Applying Dynamic Separation of Aspects to Distributed Systems Security

Miguel García [a] David Llewellyn-Jones [b] Francisco Ortin [a]
Madjid Merabti [b]

[a] *University of Oviedo, Asturias, Spain*
[b] *Liverpool John Moores University, Liverpool, United Kingdom*

Abstract

Distributed systems are commonly required to be flexible and scalable, as the number and arrangement of the devices can easily vary. As in any system, it is necessary to exchange information in a safe and controlled way. Security in distributed systems is a complex issue which can produce several problems such as eavesdropping, phishing or denial of service. To overcome these problems, there are various security measures that can be applied in different ways. Common examples are encryption of communication, use of credentials or audit. This paper proposes the idea of using *Aspect Oriented Software Development (AOSD)* to implement security mechanisms in distributed systems. By applying dynamic separation of concerns using AOSD it becomes possible to make corrections in the software at runtime. In this way a distributed system is able to adapt its security measures when needed, and can vary in size and arrangement without compromising security. These changes can be applied even when the distributed system is running, without stopping its execution. Using the AOSD platform Dynamic and Static Aspect Weaving (DSAW) AOSD platform, we have implemented solutions for two common but difficult problems related to security in distributed systems: access control and data flow, and encryption of transactions. An evaluation of both implementations is presented to estimate the possible advantages of using dynamic AOSD in the development of distributed systems

Key words: security, distributed systems, dynamic separation of concerns, aspect oriented software development

Email addresses: BE37378@uniovi.es (Miguel García), D.Llewellyn-Jones@ljmu.ac.uk (David Llewellyn-Jones).
URLs: http://www.uniovi.es (Miguel García), http://www.ljmu.ac.uk (David Llewellyn-Jones).

1 Introduction

Technology is continually evolving better connected world, with improved communications and featuring devices that are now smaller and more powerful than ever. Information systems are increasingly distributed in both geographical and functional terms. Distributed systems involve the interaction between disparate and independent entities working towards a common goal [3]. For instance, nowadays mobile device users expect to be able to connect to the Internet from anywhere to check email or use social networks. The creation of *ad hoc* device networks in order to exchange information is also becoming increasingly feasible, to be used in different scenarios such as at big events (concerts, shows, competitions) or even in emergency situations.

These examples of networks must be flexible and scalable, as the number and arrangement of the devices can easily vary. Moreover, in general, the devices that make up these networks are heterogeneous as they may come from different kind of services and clients. For example, in a crisis management scenario there are different emergency services (police, fire department or army) and clients (firemen, nurses or soldiers). This makes it difficult to apply security measures uniformly. It is necessary to exchange information in a secure and controlled way. In any information system, security measures are important, but in these cases they are crucial as the transmitted information is often highly confidential. That is why one of the main challenges concerning distributed systems is security. The complexity of this issue relies on the different vulnerabilities existing in distributed systems. Information sources, transmissions, the intermediate nodes or the final receivers are the first points that need to be protected in any distributed system. It is necessary to analyse in detail all the threats and the possible solutions to them. In systems with a single device or in private networks, most of the security threats can be easily controlled using static hardware or pre-deployed software, but this solution is not suitable in distributed systems.

Several problems can arise, intentionally or not, in distributed systems. Examples include eavesdropping, denial of service (DoS), spoofing or repudiation of transactions. The vast majority of measures used to prevent or solve these problems are performed using software. Encryption, digital certificates, access control lists or monitoring are some of the solutions used. Since there are different techniques used in order to solve security problems, in this paper we analyse the benefits on using the *Aspect Oriented Software Development (AOSD)* [15] paradigm. This paradigm allows developers to make good use of the *Separation of Concerns* (SoC) principle [14] when developing applications. If the AOSD platform employed offers dynamic aspect weaving [38], it is possible to dynamically adapt distributed systems in order to include new security concerns when a new threat appears at runtime. Security measures can

be modified, inserted or removed with no need to stop the distributed system, and without changing the software or devices of which it is comprised. Corrections performed once the distributed system is running are motivated by two main causes: new vulnerabilities (not taken into account at design time) or performance (the software has security measures, but they are disabled for efficiency reasons).

In this paper we use an AOSD approach to solve two common problems in distributed systems security: access control and data flow, and encryption. These solutions are based on AOSD, using the *Dynamic and Static Aspect Weaving (DSAW)* [48] platform for their implementation. DSAW is an AOSD platform that supports static and dynamic code weaving, which among other things allows the modification of application functionality at runtime. As shown below, applying this technology it is possible to control the flow, access and encryption of data dynamically. In this way, the distributed system is able to adapt to security measures when required, and can vary in size and arrangement without compromising security. The results of these use cases might also be applied to other distributed systems security problems.

The remainder of this paper is structured as follows. Section 2 describes and analyses the main problems and possible solutions in distributed systems security. Static and Dynamic AOSD and the DSAW platform are presented in sections 3 and 4 respectively. An aspect-oriented implementation of the proposed solutions to the specific problems of data access and data flow and encryption are described in Section 5. In Section 6 we evaluate the advantages and performance costs of our proposal. Section 8 presents the conclusions and future work.

2 Distributed Systems Security

The security of a distributed system is founded on its ability to protect itself. As far as security is concerned, two main questions must be taken into account. 1) What must be protected. In general, distributed systems need to protect the data, services and system resources being used. 2) What must be protected from. There are several ways to attack systems, as there are different ways to repel these attacks. In the case of distributed systems, one of the main threats is unauthorised users attempting to exploit vulnerabilities to access the system. Although this is the greatest threat, other problems such us eavesdropping or DoS are apparent as well.

The security measures of distributed systems primarily try to prevent, detect and correct threats that endanger the integrity of the system. The issues related to security in distributed systems can be divided into three general ar-

eas [5]: confidentiality, availability and integrity. Both security measures and threats fall into one of these categories. To consider a distributed system as secure, it has to be ready for possible problems in each of these categories.

2.1 Confidentiality

Confidentiality is everything that is related to accessing to information by users (whether this access is authorised or not). Based on attempts to break confidentiality of distributed systems, the main attacks are traffic analysis, eavesdropping, spoofing, unauthorised access, MitM (man in the middle) and re-injection attacks [33]. Security measures to prevent these attacks try to ensure that only authorised users can access data and services. In the same way, it should be guaranteed that users with access do not constitute a new threat. It must also be checked that a user with appropriate permissions can access the system and not be mistakenly identified as dangerous. The main security measures used to ensure confidentiality are: authentication, authorisation, audit, data flow control/analysis and encryption.

2.1.1 Authentication

The process of ensuring that a user is who they are supposed to be is called *Authentication* and it is essential in any distributed system. This process must be conducted before allowing someone to access system resources. Typically the user provides a password, signature or, in general, some information (a *token*) that identifies them as who they claim to be. There are three main types of authentication in distributed systems [50].

(1) *Source authentication.* It involves verifying that the information source and author are the same. To verify the identity of the sender, techniques based on *Public Key Infrastructure (PKI)* [12] such as digital signatures are typically used.
(2) *Content authentication.* It involves verifying that the information received is the same as the information sent. As in the previous case, keys are often used, but in this case they are used to encode the message content, concretely a hash code obtained from the content is encoded.
(3) *Identity authentication.* In this case, we must verify that someone is who they claim to be. For identity verification, we use *authentication protocols* such as *Kerberos* [29], for the exchange of credentials. These credentials are often in the form of "username" and "password".

The main attacks trying to break these types of security measure are based on exploiting weaknesses in the protocols used, since nowadays if we use keys

properly, there is not enough computing power to decode them using *dictionary attacks*, *brute force* or *crackers*.

2.1.2 Authorisation

A very common requirement for distributed systems security is the provision of different levels of access to resources. This is known as *Authorisation*. Each identified user has a level of access, based on security policies, that deny or allow access to resources. The most widely used security mechanisms in distributed systems to ensure authorisation are [3]:

(1) *Access control lists*. Each user has a list of resources that can be accessed and the operations that can be performed on them. This mechanism is similar to that used in many operating systems.
(2) *Roles*. The system has a number of roles, and different roles are able to access different resources. Each user belongs to a role, and only if a role is authorised to access a resource may the user with that role do so. This system is widely used in *database management system (DBMS)*. In the case of distributed systems often there is no central server role. Instead point to point connections are used and users need to transmit their roles to access resources. This can be very dangerous if encrypted connections are not used.
(3) *Mandatory access control*. In this case, both users and resources have an access level. If the user level is high enough then the user can access to the resource. This security system is commonly used in military systems or in high security systems.

To bypass these security measures, malicious users use techniques of *privilege escalation*. Users with access to the system gain greater privileges to gain access to more restricted resources. The most common attack used to break these measures is the *buffer overrun* attack. This attack consists of taking control of a node by sending it a greater amount of information than it is capable of processing, in order to cause an unchecked buffer to overflow. To avoid this, some intermediate components can be introduced to divide the information or limit the size that the source can send with respect to the recipient [23].

2.1.3 Audit

All the aforementioned security measures are aimed at preventing possible attacks against a system. Audit and monitoring are focused on detecting anomalies in system security [51]. One way to achieve this is through *Intrusion Detection*, which involves two tasks: monitoring activities and processing the results to search for evidences indicating whether a system has been attacked

or not. Usually any access to a system or to its resources is registered like other events that occur in the system. It is principal to log failures in access, services and communications. This way, if many failed attempts to access the system are registered, it may be possible to deduce that someone is trying to access the system in an unauthorised way. Should this be established, appropriate counter-measures can be taken. However, registering and storing all of this information can penalise system performance. Registering every single event can make the system to spend more time managing audit data than performing functional work.

Attacks related to auditing try to obtain the information stored in the log. This information is often critical (it can relate to users, resources, addresses, dates or times) and it could be used by attackers to break other security measures. Therefore, the audit records must be stored with maximum security.

2.1.4 Data flow

In distributed systems, the network topology is not pre-defined and it is common to use intermediate nodes to exchange information between sender and receiver (an approach sometimes referred to as multi-hop routing [7]). However, this potentially introduces the problem that each intermediate node may be able to access information passing through it, even if that information is address to a different node. In this case, it is necessary to protect the information from intermediate nodes by restricting their access. One common action taken in these cases is to use different authorisation levels for nodes [27]. Besides, a security policy with traffic restrictions can be established to prevent nodes from receiving information that they shouldn't have access to. The main disadvantage for this solution is that the topology of the network must be known at any time, a condition that may involve an important computational cost. Every time a node is introduced, removed or reordered, a re-analysis of the topology and update of the security policies should be performed. Distributed systems like CORBA solve these problems by establishing restrictions on data flow, taking into account only the relationships between pairs of nodes, ignoring their role within the larger system or structure that they are part of [22]. However, due to these restrictions there may be cuts in the flow, impeding the movement of information from reach its destination.

The main solution to avoid these problems is to carry out an efficient network analysis [24] and a subsequent reorganization of traffic through the nodes in the network [52].

2.1.5 Encryption

The purpose of encryption of communications is to transform information into data that is unreadable to unauthorized users. The most common way to achieve this is through the use of complex key-based algorithms for both encryption and decryption of messages [28]. With exiting techniques, if the key or algorithm are unknown, it is extremely difficult to decrypt messages [49]. A *brute force* attack is one of the main methods used to break encryption, although having access to a large amount of computing power and time is required to make it effective. It is also common to use techniques to analyse messages, looking for patterns that allow the key to be deduced.

Due to the high effectiveness of these security techniques, measures to counteract these attacks are often more logical than technological. Keeping keys safe and changing them frequently are usually the most effective measures. The main disadvantage is that encryption is a complex process that may require some computing power and time, which in some scenarios, such as in mobile networks, may not be possible because the computational cost of encryption affects the battery consumption of the devices. In these cases, special intermediate nodes can be used, with greater processing power, for the purpose of encryption and decryption. The communication with these nodes must be done within controlled environments, while communication between nodes that encrypt and decrypt data can pass through hostile environments.

2.2 Availability

Availability has to do with the fact that both data and services should be available at any time. The main threats to the data are *SQL injection* attacks. These attacks attempt to execute (through services) malicious SQL statements to corrupt the data stored in the database. Although most *Database Management Systems (DBMS)* have mechanisms to prevent this type of threat, it is still advisable to protect services against this practice [10]. On the other hand, the main attacks against availability try to disrupt the proper working of services by flooding them with a lot of requests. These attacks are known as *Denial of Service (DoS)* attacks. Common measures used to protect against these attacks are *firewalls* and *load balancers* [4]. If one address is making too many requests in a short period of time, its access is blocked or it is redirected to another machine, avoiding the saturation of the service.

An issue related to availability is fault tolerance, which attempts to guard against problems such as power cuts, cuts in connections or hardware failures. Due to decentralization (point-to-point connections) and the characteristics of the devices (small mobile devices) of distributed systems, it is not always pos-

sible to apply the same security measures than in traditional systems: server mirrors, *uninterruptible power supplies (UPS)*, redundant paths or redundant hardware. Instead, techniques of reconfiguration for distributed systems might be used [26]. Thus, it is possible to modify the system behaviour on the nodes, featuring failures or cuts in communications. The reconfiguration of the system allows the "removal" of nodes with security problems, avoiding erroneous communication lines, and continue working without them.

2.3 Integrity

In distributed systems, as in any system that exchanges data, it is necessary to ensure that during transmission, the data reaches its destination without being manipulated. It should be assured the information is correct, valid and according to what the sender has sent. In this case, attacks may simply corrupt the message through interference to make the information unreadable to the recipient. Other more complex attacks alter the content of the messages without the recipient being aware that the information has been changed (MitM). In the same way, *non-repudiation* may be required, to ensure that a user cannot deny an action that they actually performed.

The most common techniques for these purposes are *message authentication codes (MAC)* (also referred to as *hash values*) and *digital signatures* based on public key infrastructure (PKI) [3].

2.4 Software approaches to implement security measures

As we have seen before, there are a variety of vulnerabilities that can compromise the security of distributed systems. But we also have diverse measures of detection and correction of threats. In general, all the security measures discussed above are based on adapting the system components. Taking into account how to do the adjustments, the following classification can be established:

- Rearranging the components. By modifying the topology of the system it is possible to prevent or to correct problems such as restrictions on the data flow or availability of data and services.
- Introducing new components (or removing some of the existing components). To avoid situations such as the *buffer overrun* or *DoS* attacks, it can be introduced *firewalls* or other components capable of solving these problems.
- Enclosing one or more of the components using a wrapper. To protect components against attacks or modify their behaviour appropriately, they can

be coated with new features to avoid dangerous situations such as adding encryption to communications or adding audit capabilities.
- Reconfiguring one or more of the components. In this case the components are reconfigured to suit the needs of the system, for example if there is a communication cut, we change the configuration of components to divert traffic so that availability will not be affected.
- Directly adapting the code. We directly modify the functionality of the components to adapt to problems. For example, adding to them a digital signature or access control information.

These adjustments can be made using different software development approaches. Due to the nature of distributed systems (heterogeneous devices, point-to-point connections, mobile devices, etc.) it should ideally be performed in a manner that is as flexible as possible: reusable and maintainable approach capable of adapting distributed systems at runtime.

- Reusable and maintainable: In software developments there are security issues that are orthogonal and independent to the main concern, with its code tangled and scattered in whole application. By separating the application functional code from these crosscutting concerns, the application source code would not be tangled, being easy to debug, understand, maintain and modify. Moreover, this high level of abstraction allows the system to reuse and share single concerns.
- Dynamic. The system should be able to adapt itself at runtime based on the dynamic environment. For example, it can use more or less security measures depending on the confidence of the environment or the capabilities of the component.

Under these assumptions, we propose the idea of applying *Aspect Oriented Software Development (AOSD)* in distributed systems security. As we will discuss in the following sections, with this approach it is possible to perform some of the adjustments mentioned above in a generic way without modifying the source code of the system and without stopping its execution, allowing the system to change its topology, capabilities and component numbers in a flexible and reusable way.

3 Aspect Oriented Software Development

Aspect Oriented Software Development (AOSD) [15] is a concrete approach to implement the principle of *Separation of Concerns (SoC)* [14]. AOSD facilitates a modularisation of different functionalities which cut across the entire system software.

An *aspect* is defined as any piece of software that cannot be encapsulated in a method or procedure, being scattered throughout the source code of an application. Common examples of aspects include transaction control, memory management, threading, persistence or logging [14].

In many cases, important functionalities in a system are not easily modularised. With the classic object oriented paradigm, the code that deals with these features is often dispersed in different parts of the application. AOSD manages the complexity of the software development by separating the functionality that is commonly tangled with the code of other features. The major benefits of this approach are the high level of abstraction, reuse of functionality, high legibility and improved software maintainability [14].

Figure 1 shows the structure of an aspect oriented program. The final application is the result of combining several modules. On one side there are the modules that contain the basic functionality and on the other hand the aspects with concrete functionalities that usually cut across the system functionality.

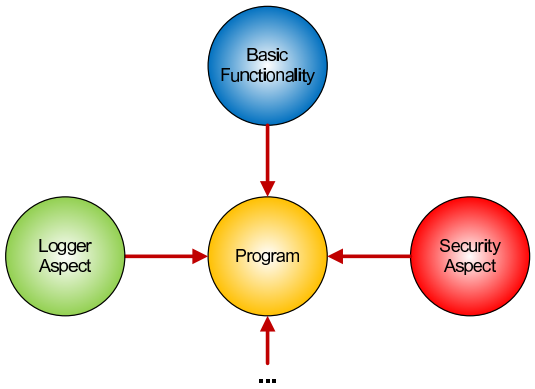

Fig. 1. Aspect oriented program structure.

In Figure 2, the difference between an object oriented program and an aspect oriented one is highlighted. Following the traditional paradigm, all of the code is mixed, making it difficult to maintain, debug and even comprehend. On the other hand, using aspect oriented development; each different functionality can be included in a separate module, facilitating its maintainability.

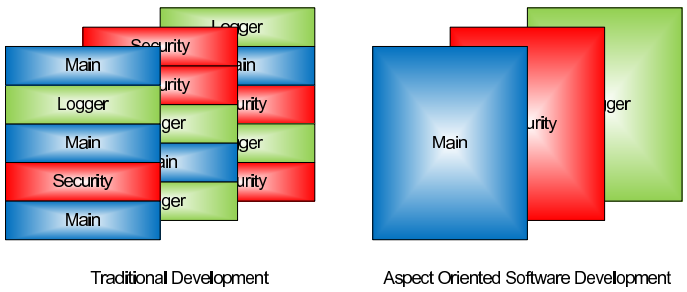

Fig. 2. Traditional development vs Aspect oriented development.

3.1 Weaving

Once both the basic functionality (components) and aspects are developed, it is necessary to take all the parts and form the whole program. This process is called *weaving*. Traditional processes to generate a program consist of passing the source code to a compiler, in order to obtain an executable. In AOSD all the code must pass through the compiler and should be treated by the weaver, to obtain the program with full functionality. This process can be performed statically (compile time or load time) or dynamically (at runtime).

To create the program there must be a relationship between the components and the aspects, i.e., the code of both has to interact with each other in some way. To allow this interaction, it is necessary to establish the points at which this interaction should occur. These points are called *join-points*. Join-points are those elements of the programming language semantics which the aspects coordinate with [15]. However, it is necessary to describe the mapping between join-points and aspects by means of *pointcuts*. A pointcut is a set of join-points plus, optionally, some of the values in the execution context of those join-points [19].

3.1.1 Static Weaving

The majority of existing AOSD implementations are based on static weaving. Static weaving consists on combining the aspects and components functionality prior to the application execution. This combination consists in inserting calls to advice in the components code. An advice is a method-like construct used to define additional behavior at join-points [19]. An advice is part of an aspect.

In AOSD, a program is developed providing a set of components implemented in a high-level programming language together with a set of aspects. There are two approaches to static weave components and aspects [43]: *Compile-time weaving*, which inserts the code belonging to the aspects into the code with basic functionality. The result is new source code, which is compiled generating an executable file. *Post-compile weaving*, (also called binary weaving) where components are first compiled and woven later.

This type of weaving causes little performance penalty because all the code is combined and statically optimized before its execution. Since the application is woven at compile time, any functionality required to be adapted at runtime requires stopping the application, recompiling and reweaving it. For this reason, this kind of weaving is not used in applications that require runtime adaptation. Dynamic weaving is used instead.

3.1.2 Dynamic Weaving

Although it is not always necessary, there are applications that need to be adapted at runtime in response to changes in the execution environment [38]. Examples include quality of service management in CORBA distributed systems [53] and cache systems management in web services [40]. The so-called *autonomic software* is another example; they are able to repair, manage, optimize or recover themselves [18]. By using a dynamic weaver, it is possible to modify the behaviour of a program at runtime. We can add, edit or remove functionality without stopping the application.

In the case of dynamic weaving the program is compiled in the traditional way and an executable is obtained. This program does not need to foresee which functionalities will be adapted. When the running program needs to be adapted, it is dynamically woven with new aspects adapting its behaviour. The original program continues unmodified, and hence it can be reused without the changes added in memory.

The main advantage of systems with this kind of weaving is that they support the dynamic adaptation of programs, maintaining the basic functionality separate from the aspects throughout the entirety of the software life cycle. Therefore, the resulting code is more adaptable and reusable, and both aspects and basic functionality can evolve independently [34]. The main disadvantage is that the dynamic adaptation commonly entails a runtime performance cost [6].

4 Dynamic and Static Aspect Weaving: DSAW

There are several tools that allow dynamic AOSD such as AOP/ST, PROSE, DAOP, JAC, CLAW, LOOM.NET, JAsCo or DSAW [48]. All these platforms allow developers to dynamically (un)weave aspects at runtime. But they also have some limitations [31]:

(1) Both dynamic and static weaving: Both kinds of weavers are supported in order to obtain better runtime performance, dynamic adaptiveness, and *edit-and-continue* interactive aspect-oriented development. Only DSAW offers full dynamism. However JBoss, Spring and LOOM.Net partially achieve it. In addition, the *Separation of the Dynamism Concern* implies the conversion of a static aspect into a dynamic one (and vice versa) without changing its implementation. Both JBoss and DSAW provide this feature. LOOM.Net and the Spring framework use different approaches for both kinds of weaving.
(2) Language neutrality. This feature implies the development of applica-

tions and aspects in any programming language. All the systems but LOOM.Net, Spring.Net and DSAW only support the Java programming language.

(3) Full dynamic weaving. Unlike many dynamic AOP approaches, unweaving and reweaving during runtime should be possible, even at join-points that were not woven before. String requires an XML file declaring advice and runtime aspect weaving and unweaving must be explicitly stated in the applications' source code. JBoss needs and XML file with pointcuts and advices that could be used at runtime. But once the application has been launched, aspects that were not specified in this XML file could not be woven. Although JAsCo, PROSE and DSAW offer a higher level of dynamism than the rest of systems, both JAsCo and PROSE show a limitation if re-weaving is required. If the aspect implementation is replaced by a new one, its new functionality is not reflected at runtime when the aspect is rewoven.

(4) Taking into account the {`constructor, method`}{`call,execution`} and `field`{`get, set`} join-points with the `before`, `after` and `around` times, AspectJ, DSAW Static and JBossAOP support all (18) and DSAW Dynamic almost all (16) (except {`constructor`}{`call,execution`} with `around` time). However, PROSE, JAsCo, Spring Java, Spring.Net have a small subset (3 or 4).

If what is required is a system able to adapt itself at runtime to functionalities not foreseen at design-time (and unweave aspects at runtime), only JAsCo, PROSE and DSAW offer this functionality.

Because of its language-neutrality, high number of join-points, and the support of both dynamic and static weaving, we selected the DSAW platform to develop our proposal. DSAW (Dynamic and Static Aspect Weaving) [47, 31] is an aspect oriented software development platform that supports homogeneous static and dynamic weaving. Its main features are:

(1) Platform independence: It is designed over the .NET virtual machine reference standard [8], without modifying or extending the semantics of the platform. This ensures complete independence from the platform, allowing the deployment of this system on any .NET implementation (for example Mono, SSCLI or DotGNU).

(2) Language independence: DSAW performs the adaptation of software at the level of the Intermediate Language (IL) of the virtual machine (executable files and libraries). This means that at weave-time the application source code is not necessary, and the platform provides language independence.

(3) Weave-time independence: This platform not only allows static and dynamic weaving, but can also take advantage of the same implementation of the components and aspects, independently of the scenario chosen.

The original application does not need to be modified in order to be injected with the aspects, and no aspect needs to be modified if the time of injection is changed from static to dynamic, or vice versa.

(4) Wide range of join-points: DSAW offers a wide and flexible set of join-points that enable it to adapt any significant point of a given application. Moreover, unlike most tools with both kinds of weaving, DSAW offers the same set of join-points for the two scenarios. DSAW supports the following join-points: method and constructor execution, method and constructor calls, and field and property reads and writes. Each one allows the specification before, after and around times.

4.1 DSAW in action

Both components and aspects can be developed in any .NET programming language. They can even be developed by third-party entities, since it is not necessary to have their source code. The components do not need to implement any special interface or feature to allow aspect weaving. Figure 3 shows the steps that occur in the development of an application in DSAW.

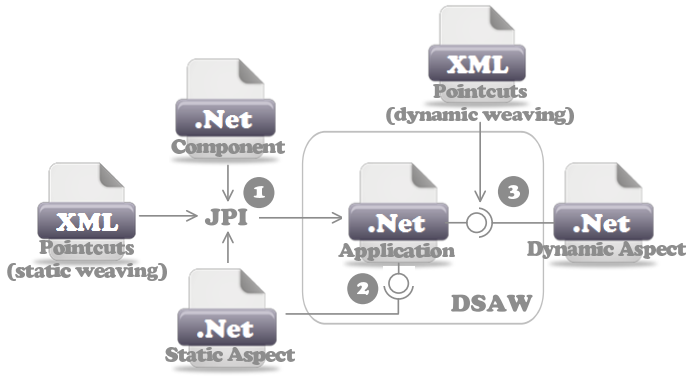

Fig. 3. Dynamic weaving steps.

Any existing .Net application, regardless of the high-level programming language that have be used to develop it can be adapted by DSAW. The Join-Point Injector (JPI) takes the application binary code (assembly) and, prior to its execution, performs instrumentation of the code in memory. If the weaving is static, a pointcut specification file must be passed as a parameter. In that case, the application is modified with calls to specific methods of the appropriate aspect specified in the static pointcut specification file.

Figure 4 shows an example of a pointcut specification. In this case two pointcuts are specified: **before executes** the `SendMessage` method of **any** class, the `AddAuthorisationLevel` method of the `AccessControlAndDataFlowAspect` class of the `AccessControlAndDataFlow.dll` assembly must be executed, and **after executes** the `ReceiveMessage` method of **any** class, the `VerifyAutho-`

`risationLevel` method of the `AccessControlAndDataFlowAspect` class of the `AccessControlAndDataFlow.dll` assembly must be executed too. If a XML is not passed to perform static weaving, all of the join-points will be available at runtime to weave dynamically.

```
<aspect_definitions>
 <pointcut_definition id="c1">
  <time>before</time>
  <joinpoint_type>
   <methodexecution>
    <method_signature>
     <return_type><type_name>*</type_name></return_type>
     <qualified_method_name>
      <qualified_class>
       <namespace><type_name>*</type_name></namespace>
       <class><identifier_name>*</identifier_name></class>
      </qualified_class>
      <name><identifier_name>SendMessage</identifier_name></name>
     </qualified_method_name>
    </method_signature>
   </methodexecution>
  </joinpoint_type>
 </pointcut_definition>

 <pointcut_definition id="c2">
  <time>after</time>
  <joinpoint_type>
   <methodexecution>
    <method_signature>
     <return_type><type_name>*</type_name></return_type>
     <qualified_method_name>
      <qualified_class>
       <namespace><type_name>*</type_name></namespace>
       <class><identifier_name>*</identifier_name></class>
      </qualified_class>
      <name><identifier_name>ReceiveMessage</identifier_name></name>
     </qualified_method_name>
    </method_signature>
   </methodexecution>
  </joinpoint_type>
 </pointcut_definition>

 <advice_definition idTypeOfInjection="StaticInjection">
  <assembly>AccessControlAndDataFlow.dll</assembly>
  <type>AccessControlAndDataFlowAspect</type>
  <behaviour>AddAuthorisationLevel</behaviour>
  <pointcut_definitionRef idRef="c1"/>
 </advice_definition>

 <advice_definition idTypeOfInjection="StaticInjection">
  <assembly>AccessControlAndDataFlow.dll</assembly>
  <type>AccessControlAndDataFlowAspect</type>
  <behaviour>VerifyAuthorisationLevel</behaviour>
  <pointcut_definitionRef idRef="c2"/>
 </advice_definition>
</aspect_definitions>
```

Fig. 4. Pointcut specification.

In case dynamic weaving is required, the JPI instruments the application with more code. Since it cannot be known in which join-points the developer of a dynamic aspect could be interested in, the JPI instruments the application so that any join-point can be intercepted at runtime.

```
<aspect_definitions>
 <pointcut_definition id="c1">
  <time>around</time>
  <joinpoint_type>
   <get>
    <field_type><type_name>*</type_name></field_type>
    <qualified_field_name>
     <qualified_class>
      <namespace><type_name>*</type_name></namespace>
      <class><identifier_name>*</identifier_name></class>
     </qualified_class>
     <name><identifier_name>swSender</identifier_name></name>
    </qualified_field_name>
   </get>
  </joinpoint_type>
 </pointcut_definition>

 <pointcut_definition id="c2">
  <time>around</time>
  <joinpoint_type>
   <get>
    <field_type><type_name>*</type_name></field_type>
    <qualified_field_name>
     <qualified_class>
      <namespace><type_name>*</type_name></namespace>
      <class><identifier_name>*</identifier_name></class>
     </qualified_class>
     <name><identifier_name>srReceiver</identifier_name></name>
    </qualified_field_name>
   </get>
  </joinpoint_type>
 </pointcut_definition>

 <advice_definition idTypeOfInjection="DynamicInjection">
  <assembly>ChangeCommunication.dll</assembly>
  <type>ChangeCommunicationAspect</type>
  <behaviour>ChangeCommunication</behaviour>
  <pointcut_definitionRef idRef="c1"/>
  <pointcut_definitionRef idRef="c2"/>
 </advice_definition>
</aspect_definitions>
```

Fig. 5. Pointcut specification.

At runtime, a new aspect could be required to be woven. For this purpose it is necessary to specify the pointcut with an XML document. Figure 5 is an example of a pointcut specification file. In this case it is specified: **before calls** to the `Connect` method of the `Chat` class, the `ChangePort` method of the `ChangeConnectionPortAspect` class of the `Aspect.dll` assembly must be executed. Dynamic weaving is performed by copying the aspect and XML pointcut description file into the application execution directory. At this time, the code added to the application by the JPI makes the application to dynamically invoke to aspect code when the execution reaches the point indicated. In this way, the behaviour of the application is dynamically modified, because when the execution of the application reaches the specified point the code indicated in the pointcut will be called. To unweave the aspect, it is only necessary to remove the file and the aspect that previously passed.

5 Use cases

The main objective of this work is to show how aspect oriented development may be an aid to develop flexible security mechanisms in distributed systems. In order to do that, we have developed in DSAW security mechanisms for two specific scenarios: communications encryption and access control / data flow

5.1 Encryption

As mentioned in Section 2.1.5, communications encryption is a security measure which aim objective is to prevent unauthorised users understand the information that is exchanged between the nodes of the system. This is typically done using algorithms that transform data into unreadable messages. The problem is that the encryption process can be quite heavy, because it is necessary to encrypt data before sending it and decrypting it by the recipient. In distributed systems composed by mobile devices, this process is rather costly and has a greater impact on battery consumption. To prevent overloading of the devices, it is very usual to use specific nodes of the system, with greater computing power, to encrypt and decrypt the information. A sample scenario could be various mobile devices exchanging data through a public insecure Wi-Fi network. If two nodes need to exchange confidential information, they can use a more secure UMTS [11] connection. This channel, although slower, has a higher level of security. Once the transmission of this data finished, the mobile devices can reuse the original Wi-Fi connection.

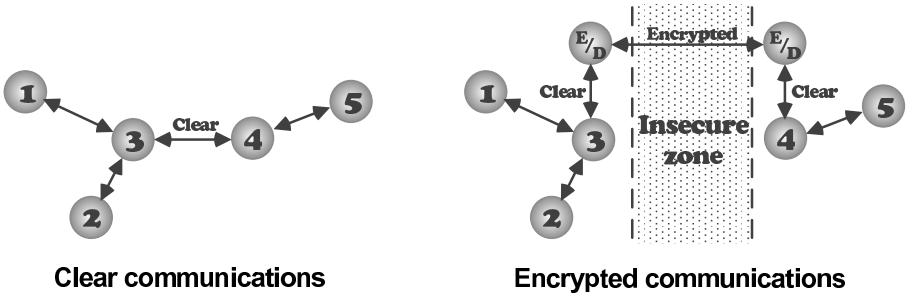

Fig. 6. Communications with and without encryption.

Figure 6 shows the difference between the two scenarios. In the first scenario there is a direct communication between nodes 3 and 4. In the second one, since there is an insecure zone between these two nodes, the communication is passed through two E/D (encrypt/decrypt) nodes. The advantage of the second scenario is that devices are freed of this work, but a delay between sending and receiving messages is introduced. It is necessary to send information to these special E/D nodes, encrypt the information, exchange it between them, decode and deliver it to the destination node. To overcome this delay, it is

better to use encryption only when it is necessary. When communications are likely to be heard (if we suspect or detect that there are intruders listening to the network), it is recommended to use encryption. But if the channel is in a controlled and trusted environment, the encryption is not required as this would introduce an unnecessary overhead and delay in the message transmissions.

We have developed this scenario where, under these circumstances, the system is able to dynamically change communications in a distributed system. Thus, when information must be encrypted, transmissions are forwarded to the nodes that encrypt and decrypt it (right part in Figure 6), and when it is not necessary, the original communication routes are used (left side in Figure 6). Using dynamic weaving it is possible to dynamically change the behaviour of nodes 3 and 4. To change the communication ways, the aspect be injected needs to establish a new connection between the source node and a E/D node. Afterwards, it is necessary to divert the traffic through this new connection. Finally when the aspect is unwoven, the original communication must be re-established.

The connection with the encryption and decryption E/D nodes takes place when the aspects are woven. At runtime, the nodes 3 and 4 are connected to an E/D node, and these are connected to each other. In this way, as we show in Figure 7, a new secure connection between nodes 3 and 4 is established.

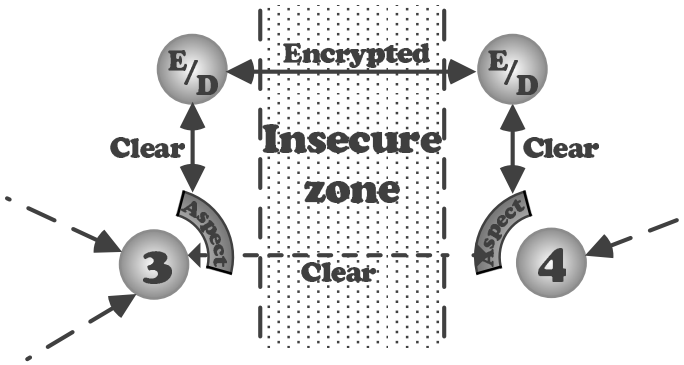

Fig. 7. Encrypted communications using aspects.

Once connected, using the same aspects, the functionality of the nodes 3 and 4 is altered in order to forward the information. In this way, all traffic is diverted through the secure connection, keeping the node unaware of that fact. Then, when nodes 3 and 4 send information, it will go to its corresponding E/D node, which encrypts and retransmits it to another E/D node. Once received by the second E/D node, the data is decoded and transmitted to the destination node using a clear communication.

The disconnection takes place when the aspects are unwoven. Then, the original communication is automatically re-established. Since the aspect does not

interfere when the nodes access their connection, the information will be sent as it originally was.

```
public class Chat {
 private TcpClient connection;
 private StreamWriter swSender;
 public StreamWriter SwSender{
  set {swSender = value;}
  get {
   if(swSender == null)
    swSender = new StreamWriter(connection.GetStream());
   return swSender;
  }
 }
 private StreamReader srReceiver;
 public StreamReader SrReceiver {
  set {srReceiver = value;}
  get {
   if(srReceiver == null)
    srReceiver = new StreamReader(connection.GetStream());
   return srReceiver;
  }
 }
 private Data ReceiveMessage() {
  String received = SrReceiver.ReadLine();
  Data message = Deserialize(received);
  return message;
 }
 private void SendMessage(Data message) {
  String toSend = Serialize(message);
  SwSender.WriteLine(toSend);
  SwSender.Flush();
 }
}
```

Fig. 8. Sample of C# code for sending and receiving messages.

The code to be modified is the one that is in charge of carrying the transmissions. Figure 8 shows a simple example of C# code that receives and sends data over a network. Attribute `connection` is the connection used to exchange information with another node. The `swSender` and `srReceiver` attributes are the streams responsible for sending and receiving information through the connection. `SwSender` and `SrReceiver` are the properties created to encapsulate the access to these attributes. The `authorisationLevel` attribute is irrelevant in this scenario, it will use in the next one. The `ReceiveMessage` and `SendMessage` methods are responsible for sending and receiving messages. To send and receive information over a connection, data has to be serialised and deserialised.

As discussed above, we want that whenever the connection is used for both sending and receiving data in an unsecure environment, an aspect should intervene and replace the connection by a more secure one. To achieve this, two around join-points are intercepted in the `SwSender` and `SrReceiver` read properties. Figure 5 shows the pointcut document describing these join-points. The `ChangeCommunication` method of the `ChangeCommunicationAspect` class is woven at both join-points. Thus, when the properties are read, aspect code

is run instead.

```
public class ChangeCommunicationAspect {
 private static TcpClient connection = Connect();
 private static StreamWriter swSender;
 private static StreamReader srReceiver;
 private static TcpClient Connect() {
  TcpClient connection = new TcpClient();
  connection.Connect(IP_ED, PORT_ED);
  swSender = new StreamWriter(connection.GetStream());
  srReceiver = new StreamReader(connection.GetStream());
  return connection;
 }
 public static object ChangeCommunication(String member ...) {
  if (member.Equals("swSender")) {
   return swSender;
  }
  if (member.Equals("srReceiver")) {
   return srReceiver;
  }
 }
}
```

Fig. 9. C# code of the aspect that modifies the connection.

Figure 9 shows the code of the aspect that modifies the connection. When the aspect is woven, a new connection with a E/D node is created (using an predefined *ip* and *port*). The E/D nodes must be out of the insecure zone, as shown in Figure 7. These nodes encrypt, decrypt and exchange data between each other through the insecure zone. Once that process achieved, when the `ChangeCommunication` method is invoked, the aspects verify which of the properties is being called (this information is passed as the `member` parameter to the aspect by the DSAW platform) and return the corresponding secure stream in each case.

Therefore, if the pointcut specification document is passed to the platform, the behaviour of the nodes is dynamically changed, as when the `ReceiveMessage` and `SendMessage` are invoked and the properties are used to access to the streams, the code of the aspect returns a secure connection.

With this approach, when the encryption of the communications at runtime is necessary, it is enough to pass to DSAW the file with the pointcut definition and the aspect. Finally, when we want to stop the encryption, it is only necessary to remove both from the platform to recover the original behaviour.

5.2 Access control/Data Flow

As we have previously stated (Section 2.1.4) distributed systems do not have a defined network topology and is very usual that the information has to pass through intermediate nodes to travel through the network. If any node has rights to see the information that travels through the system, this is not a

problem. However, there may be nodes that have no privilege to see certain information, causing a security problem. In that case, it is necessary to restrict the access to the information, in order to prevent that nodes without the appropriate permission may have access to unauthorised data. *Access control* (Section 2.1.1) establishes who can access to the information at any time, and data flow (Section 2.1) determines how information flows across the network.

Traditional mechanisms as CORBA [22], solve these problems considering only "one-to-one" relationships, without taking into account that the devices are part of a distributed system. This solution is effective in client-server networks or with fixed topologies. However, in point-to-point networks or with fuzzy topologies, the usage of these mechanisms may imply restrictions on the data flow. If a node is not authorised to send data to another one, but the latter is the only way to get to a third one, data transmission to the third node is not possible, even without a specific restriction on the system to do it. Other solutions analyse network topology in order to identify potential problems in the distribution of the information and to establish restrictions on access and data flow. To do this, they use security policies with different levels of authorisation for the nodes. This kind of security policies are widely used in military systems or catastrophes management, where the access control to information is essential[27]. Nodes can only send information to those nodes that have an authorisation level greater or equal to theirs, but they need to know the authorisation levels for all nodes they are connected to. Figure 10 shows an example of this scenario.

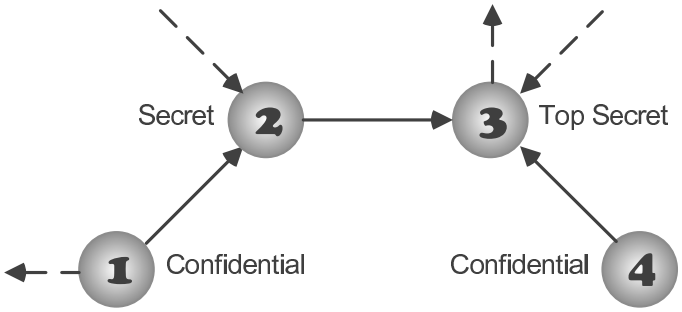

Fig. 10. Distributed system with different authorisation levels.

Nodes 1 and 4 may send information to any other node because the `confidential` level is the lowest. Node 2 can only send information to node 3, since `secret` is less restrictive than `top secret`. Finally, node 3 cannot send information to anyone because its level is the highest. The problem is that approach lack flexibility; whenever the network change its number or topology, it is necessary to reanalyse the whole system and update the policies with the new restrictions. As in the previous case, in diffuse topologies, this solution may be very costly, and data flow restrictions can arise, preventing two nodes from exchanging information. For example, nodes 1 and 4 in Figure 10 have the same access level, but they cannot exchange information because node 3

cannot relay messages.

In distributed systems scenarios, these kind of solutions are difficult to implement, mainly because they are very flexible networks, where the number and topology change frequently. Therefore, flexible security mechanisms are required. It should be allowed to control the access to the information at any time without interfering with the flow, regardless of the number of nodes and shape of the system.

We have used the static weaver of DSAW to implement a distributed system with a security policy that guarantees the secure transmission of information over changing topologies, allowing the tagging data with the authorisation levels of nodes and encrypting the information. Applications are built relying on the classical *send* and *receive* operations, and aspects intercept these messages to include the following functionalities:

(1) Authentication to grant the user the appropriate authorisation level.
(2) Encryption of information to avoid unauthorised access to it.
(3) Data tagging to determinate how information flows across the network and to control the access to it.

When a node sends the information, an aspect encrypts and labels it with the authorisation level of the source node. At destination, another aspect receives and decrypts the information and verifies whether the authorisation level of the node is high enough to access the data. If the level suffices, the aspect passes the data to the node; otherwise, it is discarded. With this mechanism, the information travels encrypted and labelled with the authorisation level in a totally transparent way to the system.

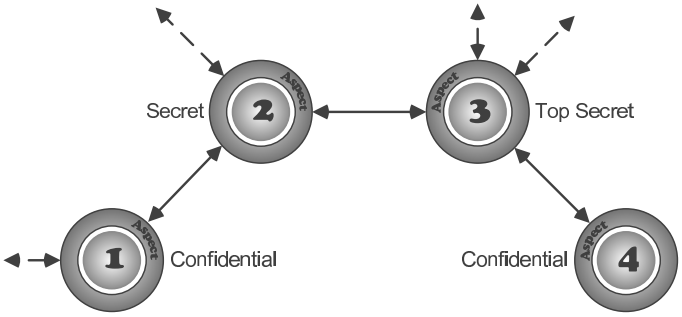

Fig. 11. Aspects controlling the access to information.

Although access is controlled, the problem in the data flow has not been avoided. To solve it, the aspects take into account information of the source node. When an intermediate node receives data to relay, it is sent with the authorisation level of the source node and not with its own information. As we show in Figure 11, with this approach all nodes can send data to any node, regardless of the authorisation level that they hold, because the aspects restrict the access and control the flow. The node 1 can now send data to node

4 through nodes 2 and 3. If an intermediate node alters the data in any way, it will automatically have the same level of authorisation and the aspects will restrict its dissemination.

```
public class AccessControlAndDataFlowAspect {
 private static DataWrapper receivedData;
 private int AuthorisationLevel() {
   //... Method that obtains the authorisation level of the node.
 }
 private DataWrapper Encrypt(DataWrapper data) {
   //... Method that encrypts a DataWrapper.
 }
 private DataWrapper Decrypt(DataWrapper data) {
   //... Method that decrypts a DataWrapper.
 }
 public static object AddAuthorisationLevel(object ResultVal, Param[] par,
     ...) {
  int nodeAuthorisationLevel = AuthorisationLevel();
  Data dataToSend = (Data) par[0];
  if (receivedData != null && receivedData.Data.Equals(dataToSend))
   nodeAuthorisationLevel = receivedData.AuthorisationLevel;
  receivedData = null;
  DataWrapper dataWrapper = new DataWrapper(dataToSend,
      nodeAuthorisationLevel);
  par[0] = Encrypt(dataWrapper);
  return 1;
 }
 public static object VerifyAuthorisationLevel(object ResultVal, Param[] par,
      ...) {
  int nodeAuthorisationLevel = AuthorisationLevel();
  DataWrapper dataWrapper = (DataWrapper)ResultVal;
  receivedData = Decrypt(dataWrapper);
  if (nodeAuthoritationLevel >= receivedData.AuthoritationLevel)
   ResultVal = receivedData.Data
  else
   ResultVal = new Data();
  return 1;
 }
}
```

Fig. 12. C# code of the aspect to tag and verifies the data.

To implement this solution, it is necessary to determine the join-points used in the implementation of this distributed system. Using the DSAW platform, we can use its static weaver to modify the values of the parameters of a method just before of his execution and the result just after that it finish. In this way we can introduce data tagging and encrypt when we send the information, and decrypt and verify the authorisation levels when we receive they. Following the code in Figure 8 the ideal places to do this are the `SendMessage` and `ReceiveMessage` methods. In Figure 4 we can see the pointcuts used to do this implementation. The `AddAuthorisationLevel` method is invoked in order to encrypt and introduce data tagging. We also call the `VerifyAuthorisationLevel` method, which will be responsible for decrypting and verifying that the authorisation level of the node allows it to view data. Figure 12 shows the code of the aspect used to develop our solution. In this aspect, using the `AddAuthorisationLevel` method, the data is labelled with the authorisation level of the code just before encrypting and sending it.

Attention must be paid to the fact that whether an intermediate node does not include (or change) new information, the authorisation level held will be the one from the source node, not the one of this intermediate one.

The first step is to obtain the authorisation level of the node. Using the `AuthorisationLevel` function we can obtain the authorisation level of the node. Then, if the node has previously received some data and they are equal to those who it goes to send (the node will relay a message) the authorisation level to use is the one of the received data. To label the data with the authorisation level, the value of the parameter `par[0]` that is the `message` parameter of the `SendMessage` method (it is provided by the DSAW platform) is changed for a wrapper composed by the data and the obtained authorisation level. Figure 13 shows the wrapper code. The `DataWrapper` class inherits from the `Data` class, making the `SendMessage` and `ReceiveMessage` methods run without problem, even if its argument is changed. Finally, the `dataWrapper` is encrypted.

```
public class DataWrapper : Data {
 private AuthorisationLevel authorisationLevel;
 private Data data;
 public AuthorisationLevel AuthorisationLevel {
  get { return authorisationLevel; }
  set { authorisationLevel = value; }
 }
 public Data Data {
  get { return data; }
  set { data = value; }
 }
 public DataWrapper(Data data, AuthorisationLevel authorisationLevel) {
  Data = data;
  AuthorisationLevel = authorisationLevel;
 }
}
```

Fig. 13. C# code of the wrapper used to introduce data tagging.

When the aspect receives the data, the `VerifyAuthorisationLevel` method decrypts the data and verifies that the authorisation level of the node is sufficient for accessing the received data –otherwise it discards the data. As in the previous case, it is necessary to obtain the authorisation level of the node. However, it is also necessary to extract the authorisation level of the data using the `ResultVal` (provided as a parameter by the platform) that is the result of the `ReceiveMessage` method execution. Levels are checked; if the level is enough, the method extracts the original data from the wrapper and returns it; otherwise, the data is discarded.

This way, we modify the behavior of the nodes, when they send and receive messages. Before sending the data, it is labeled; and before accessing it, the permissions for reading it are checked. Using this approach, the distributed system can vary in number and topology without compromising safety, since the access control and the data flow is controlled at all times, and there is no

need to analyse the network or update the policies. The entire process works in a transparent way with respect to both the nodes and the system. The functionality of the application is modularised apart from the communication and security concerns, implemented as separate aspects.

6 Evaluation

In this section we evaluate the utilisation of DSAW to develop security measures, comparing it with a traditional object-oriented programming (OOP) implementation. Our experimental methodology is outlined first. Afterwards, benefits of using DSAW are stated and quantitative evaluations are performed. Finally, we present a discussion regarding to the evaluation obtained.

6.1 Methodology

We highlight what are the benefits of dynamic separation of aspects in distributed systems security using DSAW. The benefits are those mentioned throughout the paper (detailed in Section 6.2). Quantitative characteristics are runtime performance and memory consumption.

Quantitative assessment can be used to contrast runtime performance and memory consumption between the DSAW platform and traditional OOP developments. We have implemented a distributed system with the security measures of the two uses cases (Section 5) using DSAW and OOP. These implementations have been compared using the .Net Framework 2.0 build 50727 for 32 bits, over a Windows 7 x64 operating system. All tests have been carried out on a lightly loaded 2.13GHz Intel Core 2 Duo system with 4GB of RAM. We developed all software in C# programming language.

In order to compare DSAW with OOP, we have created the networks of the two use cases with each implemented distributed system. In the first case (Section 5.1), the distributed system is composed by two single nodes and two E/D nodes. In the second one (Section 5.2), the system has three nodes. To evaluate runtime performance, we have instrumented the code with hooks to record the value of the processor timestamp counter. We have measured the difference in the value between the beginning and the end of exchange a set of messages to obtain the total execution time of each system. To suppress the cost of native code generation by the JIT compiler, we first make a single use of the distributed system exchanging a little set of messages between the nodes. This first invocation is not taken into account in our evaluation. Therefore, this assessment ignores the time required to dynamically generate native code

by the virtual machine JIT compiler.

All the tests have been executed utilizing the Windows 7 performance monitor. We have measured the maximum size of working set memory used by the process since it started (the `PeakWorkingSet` property). The working set of a process is the number of memory pages currently visible to the process in physical RAM memory. These pages are resident and available for an application to use without triggering a page fault. The working set includes both shared and private data. The shared data comprises the pages that contain all the instructions that the process executes, including instructions from the process modules and the system libraries.

6.2 Benefits of using DSAW

The following are the main obtained benefits of using the DSAW platform to apply dynamic separation of aspects to distributed systems security.

(1) **Higher abstraction level**. Developers can be focused in concrete concerns in isolation. The security measures may be developed by experts in the domain, apart of other components of the distributed system.
(2) **Reuse, maintenance and legibility**. Separation of concerns attains decoupling of different modules, allowing the distributed system to reuse and share single security concerns. The code is not tangled and scattered throughout the whole application. The code of each security issue is not coupled to the rest of the code. Since the code is less complex, it is more easier to understand and maintain.
(3) **Increase of application development productivity**. In addition to previously mentioned advantages, the use of security concerns (such as encryption or control access) might facilitate the security measures construction without needing to modify the functional source code.
(4) **Dynamic adaptation**. It is possible to adapt distributed systems depending on the runtime environment. For example, if a better performance is needed, it is possible disable some security measure such as encryption. If a new threat appears at runtime, it is possible to dynamically adapt distributed systems in order to include new security measure to solve it. This way, security measures always take into account the necessities of the system.
(5) **Platform and language independence**. DSAW is language and platform neutral. Thereby, it is possible to develop components and aspects in any .Net language and they can be executed in any .Net implementation. This is an interesting feature for heterogeneous distributed systems, composed by elements with different operating systems and software. Security measures may be developed in different .Net languages and deployed

throughout the system.

(6) **Weave-time concern**. With DSAW the implementation of the components and aspects is independent of the weave-time concern. Neither applications nor aspects are modified if the weaver needs to be changed from static to dynamic, or vice versa. The original application and aspects do not need to be modified if the time of injection is changed from static to dynamic, or vice versa. This way, the same security measure may be established as dynamic (to be disabled or enabled at runtime) or static (obtaining a better performance).

(7) **Join-points**. Unlike most tools with both kinds of weaving, DSAW offers the same set of join-points for the two scenarios. In addition to previously mentioned advantage, it is possible to use the same security measures.

(8) **Source code**. DSAW performs the adaptation at virtual machine level, the application and aspects source code it is not necessary. Thus, it is possible to adapt legacy distributed systems or inject security measures developed by others.

6.3 Quantitative evaluation

To obtain an evaluation of runtime performance and memory consumption, we have assessed the cost of using static and dynamic weaving in DSAW. For both use cases (Section 5), we have developed an distributed system using the traditional object-oriented programming paradigm. In both cases, using OOP, we have extended the developed system adding the following security measures: encryption (in the first use case), and access control and data flow (in the second one).

In the encryption scenario, we have used dynamic weaving to inject the aspect in the original developed system, and in the control access and data flow use case static weaving was used. Thus, we have compared the perform and memory for OOP and AOSD approaches. For each scenario, we have sent and received a set of messages between the distributed system nodes. The first call has not been included to rule out the cost of JIT compilation. Figure 14 shows the results of the four scenarios: Encryption {OOP vs. DSAW Dynamic} and Access Control/Data flow {OOP vs. DSAW Static}; execution time (columns one to five) of a set of messages expressed in milliseconds and memory consumption (last column) in bytes. Memory consumption is the same regardless of the number of messages exchanged.

Encryption	50.000	100.000	150.000	300.000	500.000	Memory
OOP	5.759,3849	11.913,9499	18.380,3240	36.433,3603	60.382,4187	29.704.192
DSAW Dynamic	9.067,1341	18.785,5415	29.101,5819	58.587,5371	97.688,9838	46.325.760

Access Control/Data flow	1.000	3.000	6.000	10.000	15.000	Memory
OOP	3.565,8766	12.028,2405	24.764,5187	41.405,2752	62.324,2203	30.199.808
DSAW Static	3.825,7490	12.476,4282	25.525,4981	42.808,5518	64.285,3317	30.625.792

Fig. 14. Execution time and memory consumption.

6.3.1 Discussion

A summary of the results is displayed in Figure 15. All values are shown relative to OOP implementation (values were divided by the values of OOP). First major discussion could be identified. It is related to the cost of weaving. In the dynamic weaving case, aspects involved a 59.2% and 55.96% cost of runtime performance and memory consumption respectively. For the control access/data flow scenario, we used static weaving. In this case, the runtime performance penalty was 4.12% and the increase of memory consumption was 1.41%. In both scenarios, performance costs did not depend on the number of messages sent and received. The standard derivation was 6,72% for the static scenario and 2,82% for the dynamic one.

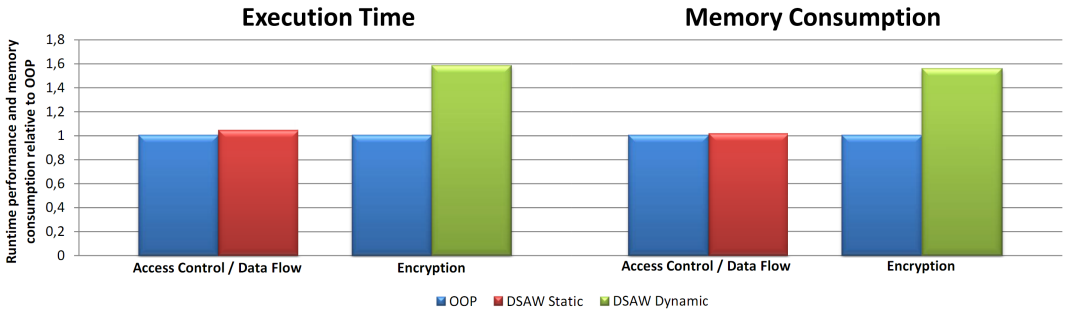

Fig. 15. Execution time and memory consumption relative to OOP.

In order to analyse the runtime performance penalty, we used a profiler obtaining the results shown in the Table 16. The first column is the percentage of the total execution time of the OOP implementation. The second one shows the performance increase of the AOSD approach relative to the OOP one. We measure the following parts of `SendMessage` and `ReceiveMessage` methods, because it is where the distributed system is adapted. 1) JPI: this value has been obtained by adding the execution time of all methods of the DSAW platform. 2) Method call: is the execution time required to call the send and receive method. 3) Method execution: is the time required to execute the code into the send and receive methods. This value is obtained by subtracting the whole execution time of the method call and the previous execution time value.

In the encryption scenario, the JPI imply a 43.4% of the total cost. The dynamic execution call to the `ChangeCommunication` produces 53,79% of the

Encryption

	OOP	DSAW penalty
Send		
JPI	0,00%	17,03%
Method calls	0,00%	20,70%
Method execution	2,88%	0,38%
Receive		
JPI	0,00%	8,66%
Method calls	0,00%	11,14%
Method execution	1,28%	0,34%

Access Control / Data Flow

	OOP	DSAW penalty
Send		
JPI	0,00%	0,56%
Method calls	0,35%	1,73%
Method execution	0,12%	0,02%
Serialize	48,92%	1,27%
Receive		
JPI	0,00%	0,56%
Method calls	0,43%	0,67%
Method execution	0,27%	0,02%
Deserialize	42,34%	0,33%

Fig. 16. Detailed runtime performance penalty.

penalty. In both send and receive, this is the part where the greatest increases occur (20,7% y 11,14%). As this is a dynamic scenario, DSAW uses reflective techniques to call the method of the aspect [31]. In the OOP implementation, it takes almost no time because it is only necessary to access the `SwSender` and `SrReceiver` properties. Finally, the increase of the method execution of the aspects regarding the original application execution time represents around 1% of the total cost.

The second table of the figure details the access control and data flow scenario. In this case, the JPI increases are only 0.56% for both sending and receiving. The `AddAuthorisationLevel` and `VerifyAuthorisationLevel` method call represent 58.3% of the total penalty, although the increases are 1,73% and 0,67% respectively. As in previous case, the performance penalty in the aspect execution is only 1%. Finally, we have added the `Serialize` and `Deserialize` methods to the analysis because there are significant increases, 1,27% and 0,33% respectively. These penalties represent around 40% of the total cost, and they are motivated because in the AOSD implementation is necessary to serialise and deserialise `DataWrapper` objects instead of `Data` ones.

In the dynamic scenario, the JPI and method calls are 97,19% of the total cost. The evaluations with static weaving are 85,5%. However, the dynamic penalty is significantly higher than the static: 57,53% versus 3,52%.

Regarding memory consumption, in the static weaving scenario it is around 1%. However, in the dynamic case the increase is 55.96 %. The DSAW dynamic weaver injects a join-points table to activate the join-points in the application that causes this memory increase [31].

7 Related Work

7.1 Existing AOSD Platforms

In the last years, AOSD significantly evolved. Nowadays, it is frequent to use aspects in order to implement concerns such as logging, tracing, security checks, testing or persistence [39]. AspectJ [20] is considered the most widely used platform for aspect-oriented programming in Java. An example use is carrying out efficient profiling of programs, allowing to evaluate issues such as heap usage, object lifetime, wasted time or time-spent to improve the performance of applications [32]. Other application areas are transaction management and persistence [41]. JBoss AOP [16] and Spring AOP [17] are other Java frameworks for AOP, which are used in the context of the JBoss application server and Spring application framework, respectively. The PROSE [37], JAsCo [42] and DAOP [34] platforms, using its dynamic features, are used for application adaptation at runtime [30, 35, 45]. LOOM.NET is based on *design by contract* using any .Net language [21].

7.2 AOSD in Security Issues

There are various studies and works that implement and introduce security mechanisms using AOSD. In [46] an extension of the C programming language to support aspects is proposed. This extension allows the definition of security policies apart of the application code. Subsequently, the policies are transformed into code, which is woven together with the main application. Using this approach, automatic security checks can be done, including protection against buffer overflow, logging or encrypted connection. When new threats are detected, the security policies are updated with the most suitable measures. Although this work is presented as an extension of the C programming language, the authors state that it might be applied to other languages.

AspectJ has also been used in application security issues. In [13] a generic and reusable library to introduce security mechanisms in Java developments is presented. However, there seems to be some lacks in AspectJ that are needed to enforce security issues successfully [2]. For example, AspectJ does not provide pointcuts to local variables defined inside methods. Security debuggers may need to track the values of local variables inside methods. With such new pointcuts, it will be easy to write advice before or after the use of these variables to expose their values.

In the network area, there are works that propose the utilisation of aspects to solve security issues. In [25] the authors present a language framework called

D. This framework untangles the implementation of synchronization schemes and remote data transfers from the implementation of the components using aspects. And in [44] a reflective architecture is proposed. This architecture allows making dynamic reconfiguration of non-functional requirements such as real-time, reliability, availability and security. For each non-functional requirement a specific AOP language is defined.

As far as the authors are aware, PROSE is the only dynamic AOSD platform used for security issues [36]. This platform was proposed to make dynamic adaptation of software architectures, where functionality is a dynamic property and applications can be adapted depending on the runtime environment. This way, it is possible to integrate specific concerns (e.g. access control) in software for existing distributed systems such as *spontaneous* networks [9]. But, this is only a preliminary work that it is not detailed.

8 Conclusions

This paper analyses how dynamic and static AOSD can be used to solve common security issues of distributed systems. Distributed systems have vulnerabilities that jeopardise their safety. We have developed a classification of security issues that can arise in distributed systems. The majority of listed problems can be solved by adapting the system components. In this paper we propose the use of *Aspect Oriented Software Development* to implement these adaptations. Flexible security mechanisms in distributed systems can be implemented with AOSD, obtaining higher maintainability, reusability and dynamic adaptability –aspects can be (un)woven without stopping the system. Thus, distributed systems can vary in number and topology without compromising safety. We have used the DSAW platform to implement two scenarios of applying AOSD to security issues of distributed systems: one scenario of communications encryption and another one of access control and data flow. Both solutions are introduced to the system in a transparently way, and they are able to solve the problems without interfering with the system. Depending on the needs of the system, the solutions may be introduced or removed at runtime.

The assessment of runtime performance has shown that real applications developed in DSAW have entailed a performance cost of 4.12% and 59.2%, respectively comparing static and dynamic weaving with the traditional object-oriented development.

The immediately future work will be focused on using this approach into other real distributed system security use cases, and in more long-term to apply this techniques to a "systems-of-systems" [1].

Current documentation and implementation of this work can be freely downloaded from its Web page at `http://www.reflection.uniovi.es/dowload/2010/ietsw`

9 Acknowledgments

This work has been funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation; project TIN2008-00276 entitled *Improving Performance and Robustness of Dynamic Languages to develop Efficient, Scalable and Reliable Software.*

References

[1] R.L. Ackoff. Towards a system of systems concepts. *Management Science*, 17(11):661–671, 1971.

[2] D. AlHadidi, N. Belblidia, and M. Debbabi. Security crosscutting concerns and AspectJ. In *Proceedings of the 2006 International Conference on Privacy, Security and Trust: Bridge the Gap Between PST Technologies and Business Services, October*.

[3] A. Belapurkar, A. Chakrabarti, H. Ponnapalli, N. Varadarajan, S. Padmanabhuni, and S. Sundarrajan. *Distributed Systems Security: Issues, Processes and Solutions*. Wiley, 2009.

[4] J. Bellardo and S. Savage. 802.11 denial-of-service attacks: Real vulnerabilities and practical solutions. In *Proceedings of the 12th conference on USENIX Security Symposium-Volume 12*, page 2. USENIX Association, 2003.

[5] M. Bishop. *Introduction to computer security*. Addison-Wesley Professional, 2004.

[6] K. Böllert. On weaving aspects. In *Proceedings of the Workshop on Object-Oriented Technology*, page 302. Springer-Verlag, 1999.

[7] J. Broch, D.A. Maltz, D.B. Johnson, Y.C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, page 97. ACM, 1998.

[8] T.C. Ecma. TG3. Common Language Infrastructure (CLI). Standard ECMA-335, 2005.

[9] L.M. Feeney, B. Ahlgren, A. Westerlund, et al. Spontaneous networking: an application oriented approach to ad hoc networking. *IEEE Communications Magazine*, 39(6):176–181, 2001.

[10] WG Halfond, J. Viegas, and A. Orso. A classification of SQL-injection

attacks and countermeasures. In *Intl Symp. on Secure Software Engineering*. Citeseer.

[11] H. Holma, A. Toskala, et al. *WCDMA for UMTS: Radio access for third generation mobile communications*. Citeseer, 2000.

[12] R. Housley, W. Ford, and W. Polk. D. Solo," Internet X. 509 Public Key Infrastructure Certificate and CRL Profile, 1999.

[13] M. Huang, C. Wang, and L. Zhang. Toward a reusable and generic security aspect library. *AOSD: AOSDSEC*, 4, 2004.

[14] W. Hürsch and C.V. Lopes. Separation of concerns. *Northeastern University, February*, 1995.

[15] J. Irwin, G. Kickzales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, and J. Loingtier. Aspect-oriented programming. *Proceedings of ECOOP, IEEE, Finland*, pages 220–242, 1997.

[16] Jboss AOP homepage. JBoss Community. http://labs.jboss.com/jbossaop/.

[17] R. Johnson, J. Hoeller, A. Arendsen, C. Sampaleanu, D. Davison, D. Kopylenko, T. Risberg, M. Pollack, and R. Harro. Spring–Java/J2EE Application Framework. *Reference Documentation, Version*, 1(7).

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

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. *ECOOP 2001Object-Oriented Programming*, pages 327–354, 2001.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[21] K. Köhne, W. Schult, and A. Polze. Design by contract in .NET using aspect oriented programming. 2008.

[22] U. Lang and R. Schreiner. *Developing secure distributed systems with CORBA*. Artech House Publishers, 2002.

[23] D. Llewellyn-Jones, M. Merabti, Q. Shi, and B. Askwith. Buffer overrun prevention through component composition analysis. In *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, pages 156–163, 2005.

[24] D. Llewellyn-Jones, M. Merabti, Q. Shi, and B. Askwith. Analysis and Detection of Access Violations in Componentised Systems. In *2nd Conference on Advances in Computer Security and Forensics (ACSF 2007), Liverpool, UK*, pages 12–13. Citeseer, 2007.

[25] CV Lopes and G. Kiczales. D: A language framework for distributed computing. *Xerox PARC, TR SPL97-010 P*, 9710047.

[26] H. Moiin and A. Pruscino. Data integrity and availability in a distributed computer system, February 20 2001. US Patent 6,192,483.

[27] National Computer Security Center NCSC. Trusted network interpretation environments guideline, 1990.

[28] R.M. Needham and M.D. Schroeder. Using encryption for authentication

in large networks of computers. *Communications of the ACM*, 21(12):999, 1978.

[29] B.C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, 1994.

[30] A. Nicoara and G. Alonso. Making Applications Persistent at Run-time. In *IEEE 23rd International Conference on Data Engineering, 2007. ICDE 2007*, pages 1368–1372, 2007.

[31] F Ortin, L Vinuesa, and J.M. Felix. The DSAW Aspect-Oriented Software Development Platform. *International Journal of Software Engineering and Knowledge Engineering*, To be published.

[32] D.J. Pearce, M. Webster, R. Berry, and P.H.J. Kelly. Profiling with aspectj. *Software: Practice and Experience*, 37(7):747–777, 2007.

[33] C.P. Pfleeger and S.L. Pfleeger. *Security in computing*. Prentice Hall, 2003.

[34] M. Pinto, M. Amor, L. Fuentes, and JM Troya. Run-time coordination of components: Design patterns vs. componentaspect based platforms. In *ASoC workshop (Advanced Separation of Concerns)*, pages 18–22. Citeseer, 2001.

[35] M. Pinto, D. Jiménez, and L. Fuentes. Developing dynamic and adaptable applications with CAM/DAOP: A virtual office application. In *Generative Programming and Component Engineering*, pages 438–441. Springer, 2005.

[36] A. Popovici, T. Gross, and G. Alonso. Aop support for mobile systems. *Paper at the OOPSLA*, 1.

[37] A. Popovici, T. Gross, and G. Alonso. Dynamic homogenous AOP with PROSE. *Switerland, Department of Computer Science, ETH Zürich*, 2001.

[38] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *In Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 141–147. ACM Press, 2002.

[39] VO Safonov and DA Grigoryev. Aspect .NET: aspect-oriented programming for Microsoft .NET in practice. *NET Developers Journal*, 7, 2005.

[40] M. Ségura-Devillechaise, J.M. Menaud, G. Muller, and J.L. Lawall. Web cache prefetching as an aspect: towards a dynamic-weaving based solution. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, page 119. ACM, 2003.

[41] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 174–190. ACM, 2002.

[42] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29. ACM, 2003.

[43] A.J. Team. The AspectJ 5 development kit developers notebook (2004).

[44] E. Truyen, B.N. Jørgensen, and W. Joosen. Customization of component-based Object Request Brokers through dynamic reconfiguration. In *Technology of Object-Oriented Languages and Systems–TOOLS*, volume 33, pages 181–194. Citeseer.

[45] W. Vanderperren, D. Suvée, B. Verheecke, M.A. Cibrán, and V. Jonckers. Adaptive programming in JAsCo. In *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 75–86. ACM, 2005.

[46] J. Viega, JT Bloch, and P. Chandra. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14(2):31–39, 2001.

[47] L. Vinuesa and F. Ortin. A Dynamic Aspect Weaver over the. NET Platform. *Metainformatics*, pages 197–212, 2004.

[48] Luis Vinuesa, Francisco Ortín, José M. Félix, and Fernando Álvarez. Dsaw - a dynamic and static aspect weaving platform. In *ICSOFT (PL/DPS/KE)*, pages 55–62. INSTICC Press, 2008.

[49] Prof Steve Wilbur and D Cn. Distributed systems security, 2000.

[50] T.Y.C. Woo and S.S. Lam. Authentication for distributed systems. *Computer*, 25(1):39–52, 1992.

[51] Y. Zhang and W. Lee. Intrusion detection in wireless ad-hoc networks. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, page 283. ACM, 2000.

[52] Z. Zhao and W. Li. Dynamic reconfiguration of distributed data flow systems. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 2, 2007.

[53] J.A. Zinky, D.E. Bakken, and R.E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1):55–73, 1997.