AO-OpenCom: An AO-Middleware Architecture supporting flexible Dynamic Reconfiguration

Bholanathsingh Surajbali
Smart Research and Development
CAS Software AG
Karlsruhe, Germany
b.surajbali@cas.de

Paul Grace
IT Innovation
University of Southampton
Southampton, UK
pjg@it-innovation.soton.ac.uk

Geoff Coulson
School of Computing
Lancaster University
Lancaster, UK
geoff@comp.lancs.ac.uk

ABSTRACT

Middleware has emerged as a key technology in the construction of distributed systems. As a consequence, middleware is increasingly required to be highly modular and configurable, to support separation of concerns between services, and, crucially, to support *dynamic reconfiguration*: i.e. to be capable of being changed while running. Aspect-oriented middleware is a promising technology for the realisation of distributed reconfiguration in distributed systems. In this paper we propose an aspect-oriented middleware platform called AO-OpenCom that builds AO-based reconfiguration on top of a dynamic component approach to middleware system composition. The goal is to support extremely flexible dynamic reconfiguration that can be applied at all levels of the system and uniformly across the distributed environment. We evaluate our platform by the capability in meeting flexible reconfiguration and the impact of these overheads.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems – Distributed applications; D.2.11 [**Software Engineering**]: Software Architectures – Doman-specific architectures; D2.13 [Software engineering]: Reusable Software – Reusable libraries

Keywords

Aspect Oriented; Middleware; Dynamic Reconfiguration

1. INTRODUCTION

Dynamic reconfiguration in aspect-oriented (AO) based middleware is very promising, but under-developed. AOP addresses these two problems by encapsulating logically independent pieces of functionality into separate modules known as *aspects*. The aspects are then *woven* into the system (this weaving process can be performed at compile-time, load-time or runtime) to build the required behaviour. An aspect defines both *behaviour* and *composition logic*, the latter describing both where and when the behaviour is executed. The compositional logical associated with an aspect is often referred to as a *pointcut*. The points in a program at which composition occurs, as directed by a pointcut, are referred to as *join points*. The declarative approach of aspect-oriented programming (AOP) is of considerable help to the developer in terms of facilitating the description and enactment of dynamic reconfiguration. However, most current AOP middleware systems [3, 4, 7, 10, 11, 12, 15] are evolutions of earlier systems that lack reconfiguration *flexibility* and are focused primarily on *local* reconfiguration with limited dynamic reconfiguration capabilities. The lack of flexibility of AO-middleware can be categorised in terms of *five* different dynamic reconfiguration variability of distributed system such as granular scope reconfigurability, vertical scope, horizontal scope, performance and resource overhead. First, granular scope of reconfigurability is important because it defines the extent to which reconfiguration can be applied and can be classified in terms of fine-grained and coarse-grained reconfiguration support. Coarse-grained composition allows entire system functionality to be added or removed, while fine-grained composition relates to smaller changes (e.g. changing protocols from Wi-Fi to Bluetooth technology when there is a drop in power and vice-versa [5]). Second, vertical scope is an important issue because many systems allow only application level reconfigurability and this is insufficient in many cases, where system infrastructures need to change. For example, reconfiguration may also be required at the infrastructure level to add new functionality such as the support for new group communication or apply updates such as correcting anomalies from existing infrastructure services. Third, horizontal scope reconfigurability is crucial as both local and distributed nodes should be reconfigurable to ensure consistent view of the middleware service. Finally, performance and resource overhead are important criteria because if a system is highly reconfigurable but it runs too slowly or consumes too much resource it is not acceptable. In particular the platform must allow aspects to be dynamically woven as needed and unwoven when no longer necessary. This decreases the resource overhead and performance of invoking aspects at a join point.

In this paper, we present a novel *dynamically reconfigurable AO-middleware architecture, AO-OpenCom* providing a principled way of dynamic reconfiguration with degrees of flexibility that go beyond the state of the art by following a component-based, reflection and AOP design approach. In particular, the architecture will address the *five* main areas of deficiency as discussed above and the design is evaluated by its flexibility and expressiveness in specifying a range of types of dynamic reconfiguration. In particular, the platform offers four flexible distributed reconfiguration operation in terms of support for: i) local pointcut – local advice; ii) local pointcut- remote aspect; iii), remote pointcut – local aspect; and iv) remote pointcut and remote aspect.

The remainder of this paper is structured as follows. First, Section 2 presents the aspect composition model of AO-OpenCom. Next, Section 3 presents the concepts and implementation of the AO-OpenCom middleware architecture which is then evaluated in Section 5. Then, Section 6 provides a discussion on how the AO-Opencom meets the requirements based on the experimental results, as well as an analysis against related work. Finally, Section 7 draws concluding remarks.

2. AO-CONNECTOR MODEL

Aspects in AO-OpenCom are composed with the base components (hereafter termed components) within component interfaces/ receptacles using connectors. The AO-OpenCom connector model have two variants: the Default-Connector which contains the direct reference of a receptacle to an interface of components (no aspectual composition); and the AO-Connector is the architectural element offering aspectual composition (weaving) of aspects between a receptacle and a provided interface of components.

2.1 Interface/Receptacle AO-Connector

The runtime composition of aspects using an interface/receptacle AO-Connector is achieved using a proxy that redirects the call or execution through the chain of advices on the interface or receptacle as illustrated in Figure 1. A key benefit of the interface/receptacle aspect composition approach is that it allows aspects to be composed even when no binding is present.

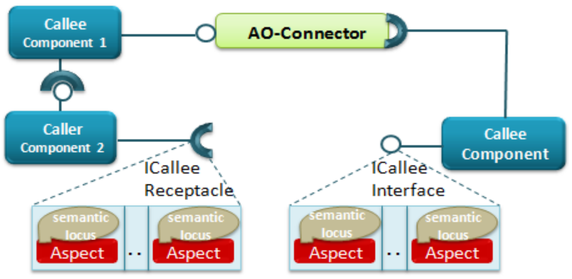


Figure 1. Aspect Composition execution chain with Semantic Locus using Interface/Receptacle AO-Connector

2.2 AO-Connector Binder Composition

The AO-Connector composition differs from component to component connectors by maintaining the metadata containing references to aspects instances in an advice chain. For example, it maintains details of all advised aspects and their types and allows these to be queried to determine the operations they support and the aspects currently advising them. It also supports the runtime manipulation of the chain to add new advices, or remove or reorder aspects in the chain of advices. Figure 2 shows the AO-Connector connecting two components, the Caller component receptacle to the Callee component provided interface, containing the advice chain with advice$_1$ to advice$_n$. The AO-Connector also supports the inspection and reconfiguration of the woven aspects in the advice chain. The inspection mechanism allows type checking of the aspect before it is woven in the advice chain. Furthermore, the introspection capability allows the detection of conflicts between the hosting aspects or the CF they belong to.

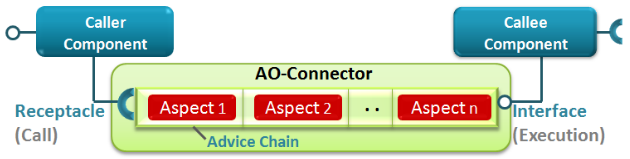


Figure 2. Aspect Composition execution chain with Semantic Locus at AO-Connector

2.3 Local Aspect Composition

Each AO-Connector is responsible for generating at runtime the appropriate advice chain for the set of possible join points that can occur at its bound interfaces. Figure 3 illustrates a local AO composition whereby the Callee Component has a *methodCallee()* method attached to the AO-Connector aspects with the respective locus semantics. It should be noted that the around advice is different to that used in AspectJ. The order follows the similar semantics as in DyMAC [7] and AspectOpenCom [4], that is:

> *before* advices or *around$_{before}$* advices are executed in the order in which they are encountered within the chain followed by the *method execution*, then followed by *after* advices or *around$_{after}$* advices in the order in which they are encountered within the chain.

For example from Figure 3, a call from the Caller component to the Callee component, results in the aspect chain to be invoked in the following order:

[before0 from Aspect A$_1$] → [around $_{before}$ from Aspect A$_2$] → [before1 from Aspect A$_4$] → [before2 from Aspect A$_5$] → [methodCallee()]→ [after0 from Aspect A$_3$] → [around $_{after}$ from Aspect A$_2$].

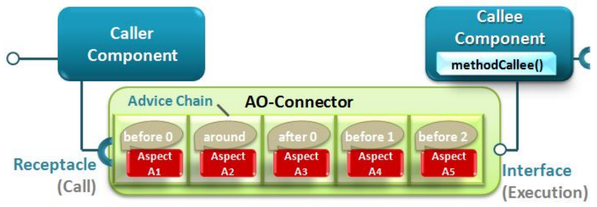


Figure 3. Aspect Composition execution chain

In the absence of the locus semantics, then each interface (execution join point) and receptacle (call join point) call gets redirected through a chain of advices attached through the AO-Connector as illustrated in Figure 4. When a call takes place from the Caller Component the execution follows the following order:

[Aspect 1] → [Aspect 2] → [Aspect 3] → [methodCallee()]

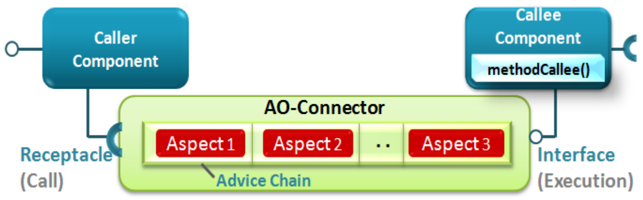


Figure 4. Aspect Composition without Semantic Locus

2.4 Remote Aspect Composition

Aspects can also specify and implement remote method invocations and are classified in terms of remote advices. These aspects can be used to provide distributed remote AO compositions. Similar to local aspect composition, remote aspects support the three locus semantics (before, after and around) of aspect execution and their references can be attached to the AO-Connector. A key difference between remote and local advice is that remote advices implement the Serialisable mechanism, e.g. Java Serialisable interface such that their method calls and return values can be used on remote CFs. A remote aspect is advisable like any other method invocation, and can capture both call and execution of components.

3. AO-OpenCom

AO-OpenCom is an extension of the OpenCom component model and its associated reflective meta-models and component framework CF architectures. The extension introduces a novel AO-meta framework layer on top of the existing underlying component-based reflective middleware substrate. This approach follows from the need of the AO-meta framework to cover the crosscutting functionality, and to ensure it preserves the separation concern [9]; that is, to ensure the middleware platform keeps separate views for better understanding and preservation of modularity within the middleware platform. The meta-layer and base-layer of each node crosscuts multiple address-spaces, and thus a top-level view provides separate viewpoint coverage of the crosscutting concerns across multiple address-spaces. This hides the complexity of the underlying reflective meta-layer from programmers. Moreover, the AO-meta framework is built as an independently-deployable service using components throughout the architecture (the AO-meta framework layer is constructed from components like the rest of the underlying reflective component-based middleware substrate). This means that the AO-meta framework can advise not only distributed applications, but also the underlying middleware services and the AO-meta framework layer itself. Being independent from the rest of the framework allows the AO-meta framework to be dynamically deployed when required and un-deployed when further reconfiguration is not required in the foreseeable future (thus avoiding any overhead when not in use). The AO-OpenCom architecture consists of the following main entities (see Figure 5): i) Base Framework; ii) Reflective-meta Framework; iii) AO-meta Framework; iv) Aspect Repository Framework; v) Distribution Framework; and vi) Configurator.

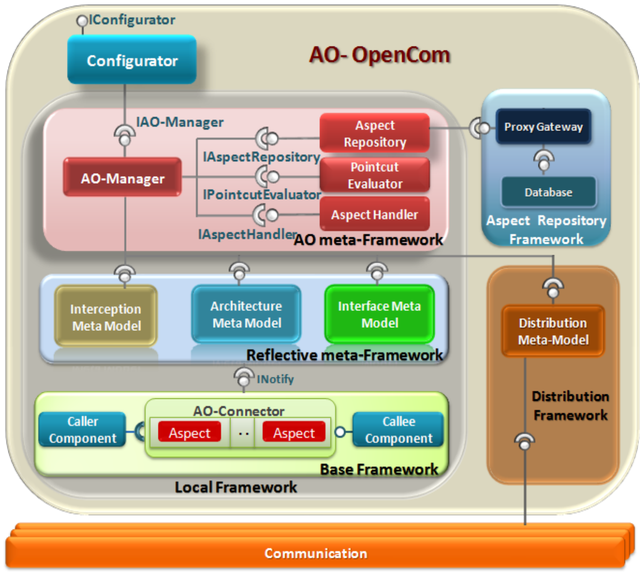


Figure 5. The AO-OpenCom Middleware

3.1 Base Framework

The base framework consists of the kernel which provides an API allowing new components and aspect-components together with the AO-Connectors to be instantiated, loaded, unloaded and destroyed. Furthermore, the kernel maintains a causal connection with the three meta-models in terms of the *distribution meta-model*, *AO meta-model* and the *reflective meta-model* such that any changes in the base runtime are reflected to the respective meta-model. Components, reflection and AO are at the core of this architecture, providing a principled approach to dynamic reconfiguration. At the base, *components* are encapsulated units of functionality and deployment that interact with other components exclusively through *interfaces* and *receptacles*. AO-Connectors represent the bindings between a single interface and a receptacle. Reflection technologies through the meta-models provide information about the current system state to inform about reconfiguration decisions; next a component-based approach allows the composition among components' provided and required interfaces. Next, AOP provides a declarative approach to support local and distributed quantification as well as local and remote aspect reconfiguration capabilities.

3.2 Reflective-meta Framework

The AO-OpenCom reflective-meta framework consists of the OpenCom reflective meta-models that provide the inspection, reconfiguration and extension of component and aspect-component composition with a local CF (e.g. of a component composition that represents a middleware platform instance). Each of the three meta-models can be optionally (and dynamically) deployed whenever required, and un-deployed when no longer required.

3.3 AO-meta Framework

The AO-meta framework internal architecture comprises a set of components that are instantiated into the host DCF. The set of components is as follows:

AO-Manager. This component is responsible for accepting and handling the configurator requests that will apply to the host DCF. Instances of the AO-Manager may run on more than one node in the DCF if desired. The AO-Manager interacts with Pointcut Evaluator and Aspect Handler components (see below) to perform the requested AO compositions within the DCF. It also caches join point information it receives from Pointcut Evaluators in case similar behaviour needs be applied in the future.

Aspect Repository. The Aspect Repository holds a set of instantiable aspect-components. Actually, the Aspect Repository is itself a (sub) CF which supports the configuration of repository functionality in a variety of ways. The sub-CF consists of a Front End component and a back-end Database component. This simple architecture enables a wide range of configurations; e.g. different Front Ends can apply different load balancing strategies and some Database components can be simple proxies to other Front Ends.

Pointcut Evaluator. The Pointcut Evaluator supports the parsing of pointcut expressions provided by the AO-Manager component and to return to the latter a list of all the matching join points found within the local address space. The supporting quantification by the Pointcut Evaluator component are DCF signatures, operation signatures, interface and receptacle signatures, component types and instances as well as dynamic properties of the CF runtime instances. An instance of the Pointcut Evaluator is present in each address space and consists of the following four sub-components:

- ***DCF Parser component,*** serves to evaluate DCF signatures.
- ***Expression Parser component***, evaluates component instances, types as well as interface and receptacle signatures.
- ***Method Parser component*** provides the parsing of method signatures of associated components.
- ***Dynamic Properties Parser component*** serves to parse the respective key, value pairs of the associated component types and instances in the runtime.
- ***Pointcut Matcher component*** compares the given pointcut specification against the corresponding identified runtime instances that are retrieved from the Distribution meta-model and Meta Architecture components.

Moreover, the Pointcut Evaluator component also supports the remote pointcut functionality to evaluate join points located in remote address spaces. To do so, it connects to the distribution framework to locate the appropriate join points in remote CFs. The inclusion of CFs in the list of quantifiable entities means that distribution is inherently supported in a network-independent manner. In addition, it can evaluate dynamic context properties associated for corresponding component instances that are stored in the AO-OpenCom runtime kernel and aspect-components by evaluating their dynamic properties from the Aspect Repository. Finally, the Pointcut Evaluator evaluates pointcuts and returns a list of matching join points within the local node in case it is a local pointcut, and for remote pointcuts, the lists of matching join points in each DCF.

Aspect Handler. This is also present in each address space. Its role is to act on instructions from the AO-Manager to weave advice at join points in its address space. The weaving is accomplished using the above-mentioned AO-Connector connector type which enables advices (i.e. an operation supported by an aspect) to be inserted between a pair of bound components. As well as purely address-space-local AO-Connectors, distributed AO-Connectors are also supported that can use distributed framework endpoints as well as local receptacle and interface pointers. This enables the AO-Connector to support the invocation of aspects that are resident in other address spaces. Finally, the Aspect Handler on receiving the join points and advice instances, performs advice weaving at specified join points in its CF, according to the advice specification.

3.4 Distribution Framework

The AO-OpenCom DCF architecture is illustrated in Figure 6 following a generic component approach to support various communication protocols (e.g. TCP, UDP, Multicast, JGroups, Broadcast, group protocols such as SCAMP [5]). Each CF maintains a basic architecture of the distribution framework, as well as a distribution meta-model containing contents of the CF as well as other instances of the DCF. The distribution module functions as a hub to the communication protocol choosing the desired communication protocol required to ensure reliable communication among the CFs'. For non-remote method invocation, based on the chosen communication protocol, the Distribution module component translates the outgoing messages using a message handler and then stores in the Queue component. The sender module extracts queued messages and sends them according to the outgoing protocol. On the receiving side the Receiver module component is responsible for receiving messages. Received messages are placed in the buffer component, which is then read by the Distribution module to update the distribution meta-model accordingly. Importantly, this communications service of the AO-OpenCom DCF is realised as a pluggable component, meaning that the service can replace the basic service (which is unreliable and does not support any ordering semantics) with a range of alternative communications services chosen according to the reliability and scalability requirements under which the DCF is deployed.

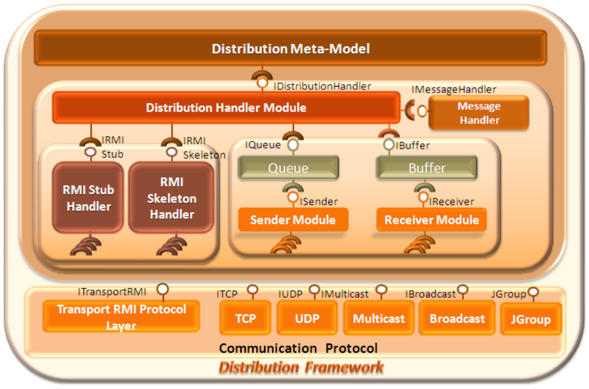


Figure 6. AO-OpenCom Distribution Framework

3.5 Configurator

Each Configurator interacts with its local Pointcut Evaluator and Advice Handler to carry out either a *reconfigure* or a *reorder* reconfiguration on its local node. The reconfigure operation provides the coarse-grained reconfiguration of aspects (add, remove and replace) at a join point and the reorder operation provides fine-grained reconfiguration, that is allowing aspects to be reordered at a join point. Consequently, AO-OpenCom provides granularity scope of reconfigurability requirements. Furthermore, the Configurator also interacts with other peer Configurators, for a distributed reconfiguration across multiple DCFs'. This allows the platform to provide for horizontal scope reconfigurability.

4. AO-OpenCom PROGRAMMING MODEL

4.1 AO-OpenCom Kernel API

The AO-OpenCom platform is supported using the minimal kernel API that is offered by each DCF. The intention of these operations is to provide the minimum functionality required to create instances and connect them. The key operations supported by the kernel API are shown in Figure 7 comprising of the eight main operations.

```
interface IAOOpenCom{
      //loads a new aspect / component and inserts it into the AO-Opencom runtime
      long load(String componentType);
      //creates a new instance of a component/aspect type
      long instantiate(long componentTypeId);
      //deletes a component/aspect instance which has been previously created in the
      runtime
      boolean destroy(long ComponentID);
      //connects a receptacle with an interface
      long connect(long ReceptacleId, long InterfaceId, AOConnector connectorType);
      //a particular property is inserted into the runtime registry
      boolean putprop(long entity, String key, Object value);
      //a particular property is retrieved from the runtime registry
      Object getprop(long entity, String key);
      //unloads an existing component/aspect type from the AO-OpenCom runtime
      boolean unload(long componentTypeId);
      //registers a callback to receive kernel events notifications
      void notifyCall(NotifyCallback c);
}
```

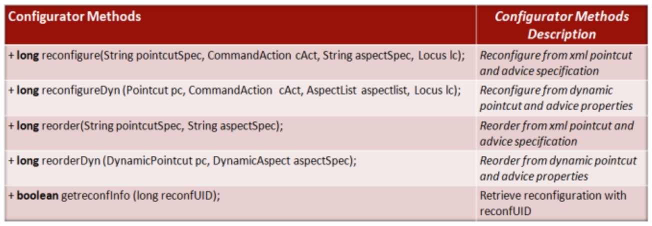
Figure 7. AO-OpenCom Kernel Base-level operations

The *load()* method loads a named component type from the component repository and aspects from the Aspect Repository, and the *unload()* method unloads an existing component type and aspect type from the runtime respectively. The *instantiate()* method provides the instantiation of aspects and component. Furthermore, a component or aspect can be instantiated multiple times if desired, with each having a different unique identifier. The *connect()* method connects a provided interface with a required interface of another. The *getprop()* returns a reference of associated entities (aspect types, component types, aspects, components, interfaces and receptacles) dynamic properties in the form of <name, value> tuples. The *putprop()* method writes into the registry the respective entities tuples. Finally, the *notifyCall()* method provides the callback operations whenever one of the methods in the kernel base-level system gets called causing call to be updated in the meta-model layers. It should be noted that the reconfiguration does not need to use the lower level AO-OpenCom base-level operations to reconfigure the platform. The base-level operations are available and can be used by the middleware developer. To perform any reconfiguration, the reconfiguration developer makes use of the configuration API that is described in the next section.

4.2 Configurator API

Aspect configuration and reconfiguration in AO-OpenCom can be specified in terms of an XML-based independent pointcut languages as well as programmatically which enables pointcuts signatures to be defined in terms of: i) capsules/hosts/DCF address expression; ii) a component type expression; iii) a component instance expression; iv) an interface/ receptacle type expression; v) a method type expression; and vi) metadata that can be attached to any of the foregoing. The main API provided by the AO-OpenCom enabled CF and DCF for AO composition and reconfiguration are shown in Table 1. The configuration specification containing the pointcut and advice specification is passed to AO-OpenCom Configurator. The configurator supports both programmatic and XML specifications to be sent to it following the BNF specification [14]. Note, that the Configurator API protects the reconfiguration developer from the low level details of actually managing and weaving the aspects among distributed nodes. Hence, facilitating the usability of the platform to support reconfiguration and to deploy new aspects, the programmer can use a deployment script similar to components deployment.

Table1. API AO-OpenCom Reconfiguration protocol



Configurator Methods	*Configurator Methods Description*
+ **long** reconfigure(String pointcutSpec, CommandAction cAct, String aspectSpec, Locus lc);	*Reconfigure from xml pointcut and advice specification*
+ **long** reconfigureDyn (Pointcut pc, CommandAction cAct, AspectList aspectlist, Locus lc);	*Reconfigure from dynamic pointcut and advice properties*
+ **long** reorder(String pointcutSpec, String aspectSpec);	*Reorder from xml pointcut and advice specification*
+ **long** reorderDyn (DynamicPointcut pc, DynamicAspect aspectSpec);	*Reorder from dynamic pointcut and advice properties*
+ **boolean** getreconfInfo (long reconfUID);	Retrieve reconfiguration with reconfUID

4.2.1 Local Reconfiguration

The local reconfiguration using the AO-OpenCom meta-layer is split into four main stages:

1. Stage I: Reconfiguration Setup

When a user issues a local reconfiguration request, the reconfiguration is initially handled by the Configurator component. This component forwards the aspect reconfiguration to the AO-Manager component. The latter, then checks if a similar reconfiguration request has not been performed. If a similar reconfiguration request is present, then the cached join point information is retrieved and reconfiguration proceeds to Stage IV (Aspect Weaving Stage). The AO-Manager maintains a time-out cache period, after which any cached reconfiguration from the AO-Repository is removed. Furthermore, using the base runtime notify() operation, any changes made to the runtime are notified to the AO-Manager, so that cached requests are removed. Additionally, after performing an update, the updated reconfiguration is cached by removing the old cached entry. If the reconfiguration is not present in the cached repository, then the AO-Manager component, first submits the pointcut specification to the Pointcut Evaluator component (Stage II) followed by submitting the advice specification and the join point (received from Stage III) to the Aspect Handler component.

2. Stage II : Join point Lookup

Next, in case the reconfiguration is a new request that is not present in the cache, the AO-Manager component submits to the Pointcut Evaluator component the pointcut specification in order to retrieve the required join points.

Stage IIa: Pointcut Specification Parsing. The Pointcut Evaluator first uses the Parser CFs components to parse each of the pointcut signatures and expressions. If the pointcut specification contains dynamic properties signatures, the Dynamic Property Parser component is used to parse and extract the associated pointcut expressions. Otherwise, the specification is parsed by the Expression Parser component to retrieve the appropriate names associated for the aspect/component and interface/receptacle pointcut signature and the Method Parser component to extract the associated operations signatures. The Parser CF then returns the parsed signatures back to the Pointcut Evaluator component.

Stage IIb: Join point Lookup. The Pointcut Evaluator then translates the respective parsed signatures/expressions whereby the aspects / component expressions are inspected using the Architecture meta-model component, followed by the interface and operation expressions. For the identified entities in the runtime, the respective connectors are returned to the Pointcut Evaluator component. In case the connector does not have an aspect then the default connector is returned to the AO-Manager component. Conversely, the AO-Connectors are returned to the AO-Manager in case of already woven aspects-components at the join points.

3. Stage III : Aspect Instance Retrieval

On receiving the list of connectors (either default-connector or AO-Connector), the AO-Manager component then submits the aspect specification and the connectors list to the Aspect Handler component. On the other hand, if the list of connectors is empty, the reconfiguration is not applicable and the reconfiguration is aborted. For *add* or *replace*, this may involve obtaining the aspect from an Aspect Repository. It will also involve weaving the aspect according to the specified scope and locus.

4. Stage IV: Aspect Weaving

The aspect weaving interaction varies according to the reconfiguration command- action.

1.) Add reconfiguration command-action.

Before the aspect weaving is performed, the AO-Handler ensures the aspects are type compatible by performing type-safety checking using the TypeValidator component. That is, it checks if the aspect exposes a matching interface and receptacle and its methods operations at the *callee* and *caller* components. Then, if at the join point there is no AO-Connector present, then the default-connector must first be replaced by an AO-Connector capable one. To do so, the Aspect Handler component, instructs the Quiescent Handler component to set the components under reconfiguration as well as the default connector to a quiescent state before the connector replacement is initiated. That is, the Aspect Handler component ensures that the associated components on the default connector are in a steady state before any reconfiguration can proceed.

For this purpose, a read/write lock mechanism is used, such that every non-reconfigure operation can access the lock as a reader (there can be *n* readers using the lock at any one time) and for any reconfiguration calls an exclusive writer lock is used. Once in a quiescent state, the Aspect Handler component then calls the AO-Connector-Factory (In the simplest form of bindings between components, the default-connector-factory component is used to instantiate connectors without any interception capability) component via the CF load() and instantiate() methods respectively. Subsequently, the interface-receptacle pair of the reconfigured components are connected by using the CF *connect()* method and parsing the instantiated AO-Connector-Factory factory as one of its arguments. Once created, a success message is returned to the AO-Manager component. In the case of failure the *fail_created_AOConnector_Timeout* failure message is sent to the AO-Manager. The error message signifies that the AO-Connector creation reached the reconfiguration timeout. If the AO-Connector has successfully been created, then the AO-Manager instructs the Aspect Handler to weave the aspects. By default for the *add* reconfiguration command-action the aspect is added in an ordered manner in the AO-Connector chain. If the order of the aspect-order is specified, then based on the specified order the aspect is woven in the AO-Connector chain.

2.) Replace reconfiguration command-action.

The reconfiguration to replace an aspect takes place in four interaction stages. Similar to the add command-action reconfiguration, the reconfiguration is first checked if they are type compatible with the interface-receptacles at the join point components, followed by placing the join point list of components and the AO-Connector aspect components to the quiescent state. Next, the Aspect Handler component extracts the execution state from the existing reconfigured aspect. This state extraction mechanism is optionally supported by aspects, the capability being dynamically discovered by the CF member using reflection. Then, the old aspect reference is removed and the new-aspect component reference added at the AO-Connector chain state is restored to the newly replaced aspect. If the old aspect state was extracted the state is restored to the newly updated aspect using the State Handler component *restore-state()* method operation.

3.) Remove reconfiguration command-action.
The remove reconfiguration command-action takes place in a three stage interaction, with the first stage consisting of setting the reconfiguration join point to quiescent. Then, in Stage II, the aspect reference is removed from the AO-Connector. In the final stage, Stage III the quiescence on the AO-Connector is removed, after completing the reconfiguration. Moreover, if there is no other aspect attached to the AO-Connector, then the AO-Connector is replaced by a default-connector. Once the weaving/un-weaving has been completed the Aspect Handler returns an acknowledgment message to the AO-Manager component. If the reconfiguration is successful the updated reconfiguration join point is cached. Finally, a reconfiguration *ack()* message is returned to the Configurator informing that the reconfiguration has been completed and the lock on the Configurator can be removed, such that the Configurator component can accept new reconfiguration requests.

4.) Reorder reconfiguration command-action.
The Aspect Reorder Reconfiguration involves the Stage I, II and III of the reconfiguration interaction. Stage IV is similar to the aspect removal reconfiguration action. However, instead of removing the aspect reference, the aspect references are reordered according to the specified advice specification.

4.2.2 Distributed Reconfiguration
The *Configurator.reconfigure()* reconfiguration protocol in AO-OpenCom is as follows:

1. *Configurator.reconfigure()* is called on the Configurator of one of the nodes supporting the DCF to be reconfigured; in the following this node is referred to as the 'initiator'.
2. The initiator determines how the specified aspect is to be applied. In the case of a per-DCF scope, it instantiates the aspect at a suitable node and sends a remote reference to the nodes where it is to be woven. Otherwise, the initiator decides if it has the specified aspect available locally (or can get it from an Aspect Repository) and wants to send it 'by value' to the nodes where it is to be woven, or if it wants to send the aspect 'by reference' and implicitly instruct the other DCF members to obtain the aspect from an Aspect Repository.
3. The initiator sends a 'reconfigure' message to all DCF member nodes. This essentially contains the parameters originally passed to *reconfigure()*. By default, the initiator employs the DCF's default communications service for this.
4. When it receives a 'reconfigure' message, each DCF member node's Pointcut Evaluator applies the specified pointcut and thereby locates all the target join points within its scope.
5. If the command is 'replace', the Aspect Handler extracts execution state from the existing aspect. Similar to local aspect reconfiguration, the state extraction mechanism is optionally supported by distributed aspects, with the capability being dynamically discovered by the DCF member using reflection.
6. Each member node's Aspect Handler then actions the 'add', 'remove' or 'replace' command as appropriate. For 'add' or 'replace', this may involve obtaining the aspect from an Aspect Repository. It will also involve weaving the aspect according to the specified scope and locus (which may involve creating a remote binding if per-DCF scope is requested).
7. Each node replies to the initiator that it has completed the reconfiguration locally.
8. When all nodes have reported completion the initiator node returns control to the caller of *reconfigure()*.

Note in passing that there is considerable scope for optimising this protocol in terms of performance. For example, the configuration of aspect repositories in the system, and the corresponding choice of whether to pass aspects by value or by reference, can have a significant influence on performance, as can the use of, and location of, remotely accessible per-DCF aspects.

5. Evaluation
To evaluate AO-OpenCom approach to offer flexible dynamic reconfiguration requirements we use a case-study based methodology (described in Section 5.1). Then, in Section 5.2 we describe the AO-OpenCom use case solution. Finally, in Section 5.3 the reconfiguration performance is evaluated.

5.1 Airport Crisis Management Scenario
The use-case scenario is inspired by an *airport crisis management* scenario taken from the EU DiVA FP7 STREP project [3]. This was chosen because it offers a realistic scenario taken from a real project and because it offers sufficient opportunities for dynamic reconfiguration. The architecture of the crisis management scenario consists of four different domains: the Main Control Room, Administration, Sales, and Terminal. The Main Control Room centralises all phases of the management of the other three domains by determining the different types of dynamic reconfiguration necessary to maintain their optimal operation. More specifically, the Main Control Room is responsible for identifying any crisis, building appropriate crisis management strategies according to the nature of the incident, collecting crisis information and providing it to all the domains dealing with crisis management. The Main Control Room contains human crisis actors and a crisis management system offering a messaging system for crisis actors so that they can communicate through the exchange of text messages. The Main Control Room dynamically reconfigures the crisis management system configuration according to the crisis type and context. The Administration domain hosts the key stakeholders (CEO, Operation Manager, CIO) representing the airport's decision making authority. In case of any crisis they need to be notified immediately. In crisis situations, the Sales and Terminal domains are notified about incidents and, based on the gravity of the incident, the sales of ticket may be stopped and Terminal operations (such as boarding) stopped or delayed.

As a crisis situation is initiated from the Main Control Room, alerts sent to the different crisis actors within the airport are logged to keep track of events and can be studied later on for service improvement. Alerts are logged during both crisis and non-crisis situations. In a non-crisis situation, all crisis actors send their logs to the main control room. Under a crisis situation only crisis actors involved in the crisis are logged.

5.2 AO-OpenCom based solution
From the use-case scenario, the MessageHandler and the Communication modules as shown in Figure 8a are two main

entities responsible for the transmission of messages among nodes before reconfiguration and after reconfiguration in Figure 8b. The Messager module is responsible to transmit messages based and requires an IMesageHandler interface which takes as parameters the MessageType, DCF, port id and communication mode. Figure 9 illustrates the MessageHandler code fragment implementation. From Figure 9, Line 1 implements the IMessageHandler interface to handle the message communication to the Communication component. Line 2 specifies the receptacle reference of the MessageHandler component to the Communication component and Line 3 details the reference to the AO-OpenCom runtime base-level kernel. Lines 4-7 contain the constructor for the MessageHandler component. Lines 8-12 detail the call to the sendMsg operation of the Communication component. In the use-case scenario, the alert logging is a crosscutting concern that is tangled across multiple nodes. In order to facilitate the reconfigurability the application developer needs to untangle this functionality from the component implementation. Another requirement from the use-case scenario is the need to provide secure transmission of the logs. To do so an encryption module is needed and since the encryption module is crosscutting similar to the alert logging module, it needs to be applied as an aspect (as shown in Figure 8c).

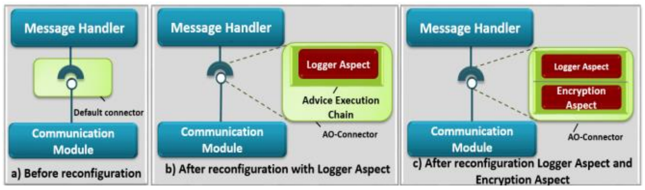
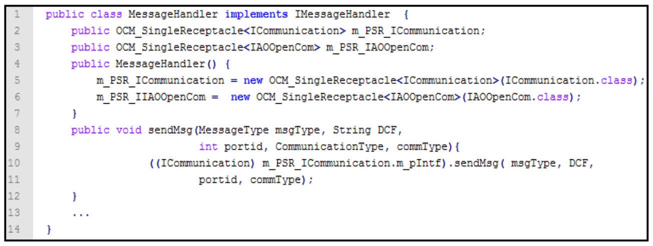


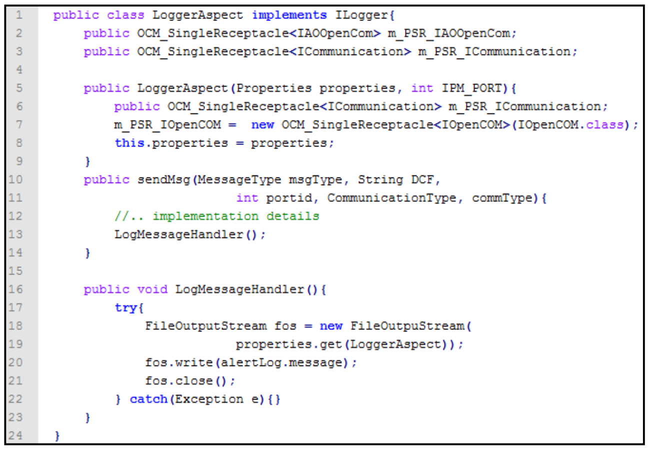
Figure 8. Reconfiguration for use case scenario.



```
1   public class MessageHandler implements IMessageHandler  {
2       public OCM_SingleReceptacle<ICommunication> m_PSR_ICommunication;
3       public OCM_SingleReceptacle<IAOOpenCom> m_PSR_IAOOpenCom;
4       public MessageHandler() {
5           m_PSR_ICommunication = new OCM_SingleReceptacle<ICommunication>(ICommunication.class);
6           m_PSR_IIAOOpenCom =  new OCM_SingleReceptacle<IAOOpenCom>(IAOOpenCom.class);
7       }
8       public void sendMsg(MessageType msgType, String DCF,
9                           int portid, CommunicationType, commType){
10              ((ICommunication) m_PSR_ICommunication.m_pIntf).sendMsg( msgType, DCF,
11                          portid, commType);
12      }
13      ...
14  }
```

Figure 9. Code extract of the Message Handler

The code-fragment of the local Logger aspect implementation is illustrated in Figure 10 and that of the remote Logger aspect in Figure 11.

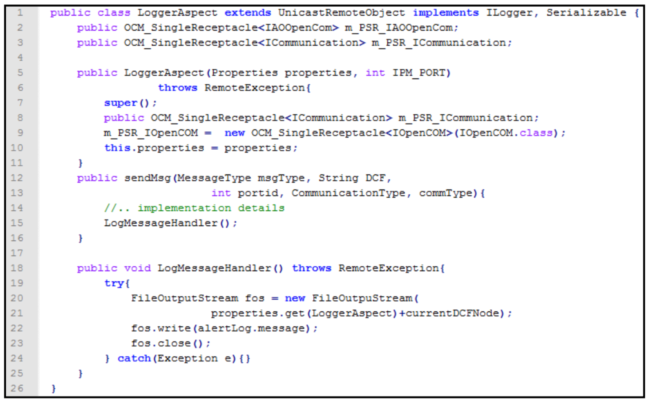


```
public class LoggerAspect implements ILogger{
    public OCM_SingleReceptacle<IAOOpenCom> m_PSR_IAOOpenCom;
    public OCM_SingleReceptacle<ICommunication> m_PSR_ICommunication;

    public LoggerAspect(Properties properties, int IPM_PORT){
        public OCM_SingleReceptacle<ICommunication> m_PSR_ICommunication;
        m_PSR_IOpenCOM =  new OCM_SingleReceptacle<IOpenCOM>(IOpenCOM.class);
        this.properties = properties;
    }
    public sendMsg(MessageType msgType, String DCF,
                        int portid, CommunicationType, commType){
        //.. implementation details
        LogMessageHandler();
    }

    public void LogMessageHandler(){
        try{
            FileOutputStream fos = new FileOutpuStream(
                        properties.get(LoggerAspect));
            fos.write(alertLog.message);
            fos.close();
        } catch(Exception e){}
    }
}
```

Figure 10. Code extract of the Local Logger Aspect

5.3 Evaluating Reconfiguration Protocol

To measure the reconfiguration protocol a small network of five standalone workstations has been employed: a 1.8 GHz Core Duo 2 PC with 3GB RAM; a 3.4 GHz Pentium IV PC with 1GB of RAM; a 2.8GHz Pentium IV PC with 1 GB of RAM; a 1.33 GHz Core Duo 2 laptop with 2GB of RAM; and a MacBook 2.4 GHz Core Duo 2 laptop with 4GB RAM. Two of the machines ran Ubuntu 12.04, two ran Windows XP with service pack 3, one ran Windows 7 SP1 and the other ran OS X Mavericks. All of these are connected via a 100Mbps local area network. While this network is small in terms of physical nodes, each physical node is used to host multiple instances of the framework and in this way the evaluation environment was able to scale to support the equivalent of 100 nodes (frameworks) under four Java VMs per machine. Each evaluation machine was installed with the AO-OpenCom framework which was executed on a Java 1.7 virtual machine (VM). Note that the different machines used to perform the experimental setup demonstrate the capability of AO-OpenCom of being deployed independently in various operating system environments and with different hardware resources as long as these machines support the Java VM. Each machine was able to scale to support 100 of these configurations as virtual nodes.



```
public class LoggerAspect extends UnicastRemoteObject implements ILogger, Serializable {
    public OCM_SingleReceptacle<IAOOpenCom> m_PSR_IAOOpenCom;
    public OCM_SingleReceptacle<ICommunication> m_PSR_ICommunication;

    public LoggerAspect(Properties properties, int IPM_PORT)
                throws RemoteException{
        super();
        public OCM_SingleReceptacle<ICommunication> m_PSR_ICommunication;
        m_PSR_IOpenCOM =  new OCM_SingleReceptacle<IOpenCOM>(IOpenCOM.class);
        this.properties = properties;
    }
    public sendMsg(MessageType msgType, String DCF,
                        int portid, CommunicationType, commType){
        //.. implementation details
        LogMessageHandler();
    }

    public void LogMessageHandler() throws RemoteException{
        try{
            FileOutputStream fos = new FileOutpuStream(
                        properties.get(LoggerAspect)+currentDCFNode);
            fos.write(alertLog.message);
            fos.close();
        } catch(Exception e){}
    }
}
```

Figure 11: Code extract of the Remote Logger Aspect

5.3.1 *Add command-action*

To evaluate the performance overhead of the reconfiguration protocol add-command-action, the logger aspect is woven at the communication stack. The reconfiguration involves weaving the logger aspect at the AO-Connector connecting the Message Handler and the Communication Module. To perform this reconfiguration, the reconfiguration developer needs to specify the reconfiguration request by writing code along the lines of Figure 12 (the code is simplified for presentational purposes).

Pointcut pc = **new** Pointcut("*", "*MessageHandler*", "*ICommunication*","msg*");

Aspect aspectlist = **new** Aspect(Logger);

Configurator.reconfigure(pc, add, aspectlist, perDCF, before);

Figure 12. Reconfiguration specification

The *Configurator.reconfigure()* call takes the given pointcut and aspect specifications which are as follows: the aspects that need to be "added"; the scope of the reconfiguration, stating that this reconfiguration need to be applied for all nodes; and that the weaving locus should be a *before* advice weaving. The results of the experiment are illustrated in Figure 13. The results confirm the expected outcome that as the number of reconfigured nodes increases, the amount of time required to perform reconfiguration increases linearly. The result shows that on a single node (as would be expected) the reconfiguration using the local pointcut and local aspect is similar to that of using remote pointcut and local aspect, and the reconfiguration using local pointcut and remote aspect is similar to that of remote pointcut and remote aspect. The differences between LL, RL and LR and RR lie in the remote aspect instantiation for LR and RR. This instantiation is an out-of-band overhead on the initiator node and if the aspect is already instantiated in the aspect repository, then the reconfiguration time is decreased, with the overhead comparable to that of LL and RL. The results also show that:

i) For less than 170 nodes LL offers significantly better reconfiguration performance than LR. This means the instantiation of the local aspect across each node is expensive as the number of reconfigured nodes gets above 170 nodes.
ii) Above 160 reconfigured nodes LL reconfiguration overhead gets worse compared to RR. The difference at 10 nodes between LR and RR when compared to LL, is due to the remote pointcut offering less reconfiguration overhead for RR.
iii) For less than 220 reconfigured nodes RL offers better reconfiguration time compared to RR. This is explained by the instantiation of the remote aspect being expensive, and the reconfiguration cost offset the instantiation time as more than 220 nodes are reconfigured.
iv) When reconfiguring more than 220 nodes RR reconfiguration is better compared to LL, LR and RL. This is mainly attributed to the instantiation cost while weaving remote aspects as well as the method *Lookup()* operation to ensure remote aspect interface compatibility as the remote aspect is woven to the AO-Connector chain.

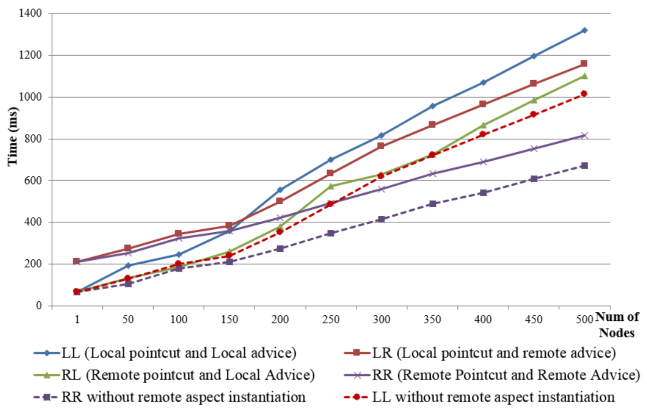


Figure 13. Add reconfiguration command-action

Overall, the experimental results show that there is a large overhead while reconfiguring on a single node using LR and RR compared to LL and RL. As the number of reconfigured nodes increases, reconfiguration using RL and RR offers better performance compared to LL and LR. The higher reconfiguration time using LR and RR is mainly due to the remote aspect instantiation on the initiator node which is on average 147ms. Having the remote aspect instantiated will amortise the reconfiguration time as illustrated in the dotted lines in Figure 13 for both LR and RR making RR more optimum for large scale reconfiguration. The time to set up the advice may not be the most important consideration overall, the in-band overhead would likely be more significant.

5.3.2 *Replace command-action*

Here the Logger aspect is replaced by the Multicast Logger aspect. This operation involves a *replace* operation of the existing Logger aspect at the message handler AO-Connector by the Alert Logger and the resulting reconfiguration. To measure the reconfiguration overhead of the replace command-action the same environmental setup as in Section 5.3.1 is used, whereby the woven Logger aspect is replaced by a Multicast Logger aspect. The measurement results of this experiment are illustrated in Figure 14. The results show an increase in the reconfiguration time to perform the *replace* command-action compared to the *add* command-action. This is due to the fact that the replace command-action requires the un-weaving of the old aspect component followed by the weaving of the new aspect component, while that of the add command-action involves only the aspect weaving. The results show:

i) RL offers better reconfiguration compared to LL, RL and RR to reconfigure up to 160 nodes. This is explained by the quantification of the pointcut being performed only on the initiator node and the instantiation remote aspect on smaller number of nodes offers better reconfiguration overhead compared to remote aspect instantiation.
ii) A steeper gradient to reconfigure LR compared to RR as the number of reconfigured nodes increases, demonstrating that pointcut quantification on each reconfigured node is expensive.
iii) RR setup offers better reconfiguration time compared to LL, LR and RR, similar to the add command-action.

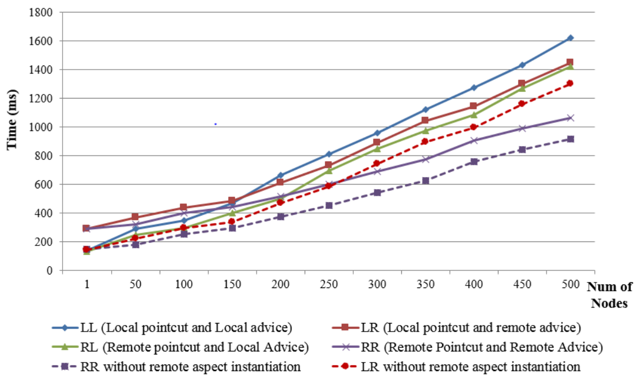


Figure 14. Replace reconfiguration command-action

An additional experiment was performed to measure reconfiguration overhead while updating the aspect using cached pointcuts by retrieving the pointcut from the AO Repository. The use of a cached pointcut avoids the use of the distribution meta-model to retrieve the join point. The measurements of the experiment are shown in Figure 15. The results show a significant decrease in the reconfiguration time for all the four reconfiguration operations. The decrease in overhead is on average by 30% per reconfigured node. It should be noted that the cached pointcut still requires the parsing of the XML specification to check if the required reconfiguration request matches the ones previously retrieved and cached. The results also show that the time needed to perform remote aspect is lower than that of local aspect. This is explained by the fact that the remote aspect is instantiated only once on the initiator node compared to local instantiation on each Aspect Repository in the case of LL and RL.

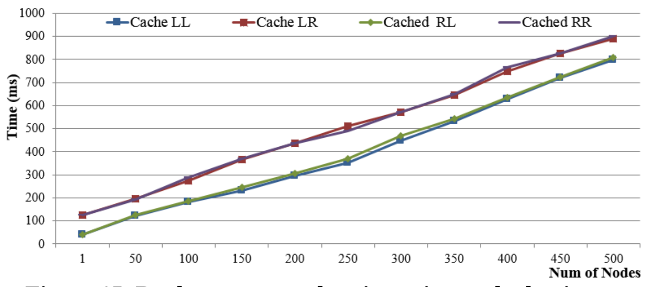


Figure 15. Replace command-action using cached pointcut

5.3.3 *Remove command-action*

Finally, the Logger aspect may no longer be necessary, and can be removed. The reasons behind a remove reconfiguration command include: removing the Logger aspect as the policy associated to it has been deleted, or being incompatible (such as semantic inconsistencies) and needs to be removed to allow a reconfiguration to be completed. This involves a remove reconfiguration command, such that the reconfiguration leaves an empty advice chain at the join point. As discussed earlier, an AO-Connector is woven to support the advice chain at the appropriate join point. The AO-Connector component should be removed when no aspect is present at the join point. This is because leaving an empty AO-Connector will result in an in-band overhead that negatively affects the system performance. The results of the remove command-action experiment are illustrated Figure 16. The results show a lower reconfiguration overhead for un-weaving an aspect compared to the weaving or replacing of an aspect. This is because, the un-weaving of aspects involves the parsing of the reconfiguration operations from the script, locating the join point and setting the reconfigured join point to quiescent mode and removing the references of the aspect from the AO-Connector. From Figure 16, it can also be observed that the un-weaving of an aspect is faster for LL and RL compared to LR and RR. This is explained by the reflective calls needed to get the aspect operations before its methods are removed at the AO-Connector. For the remote aspect, the reflective call involves the *Lookup()* method for the remote aspect causing higher performance penalty. The next measurement involved measuring the amount of time required to remove an aspect and then remove the AO-Connector by reinstalling the default connector. The additional reconfiguration time per node is about 10ms for all the setup reconfigurations (LL, LR, RL and RR). This lower increase is mainly due to the fact no reflective calls are needed with an *Unload()* followed by a *Connect()* method call executed.

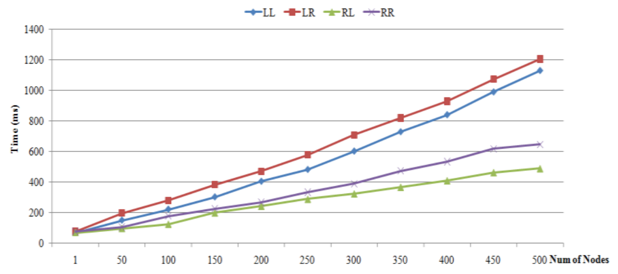


Figure 16. Remove command-action reconfiguration

5.3.4 *Reorder reconfiguration protocol*

A reorder command action may be applied when more than one aspect is woven at a join point. To measure the overhead of the reorder command action all messages sent are encrypted and then logged. This reconfiguration involves the reorder operation which reorders the advice chain.

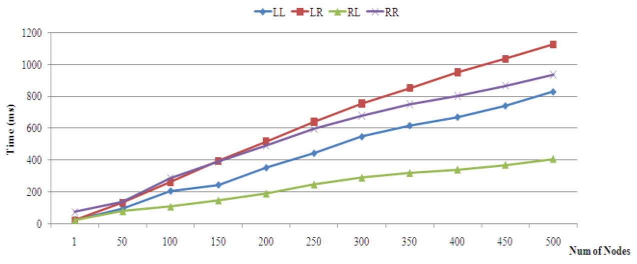


Figure 17. Reorder command-action reconfiguration

The results of Figure 17 show that the reorder command-action has a significantly lower reconfiguration cost than the coarse-grained operations. Additionally, it can be observed that the cost of using LR and RR to perform the reorder reconfiguration is significantly higher (by 50%) compared to LL and RL. The higher overhead is explained by the *Lookup()* reflective method call for remote aspects, introducing significantly higher overhead.

5.3.5 *Evaluating resource overhead*

This section examines the resource costs (in terms of memory) in reconfiguring the middleware platform using a reliable and an unreliable communication protocol. Figure 18 shows the resource overhead on the initiator node of AO-OpenCom using first a reliable communication protocol (JGroups) and then an unreliable multicast protocol. The measures represent the resource overhead of the Distributed Meta Architecture: i.e. configurations for the binding of the case study application and the base elements of the AO-OpenCom platform. Furthermore, it can be observed that there is an extra memory overhead from the use of reliable communications. The additional cost ranges between 3.9% to 119.2% increase in the amount of memory consumed by each node. Additionally, it can be observed there is a linear increase in resource overhead as the number of nodes increases. This measure demonstrates that a large part of resource overhead is incurred to ensure reliable communication and is representative of the increase in overhead as applications are reconfigured across a distributed system.

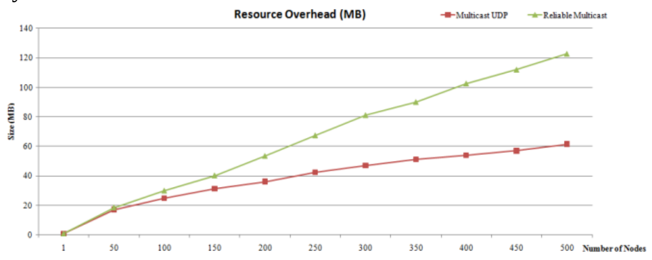


Figure 18. Reconfiguration resource overhead

6. DISCUSSION AND RELATED WORK

The experiments results demonstrate the flexibility of AO-OpenCom to robustly support a wide range of dynamic reconfiguration variability in terms of i) *granular scope reconfigurability* supporting coarse-grained reconfiguration using the reconfiguration command-actions (add, replace and remove and operations) provide coarse-grained and fine-grained reconfiguration (reorder operation); ii) *vertical scope reconfigurability* allowing both infrastructure services reconfigurability as demonstrated and measured in Section 5.3.5 and application services reconfigurability as the demonstrated in Section 5.3.1 to Section 5.3.4; iii) *horizontal scope reconfigurability* supporting both local and distributed reconfiguration as demonstrated in Section 5.3.1 to Section 5.3.4; and iv) *performance;* and v) *resource overhead* with the main resource overhead within the AO-OpenCom being from the distributed framework which is influenced by the choice of the communication protocol. The resource overhead on each node can be minimised by creating group nodes and having one node hosting the distributed framework of the group of nodes.

Turning to related work, a number of AO-middleware platforms have emerged. Most of the AO-middleware platforms offer only coarse-grained reconfiguration. However, AspectOpenCom [4] and JAC [10] provide support for fine-grained reconfigurability, by allowing the reordering of aspects at a join point. Regarding application-level vertical scope of reconfigurability most middleware platforms provide support to weave and un-weave aspects that are applied to an application at runtime. However, with the exception of FAC [11], none of the AO-middleware platforms supports infrastructure-level evolution, but FAC is limited to local infrastructure-level only. With respect to horizontal scope reconfiguration, AO-middleware platforms support three types of aspect composition. First *aspect composition being separate from the distribution model*, such that the middleware architectures use their own distribution specific technologies to provide distribution. Most of the AO-middleware platforms (PROSE [13], JBoss-AOP [2], Lasagne [16], DyReS [17] and CAM/DAOP [12]) have aspect being separate from the distribution model. These AO-middleware platforms use distribution technologies to provide distribution. Second, *aspects abstractions are used with the distribution model*, such that the middleware architectures use aspect technology to provide reconfiguration. DJasCo [1], JAC [10] and ReflexD [15] platforms use aspects abstractions with the distribution model by offering the remote pointcut functionality. Third, *aspect form an integral part (i.e. as a first class entity) of the distribution model*, such that the middleware platforms (DyMAC [7] and Damon [8]) use aspects to provide reconfiguration and build the distribution models. The DyMAC platform supports both remote advice and remote pointcut functionality but the platform only allows remote aspect deployment (aspects are non-reconfigurable in the platform). In the case of Damon the explicit connector defined for each composition makes the composition of distributed aspects non-transparent. However, none of the AO-middleware platforms provide for flexible distributed reconfiguration with the support of local pointcut - local advice; remote pointcut - local advice; local pointcut – remote advice; and remote pointcut – remote advice.

Finally, with respect to performance and resource overhead, AO-middleware platforms using byte-code instrumentation weaving (DJasCo, JBoss AOP, JAC, Damon, ReflexD) usually introduce some level of overhead in the system while performing reconfiguration, while CAM/DAOP and Lasagne which use a message interception mechanism to invoke aspects introduce significant overhead. PROSE uses a two-way weaving mechanism such that alternate weaving mechanism can be chosen based on the performance need. In DyMAC, since the weaving is done on all possible join points at load-time the runtime weaving of aspects is not significant. However, similar to AspectOpenCom the use of proxy-based interceptors on all join points even those not having any aspects behaviour bound to them, introduce an indirection in the call invocation for all component communications as the calls need to pass through the proxy. In our approach, the use of default-connector and AO-connector at runtime diminishes consequently the indirection when no aspects are present.

7. Conclusions

In this paper we have presented an aspect-oriented component framework architecture that offers comprehensive AOP support for both local and distributed reconfiguration. The AO meta-framework can be independently deployed such that it imposes no overhead when it is not used and can be dynamically deployed/un-deployed where and when required. In addition, the AO meta-framework is built using the same programming language independent component-based principles as the underlying reflective middleware layer, and the overlying application.

The AO-OpenCom platform provides the development of a fully distributed realisation of dynamic aspects. This is achieved by layering our AO provision on top of the distribution framework and by providing a pointcut language that is inherently distributed in nature (i.e. it supports quantification over capsules). In addition, the AO-OpenCom middleware supports in a natural way the composition of advices that is remote from the advised join points. Furthermore, the AO-OpenCom approach significantly decreases the complexity of deploying new functionality in a distributed environment as compared to the reflective middleware approach. Nevertheless, the lower-level reflective APIs are still available to the developer should they be required. Additionally, the experimental results show that AO-OpenCom is scalable and achieves flexibility providing an important step towards the path of enhancing dynamic reconfiguration in AO-middleware for real-world critical distributed applications.

Acknowledgement

This research work has been supported by the European FP7 Marie Curie ITN “RELATE” (Grant Number, 264840).

8. References

[1] Benavides, et al. Explicitly distributed AOP using AWED. RR INRIA 5882 Technical Report March 2006.

[2] Burke, B. & Fleury, M. ‘JBoss: Aspect-Oriented Middleware’. Tutorial at AOSD 2004.

[3] DiVA. 2009 - Diva-dynamic variability in complex, adaptive systems.ftp://ftp.cordis.europa.eu/pub/fp7/ict/docs/ssai/project-diva_en.pdf.

[4] Grace, P., Truyen, E., Lagaisse, B. & Joosen, W. ‘The Case for Aspect-Oriented Reflective Middleware’. In Proc. of the 6th Workshop on Adaptive and Reflective Middleware 2007.

[5] Grace, P. 2009. ‘Dynamic Adaptation’. In Middleware for Network Eccentric and Mobile Applications, B. Garbinato, H. Miranda and L. Rodrigues (Eds.), pp. 285-304, Springer.

[6] Kephart, J.O and Chess, M. 2003. ‘The Vision of Autonomic Computing.’ In IEEE computer Society Press pg. 41-50.

[7] Lagaisse, B. ‘A Comprehensive Integration of AOSD and CBSD concepts in Middleware’. Ph.D. Thesis, Department of Computer Science, K.U.Leuven, Belgium, Dec 2009.

[8] Mondejar, R., García, P., Pairot, C., Urso, P. & Molli, P. ‘Designing a Distributed AOP Runtime Composition Model’. In 24th ACM Applied Computing, USA 2009.

[9] Parnas, D.L. 1972. ‘On the Criteria to be Used in Decomposing Systems into Modules’. In Proc. ACM, Vol.15.

[10] Pawlak, R., Seinturier, L., Duchien, L. & Florin, G. 2001. ‘JAC: A Flexible Solution for AOP in Java’. In Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns.

[11] Pessemier, N. 2007. ‘Unification des approaches par aspects et a composants’. PhD Thesis, University of Lilles.

[12] Pinto, M., Fuentes, L. & Troya, J.M. 2005. ‘A Component And Aspect based Dynamic Platform’. The Computer Journal, Volume 48, Issue 4, 401-420, 2005.

[13] Popovici, A., Gross, T. & Alonso, G. 2001. Dynamic Homogenous AOP with PROSE. Technical Report, Dept. of Computer Science, March 2001.

[14] Surajbali, B., Coulson, G., Greenwood, P., and Grace, P. 2007. Augmenting reflective middleware with an aspect orientation support layer. In Proc. of the 6th International workshop on ARM 2007, ACM Press, NY, Article 1.

[15] Tanter, E. & Toledo, R. ‘A Versatile Kernel for Distributed AOP’. In Proceedings of the IFIP Conference on DAIS 2006.

[16] Truyen, E. 2004. Dynamic and context-sensitive composition in distributed systems. Ph.D. Thesis, Department of Computer Science, K.U.Leuven, Belgium.

[17] Truyen, E., Janssens, N., Sanen, F., & Joosen W. 2008. ‘Support for distributed adaptations in aspect-oriented middleware’. In Proc. of the 7th Conference on AOSD, 2008.