Σ*, which is a collection of *event sets* (*A, B, C, etc*). An event set comprises of one (primitive) or more (composite) events eligible for firing whose processing may cause the transition to a next state. In any given state only these events are considered of all the existing events in the system.

A detection automaton can have one of 4 different types of state (Figure 15). A detection starts at the *initial state. Ordinary states* are normal states the event automaton moves to deterministically when one or more of the events in the input domain are detected. A *generative state* is the state entered at the moment that the composite event pattern is triggered, and therefore a composite event is generated. Finally a *generative time state* is a generative state in which the automaton enters only temporarily for the duration of a timestamp interval while waiting for an event in the input domain to occur. If it does not and the timestamp expires, the event composite does not generate a notification. The *generative time state* captures the temporal non-deterministic conditions that may appear for the trigger of the composite event.

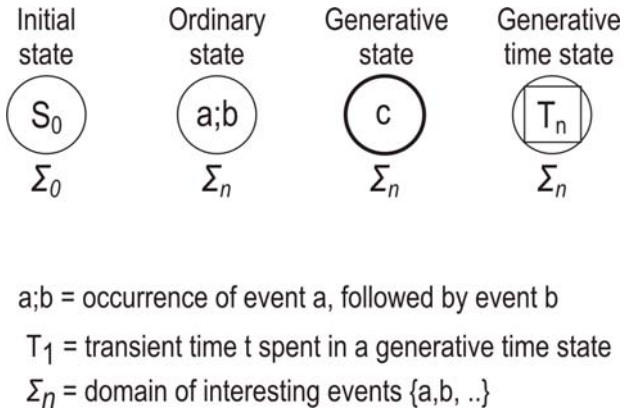



**Figure 15 Event detection automata states**

**The transitions between states can have one of two forms (**

Figure 16). *Strong transitions* take place when an event occurring at a state follow a previously detected event with strong ordering (<) (non overlapping timestamp intervals). *Weak transitions* take place when an event occurring at a state follow a previously detected event with weak ordering (□) (possibly overlapping timestamp

intervals). Strong and weak transitions therefore represent different temporal relationships between events detections. Finally, empty transitions (e-transitions), that is transitions not in-response to an event detection, are possible internally in the automaton and may occur non-deterministically.

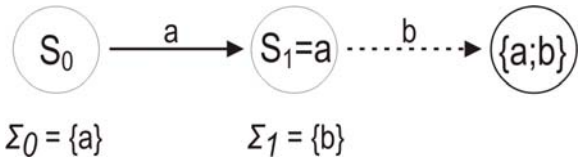



**Figure 16 (a) Strong transition (b) Weak transition**

**In**

Figure 16 an event *a* occurs after the initialisation of the automaton, leading to a strong transition to state $S_1$. The detection of event *b* then occurs triggering a weak transition from state $S_1$ to the generative state for the composite event notification. The detection of event *b* might precede the lapse of the timestamp interval of the detection of event *a*. This simple composite event captures the event pattern where event *b* occurs after event *a*.

To exemplify the operation of the event detection automaton, we present an example from the literature.

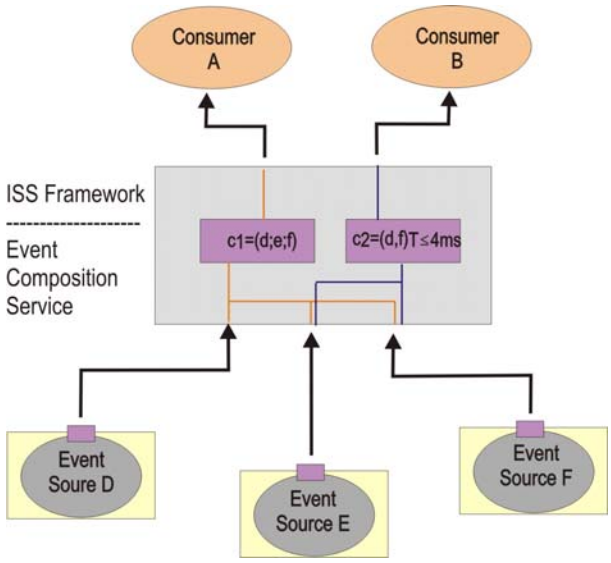



**Figure 17 Event system scenario**

In Figure 17 there are two *consumers (A, B)* and three event sources (*providers: D, E, F*). *Consumer A* is interested in the occurrence of events *d, e, f* and carries out an operation if they have occurred in sequence *(d;e;f)*. *Consumer B* is interested in the occurrence of events *d, f* and performs some operation when they have both occurred within an interval $t \leq 4$ *msec* from each other.

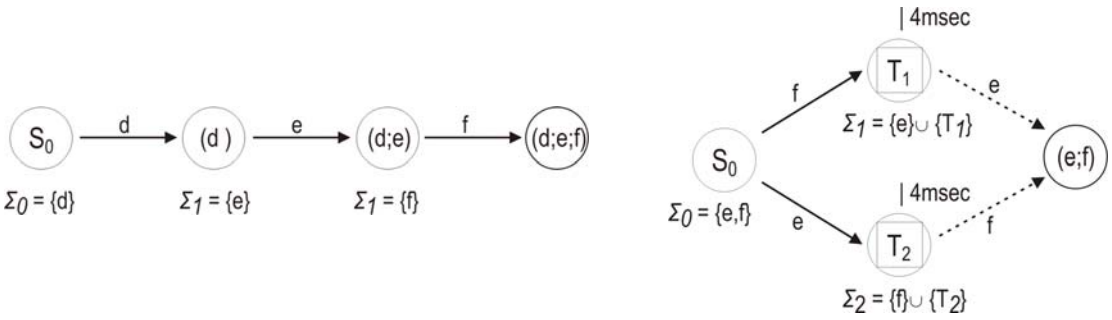
Without the use of the composition service the two consumers need to explicitly subscribe to their events of interest, so as to receive all notifications from them. *Consumer A* would subscribe for notifications from *provider D, E* and *F*, while *consumer B* would subscribe for notifications from *providers D* and *F*. In ISS this precipitates the existence of two event multiplexes, one for each consumer, each multiplexing the appropriate corresponding event sources.

Thereafter whenever notifications are published each consumer would internally need to identify whether the anticipated event pattern has occurred and decide whether to carry out its intended operation. For example *consumer A* would need to be able to infer if the events have occurred in the sequence *(d;e;f)* or any other in order to decide if it needs to respond to it. Although in this example it is easily feasible as each notification in ISS may optionally convey an event occurrence timestamp (which will allow each consumer to infer the event occurrence pattern), however in more complex scenarios this may turn out to be an intensive task, often imposing design constraints for a consumer module (e.g. if base events occur at a high rate the consumer operation will not permit an asynchronous design).

If an event composition service is used however there will be two composite events constructed internally in the event system. The logic required for constructing expected pattern detection automaton is mitigated to the composition service within ISS.

Looking at them from the provider's point of view each detection automaton is perceived as a consumer (let $C_1$ and $C_2$). Looking at them from the consumer's point of view on the other hand each automaton represents an event (let $c_1$, and $c_2$). In other words they are at the same time both consumers (of base events) and providers (of composite events), and thus they exhibit properties of both identities. As event abstractions they represent the exact event patterns that *consumer A* and *B* expect, that is $c_1 = (d;e;f)$ and $c_2 = (d, f)_{T \leq 4msec}$ respectively.

As *A* and *B* essentially subscribe to the composite events ISS needs to be able to capture and so specification of the expected event patterns needs to be accurate. This is formalised using the extended event automata presented earlier.




**(a) Detection automaton for $c_1$** **(b) Detection automaton for $c_2$**

Figure 18Figure 18.a shows the automaton for composite event $c_1$. The automaton starts at state $S_0$ which is the initial state and with input domain *{d}*. Upon detection of d from *provider D*, it moves to ordinary state $S_1$ with input domain *{e}* through a strong transition, and upon detection of *e*, it moves to ordinary state $S_2$ with input domain *{f}*

again with a strong transition. When *f* occurs the automaton moves to the generative state $C_3$ where the composite event $c_1 = (d;e;f)$ fires a notification to the *subscriber A*.

Similarly Figure 18.b shows the automaton for composite evebt $c_2$. This automaton now starts at initial state $S_0$ which is the initial state and with input domain *{d, f}*. If either of these events is detected from their respective providers, the automaton will move to one the two (or both) states $T_1$ or $T_2$. $T_1$ is a generative time state with input domain $\{f\}\chi\{T1 = 4msec\}$, while $T_2$ is a generative time state with input domain $\{d\}\chi\{T2 = 4msec\}$. A weak transition from $T_1$ upon detection of *f* or from $T_2$ upon detection of *d*, within the time interval of *4msec* in either case, will lead to a generative state $C_3$ that signals the composite event $c_2 = (d, f)_{T \leq 4msec}$ and a notification will be sent to *B*.

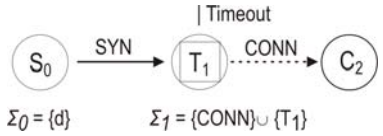
## 5.6.3 Composition in ISS

In ISS the composite event detector is practically embodied in the mux aggregator structures within the multiplex, by means of *event abstractions*. To evaluate the operation of the service composition mechanism we use the following example, which captures the incentive of information sensing as a means to leverage functional composition in ANA.

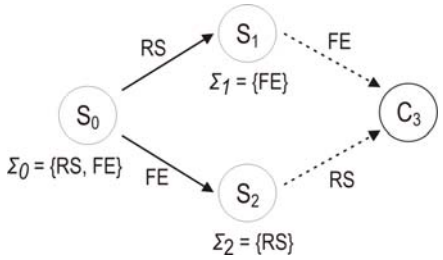
We assume a consumer which is a module that deploys a link layer error correction mechanism dynamically upon detection of a composite event. The composite event captures the following event pattern. We assume three event sources. As in the last example, a packet sniffer (*provider D*) is expected to fire a notification upon detection of a TCP SYN packet, and deliver the source information of the packet as well as the interface where the packet was detected. A TCP state monitor (*event source/provider E*) instantiates a second event source that triggers notifications whenever a TCP server connection is established (we may assume that the TCP monitor is sophisticated enough to only fire notifications for certain flow connections, or not, in which case the information packet source from the packet sniffer may be used to dynamically program the TCP monitor– the sophistication of ISS enables this functionality as shown in the previous evaluation example earlier). These two events compose an event abstraction $c_1 = (d;e)_{T \leq TCPcontimeout}$ that signifies the detection of a "*reliable stream request*". This means that the composite event is generated when event *d* is followed by event *e* within a time interval that equals the TCP connection timeout. The third event source (*provider F*) is a link layer frame error monitor which fires an event whenever the number of link layer frame errors at an interface exceed a watermark level. Again ISS is flexible enough (as in the example of the TCP state monitor) to allow the use of information from the packet sniffer's notifications to dynamically program the frame monitor for which interface to monitor; alternatively we may assume the existence of such a frame error monitor on every interface and let the cross-correlation of notification information from the two sources for some later stage in the detection process (through a *mux* operator), or finally we can leave it for the consumer to validate it. A second composite event abstraction $c_2 = (c1, f)$ captures the pattern "*if there exists a reliable stream request from an interface on which frame errors are observed*", signifying the need for link layer error correction.

In Figure 19.a is shown the detection automaton that implements the "*reliable stream request"* composite event; in Figure 19.b is shown the corresponding automaton for the

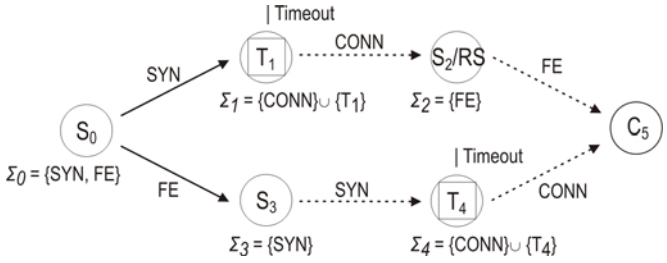
"*error correction need"* composite event. Finally Figure 19.c shows the complete event automaton for the entire system that will be used to generate the multiplex.




**(a) Reliable stream detection automaton**




**(a) Error correction detection automaton with reliable stream abstract event**




**(c) Complete dynamic link layer error correction deployment automaton**

**Figure 19.**

In Figure 19.c the initial state $S_o$ has input domain the TCP SYN packet detection event. When this is detected the automaton moves strongly to a two independent states $T_1$ and $S_3$. $T_1$ is a generative time state where the automaton waits for a time interval equal to TCP connection timeout, and has input domain a TCP connection establishment event. If in that time interval the TCP connection establishment event occurs, a weak transition brings that branch of the automaton to state $S_2$ ($C_2$ in Figure 19.a). This is a generative state for the composite event $c_1 = (d;e)_{T \leq TCPcontimeout}$ denoting the existence of a reliable stream and has input domain the detection of frame errors. The detection of frame errors will cause the transition of $S_2$ to the state $C_5$ ($C_3$ in Figure 19.b) which is a generative state for the composite event $c_2 = (c1, f)$ and therefore notify the *consumer A* which should deploy the error correction mechanism.

On the other branch $S_3$ is an ordinary state with input domain the frame error detection event as well. If this event occurs then the automaton will weakly move from state $S_3$ to state $T_4$ awaiting for a reliable stream composite event to occur in its input domain. If a reliable stream request composite event ($c_1$) is detected once again the automaton will move to $C_5$ state which is a generative state for the composite event $c_2 = (c1, f)$.

One may argue that the presence of the $S_3 - T_4$ states may be skipped. Although this is true, however their presence make the event detection process more complete allowing that either order of events is part of the event pattern for $c_2$.

Figure 20 shows a block diagram of a multiplex construction that implements this automaton in ISS. It is worth noting that this is not the only possible multiplex but one that can be deterministically be derived from the automaton presented earlier. Optimisations of a multiplex structure or different automata that capture the same composite event patterns may lead to equally alternative multiplexes in ISS.

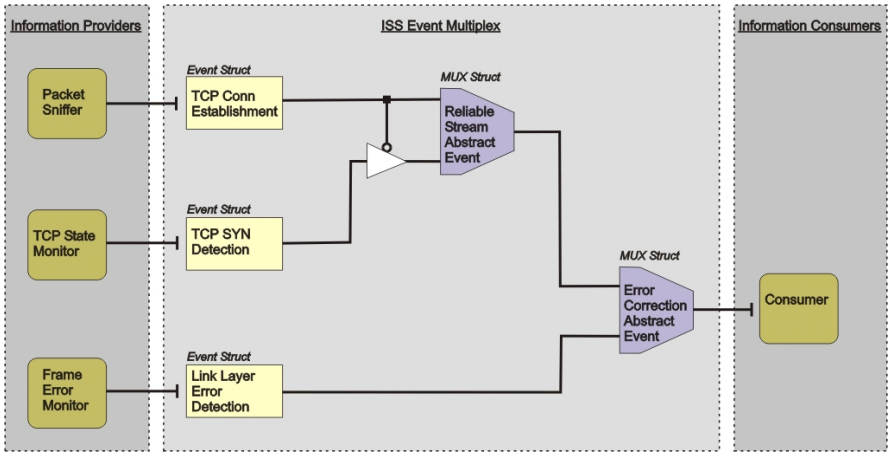



**Figure 20. Event multiplex for the error correction abstract event detection**

Finally, upon execution of the above multiplex for this example we are able to evaluate its correctness in Figure 21, where we can see that a notification is trigger to the consumer only if the event occurrence pattern of the low level events satisfies the sequence of states in the event automaton in Figure 19.c (green rectangular area).

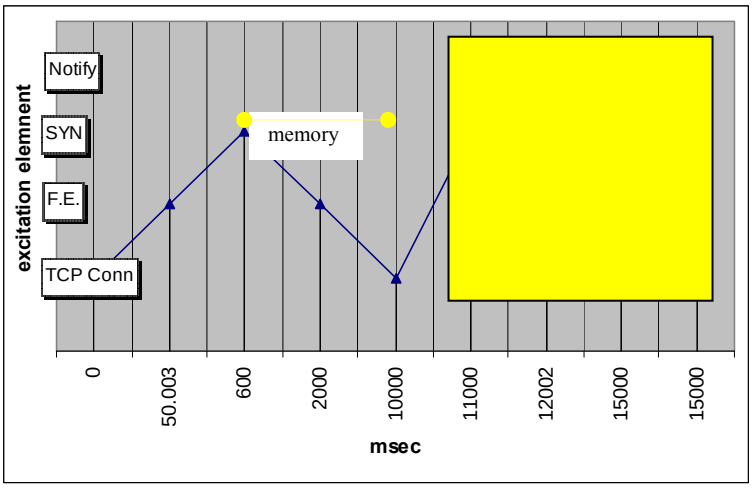



**Figure 21. Evaluation results of the operation of the composite event detection mechanism**

## 5.6.4 Distribution & Mobility

So far most of the descriptions and examples presented here have focus on scenarios within a host system. However, a composite event service can also implement the detection of composite events in a distributed fashion. The fundamental principle of the composition service being the composite event, allows *localization of the detection*

*process* for the composed event. Cascading and recursive event composites are possible and easy to decompose in a very flexible fashion that enables the placement of the detection mechanism or different parts of it, in disparate locations across a network. Among the advocated benefits of event composition is efficient resource use that derives from the localization property of composite events and which makes the distribution of an event system economic through aggregation (as opposed to exchanging notifications of all primitive events across a network infrastructure). In particular, the amount of communication is reduced because event detectors for sub-expressions or sub-patterns can be positioned close to the primitive event publishers that generate the events necessary for the detection. The detection automata described earlier directly support distribution because they can subscribe to composite events detected by other automata in the system.

Taking the discussion one step further one can implement easily *mobile composite event detectors* that can be agent-based entities co-existing within event brokers. These event brokers essentially accommodate multiple independent composite event detection automata (as described earlier). The detection automata essentially are the "*citizens*" of a large distributed event distribution and information exchanging hosting system in which they process events and exchange data. When a consumer or event subscriber submits an new composite event subscription, this is decomposed to one or mobile event detectors that can be instantiated at an event broker and are responsible thereafter for the detection of the sub-patterns of the event composite. Each mobile detector will have a lifecycle that includes its instantiation, task delegation and workout, and finally its collapsing/destruction when it is no longer needed.

This model should be particularly easy to implement in ISS as the fundamental event detection entity is the multiplex. It can be constructed from consumer requirement specifications and be decomposed easily in more than one cooperating sub-multiplexes at the composite event abstraction level (as shown in the earlier example of dynamic error correction request detection). At the same time it is an opaque and modular enough entity that can be constructed in one ISS event broker and executed in another.

Remote communication between and among the ISS framework brokers is implemented easily in a way that decouples the event detection from the event distribution. The remote ISS facility as briefly described in the general description accommodate a set of plug-ins that offer different transports (which may meet different criteria for delay, bandwidth, etc) for the remote communication of events over a network. Based on the consumer requirements and the composite event type one or the other plug-in mechanism may be wired to a multiplex for remote event exchange. And the flexibility is not limited to this option only as a consumer may easily operate as a more sophisticated composite event broker between remote ISS instances, leveraging complex aggregation and other computational operations before distribution.

The remaining difficulty for the distribution of mobile event detectors is the implementation of suitable and possibly extensible *distribution policies* that will enable the effective distribution of multiplexes across a infrastructure of ISS runtime brokers.

## 5.7 Conclusion & Future Work

The development of the ISS architecture is by no means complete. At this stage, it can be used to provide the core functionality that was evaluated in the previous sections, but there is are still plenty of features to implement, and this will be the goal of the upcoming year in the ANA project for this task. Namely work is anticipated on the ontology manager as well as the distribution and mobility aspects that were briefly introduced in the previous section. Finally as the current ISS core is not yet embedded in the ANA architecture one work thread in the coming year will be involve an effort to package ISS in a functional block. Finally time allowing future work that is anticipated in this task will include research on basic logic that will bind/glue the event based output of the ISS architecture with the composition framework in a way that may drive evolving functionality in the ANA system.

# 6 CONCLUSIONS

This deliverable introduced the work that has taken place as part of Task 2.2 in ANA. Drawing from the identified capabilities that are expected to enable autonomicity in a system, throughout the past year we have focused on the analysis and development of a framework that enables functional composition and an architecture that provides event-driven awareness; the latter aims to drive the process of functional composition.

In the past year we have however covered a substantial ground towards achieving our goals. This work is not completely integrated in the ANA prototype at the time of the writing of this document, however according to the time plan set, it is expected to reach completion in the near future.

The future directions of the work in this task for the remaining of the project include more extensive evaluation of this work and exploration of real world scenarios that will demonstrate its benefits. We anticipate enough time in the future for complementing this work by exploring the space of introducing logic in the operational cycle of an autonomic system:

*Sense* → *Infer/Decide/Select* → *Adapt* → *Sense*

# REFERENCE LIST

## FUNCTIONAL COMPOSITION FRAMEWORK


1. Autonomic Network Architecture Project. Online available at: http://www.anaproject.org
2. F. Sestini, “Situated and Autonomic Communication—an EC FET European initiative”. ACM Computer Communication Review, Vol 36, Number 2, April 2, 2006
3. S. Schmid, M. Sifalakis, D. Hutchison. *"Towards Autonomic Networks"*. Proc. 3rd Annual Conference on Autonomic Networking, Autonomic Communication Workshop (IFIP AN/WAC), Paris, France, September 25-29, 2006.
4. D. Engler and M.F. Kaashoek, "*DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation”*. ACM *Computer Communication Review*, Vol 26, Number 4, October 1996.
5. M. L. Bailey, B. Gopal, M. A. Pagels, L. Peterson, and P. Sarkar. “*PathFinder: A Pattern-Based Packet Classifier*”. Proc. 1st Symposium on Operating System Design and Implementation. USENIX, November 1994.
6. S. McCanne and V. Jacobson. “*The BSD Packet Filter*”. *Winter USENIX*, pp. 259-269. USENIX, January 1993.
7. N. C. Hutchinson and L. L. Peterson. "*The x-kernel: An Architecture for Implementing Network Protocols*". IEEE Transactions on Software Engineering, Vol. 17, pp. 64--76, January 1991.
8. M. Rozier et. al. “*Overview of the Chorus Distributed Operating System*”, Proc. USENIX Workshop on Micro-kernels and Other Kernel Architectures, pp. 39-69.
9. A. Montz et Al., “Scout: A Communications-Oriented Operating System”, In Operating Systems Design and Implementation, pages 200, 1994.
10. R. Morris, E. Kohler, J. Jannotti, M Kaashoek, “The Click Modular Router“, In Proc. of ACM Symposium on Operating Systems Principles, pages 217-231, December 1999.
11. S. Schmid, T. Chart, M. Sifalakis, A.C. Scott. ***"A Highly Flexible Service Composition Framework for Real-life Networks".*** Computer Networks, Special Issue on Active Networks, Elsevier, Vol. 50/Issue 14, p. 2488-2505, 5 October 2006.
12. L. Ruf, K. Farkas, H. Hug, B. Plattner. “***Network Services on Service Extensible Routers***”. Proc 7th Annual International Working Conference on Active Networking (IWAN 2005). CICA, Sophia Antipolis, France, November 21.-23, 2005.
13. S.Merugu, S.Bhattacharjee, Y.Chae, M.Sanders, K.Calvert and E.Zegura, *"Bowman and CANEs: Implementation of an Active Network"*. Proc. 37th annual Conference on Communication, Control and Computing, Monticello, Illinois, September 1999
14. G. Coulson, G. S. Blair, D. Hutchison, A. Joolia, K. Lee, J. Ueyama, A.T. Gomes, Y. Ye, *“NETKIT: A Software Component-Based Approach to Programmable Networking”*. ACM SIGCOMM Computer Communications Review , Vol 33, No 5, pp 55-66, October 2003.
15. D.UJ. Wetherall, J.V. Guttag and D.L. Tennenhouse, “*ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols”*, Proc. of OPENARCH, April 1998.
16. Y. Yemini and S. da Silva, “*Towards Programmable Networks*”. Proc. of IFIP/IEEE International Workshop on Distributed Systems Operations and Management, October 1996.
17. M.W. Hicks and J.T. Moore and D.S. Alexander and C.A. Gunter and S. Nettles, “*PLANet: An Active Internetwork*”. Proc. of IEEE INFOCOM (3), pp. 1124-1133, 1999.

18. B. Schwartz et al., "*Smart Packets for Active Networks*", Proc. of OPENARCH, 1999.

19. M.W. Hicks and S. Nettles, "Active Networking Means Evolution (or Enhanced Extensibility Required)", Proc. of IWAN 2000, October 2000

# INFORMATION SENSING AND SHARING ARCHITECTURE

1. Borgia, E., Conti, M., & Delmastro, F., 'MobileMAN: Design, Integration and Experimentation of Cross-Layer Mobile Multihop Ad Hoc Networks', IEEE Communications, 44,7 (2006).

2. R. E. Gruber, B. Krishnamurthy, and E. Panagos. *"The architecture of the ready event notification service*". In Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Middleware Workshop, 1999.

3. L. Fiege, G. Muhl, and F. C. Gartner. *"A modular approach to build structured event-based systems"*. In Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02), pages 385--392, Madrid, Spain, 2002.

4. J. Bacon, J. Bates, R. Hayton, and K. Moody. *"Using Events to Build Distributed Applications"*. Proc. 7th. ACM SIGOPS European Workshop, Connemara, Eire, Sept.1996.

5. Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. 2001. "*Design and evaluation of a wide-area event notification service*". ACM Transactions on Computer Systems 19, 3, Aug. 2001

6. IBM T.J. Watson Reearch Center. *"Gryphon: Publish / Subscribe over Public Networks*". http:// researchweb.watson.ibm.com/gryphon/Gryphon, December 2001

7. Sifalakis, M., Hutchison, D., Sterbenz, J., Zseby, T., Salamatian, K., "*Functional Composition Framework, Autonomic Network Architectures"*, Deliverable D2.2, Feb 2007, http://www.ana-project.org/images/deliverables/D.2.2.-Func-Comp.pdf

8. Pnueli, A. *"The Temporal Semantics of Concurrent Programs*". In Proceedings of the international Sympoisum on Semantics of Concurrent Computation, July 02 - 04, 1979.

9. L. Fiege, F. C. G artner, O. Kasten, and A. Zeidler. *"Supporting mobility in content-based publish/subscribe middleware*". In Proceedings of ACM/USENIX/IFIP Middleware 2003.

10. Lamport, L. *"Proving the Correctness of Multiprocess Programs*". IEEE Transactions on Software Engineering 3, 2, pp 125-143, March 1977.

11. Alpern, B. and Schneider, F. B. "*Defining Liveness"*. Information Processing Letters, Vol 21, pp181-185, 1985

12. M. Broy, E.-R. Olderog. *"Trace-Oriented Models of Concurrency"*. *J.A. Bergstra, A. Ponse, S. A. Smolka (Eds.),* pp. 101 - 195, Elsevier Science B.V., 2001 (Book chapter)

13. Felix C. Gartner. *"Fundamentals of fault-tolerant distributed computing in asynchronous environments*". ACM Computing Surveys, 31(1):1--26, March 1999.

14. G. Mühl, L. Fiege, L., P. Pietzuch. *"Distributed Event-Based Systems"*. 2006, XIX, ISBN: 978-3-540-32651-9, Springer.

15. Lamport, L. *"Time, clocks, and the ordering of events in a distributed system*". Communication of the ACM 21, 7, pp 558-565, Jul. 1978.

16. C. Liebig, M. Cilia, and A. Buchmann. *"Event Composition in Time-dependent Distributed Systems*". In Proceedings of 4th International Conference on Cooperative Information Systems, September 1999.

17. JE Hopcroft, J. D. Ullman, R. Motwani. "*Introduction to Automata Theory*, *Languages and Computation*", Pearson Education Limited (a.k.a. the Cinderella Book).

18. M. Sifalakis, M. Fry, D. Hutchison. *"A common Architecture for Cross Layer and Network Context Awareness".* In proceedings of 1st International Workshop on Self-Organising Systems (IWSOS '07), The Lake District, UK, September 11-13, 2007.