Preserving Dynamic Reconfiguration Consistency in Aspect Oriented Middleware

Bholanathsingh Surajbali, Paul Grace and Geoff Coulson

Computing Department
Lancaster University
Lancaster, UK
{b.surajbali, p.grace, geoff} @comp.lancs.ac.uk

Abstract

Aspect-oriented middleware is a promising technology for the realisation of dynamic reconfiguration in heterogeneous distributed systems. However, like other dynamic reconfiguration approaches, AO-middleware-based reconfiguration requires that the consistency of the system is maintained across reconfigurations. AO-middleware-based reconfiguration is an ongoing research topic and several consistency approaches have been proposed. However, most of these approaches tend to be targeted at specific contexts, whereas for distributed systems it is crucial to cover a wide range of operating conditions. In this paper we propose an approach that offers distributed, dynamic reconfiguration in a consistent manner, and features a flexible framework-based consistency management approach to cover a wide range of operating conditions. We evaluate our approach by investigating the configurability and transparency of our approach and also quantify the performance overheads of the associated consistency mechanisms.

Categories and Subject Descriptors D.2.7 11 [**Software Engineering**]: Distribution, Maintenance, and Enhancement.

General Terms: Algorithms, Design, Management.

Keywords: middleware; reflection; aspects; dynamic reconfiguration; consistency.

1. Introduction

A key and growing challenge for distributed systems is their need to support *dynamic reconfiguration* in order to maintain optimal levels of service in diverse and changing environments. In response to this challenge, *aspect-oriented middleware* [10, 12, 13, 14, 16, 19] has recently emerged as a promising basis on which to build reconfigurable distributed systems. The core concept of AO middleware is that of an *aspect*: a module that deals with one specific concern and can be changed independently of other modules. Aspects are made up of individual code elements that implement the concern (*advices*). Advices are deployed at multiple positions in a system (*join points*) which are expressed by *pointcuts*—a particular form of composition language.

Dynamic reconfiguration of distributed systems requires assurances that the reconfiguration does not leave the system in an inconsistent state that can potentially lead to incorrect execution or even complete system failure. In AO middleware environments reconfiguration inconsistencies arise from a range of characteristic sources which we classify under two broad headings: *system environment related* sources and *composition related* sources. System environment related inconsistencies occur due to the runtime system environment (e.g. message loss or node crash); whereas composition related inconsistencies refer to application-specific semantic relationships between modules or aspects (e.g. if one aspect is dependent on another than removing the first will result in inconsistency; or if two aspects are mutually exclusive then deploying both simultaneously will result in inconsistency).

In general, avoiding these sources of inconsistency is a difficult task due to the diversity of distributed applications (e.g. centralised/decentralised, static/mobile, small scale/large scale etc) and also because of diverse application-specific factors (e.g. varying dependability requirements, or varying trade-offs between consistency and scalability). Relying on the application developer to ensure the consistency of the system is not feasible under such heterogeneous conditions. Moreover, a *one-size-fits-all* approach to consistency management is not feasible either. Instead, *multiple* consistency strategies should be supported within a framework-based approach so that appropriate strategies can be applied to each set of arising circumstances.

Supporting multiple consistency strategies entails meeting the following key requirements:

- *Configurability*. It must be possible to configure and even reconfigure the consistency-related functionality of the system.
- *Transparency*. Managing reconfiguration across each node is a complex and error prone task for the application programmer. Achieving consistency must therefore involve minimum programmer effort.

To address the above issues and requirements we propose in this paper a *distributed consistency framework* that ensures consistent AO-based dynamic reconfiguration while being tailorable to specific conditions and environments.

The rest of the paper is organised as follows. Section 2 provides a detailed discussion of the various threats to consistency to which distributed applications are prone. In Section 3 we present necessary background on the AO composition technology on which we base our proposal (i.e. our AO-OpenCom platform). Section 4 then presents our distributed consistency framework, which is then evaluated in Section 5. Finally, Section 6 discusses related work, and we offer our conclusions in Section 7.

2. Threats to Consistency

To illustrate threats to consistency under dynamic reconfiguration in distributed systems we now present a simple case-study (see

figure 1) which comprises a multimedia peer to peer network in which heterogeneous peers share data files and interact among themselves. The peers (laptops, PCs, and PDAs) can also operate in different network domains (Internet, Wi-Fi, ad-hoc wireless networks, etc.). Given this environment a wide range of dynamic reconfiguration scenario are feasible. For example:

(i) when a new video codec become available we may want to encapsulate it as encoder and decoder aspects and dynamically deploy it on all nodes with video capabilities;

(ii) when nodes move from a fixed to a wireless network environment we may want to deploy fragmentation and reassembly of the video and audio media frames;

(iii) when application performance degrades at a given node we want to deploy a cache aspect while ensuring that cache consistency is maintained across nodes.

We now present important threats to the consistency of such reconfiguration scenarios. While we do not claim this to be an exhaustive list, we believe it to be strongly indicative of the challenges that must be addressed.
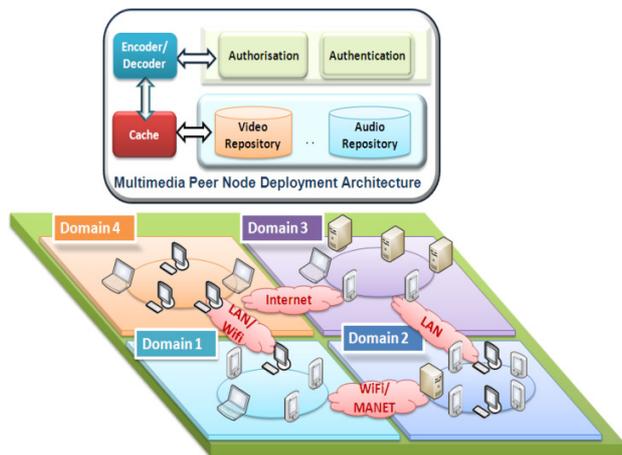

Figure 1. Multimedia application case study scenario

2.1 System environment threats

These relate to reconfiguration inconsistencies that occur due to the instability of the underlying distributed environment in which the reconfiguration takes place. The inherently unstable characteristics of the networks and nodes employed in the scenario increase the chances that a reconfiguration will be compromised. These threat include:

Protocol message disruptions. If reconfiguration-related messages are lost, re-ordered, duplicated or delayed, the consistency of the reconfiguration is clearly compromised. For example, as messages get lost, the initiating node (referred as the coordinator) of the reconfiguration can be mislead into waiting for the reconfiguration to complete.

Local node disruptions. The reconfiguration requests (i) to (iii) sent by the initiator of the reconfiguration may not reach some of the peer nodes. Even if the messaging is unproblematic, individual nodes may still fail to apply a requested reconfiguration. For example:

• the node may be overloaded or may crash;

• a aspect composition request may fail because of resource scarcity on the target node or because the node's local policy forbids it to make the requested change;

• modules or aspects may still be performing computations when an attempt is made to remove or recompose them.

Again, such factors can lead to parts of the intended reconfiguration not being carried out, and consequent inconsistency.

Infrastructure service failures. Aspects to be reconfigured into the system are typically stored in repositories which may get congested with requests, or crash, meaning that aspects may not be available to be deployed (or may perhaps be only deployable in parts of the system). Additionally, different repository instances may have different versions of the aspects: e.g. different versions of the encryption aspects may be produced over time, so that different nodes configure different codec versions and be inconsistent with one another.

Simultaneous reconfigurations. Different reconfiguration requests may arise simultaneously so that reconfiguration-related messages relating to distinct requests may be interleaved and potentially be received in different orders at different nodes. For example, one request might ask for a fragmentation aspect to be replaced, while another asks for it to be removed. There will clearly be different outcomes depending on the execution order of these two requests—and furthermore the outcomes might be different at different nodes.

Unauthorised nodes initiating reconfiguration. Reconfiguration messages may be spoofed by malicious nodes in an attempt to directly and deliberately compromise consistency.

2.2 Compositional threats

These relate to faulty interactions, following reconfiguration, between the newly-reconfigured entities and prior non-reconfigured entities. The associated threats typically involve conflicts and dependencies: conflicts are threats causing negative interactions between system entities; while a dependency threat relates to a 'required' relationship that needs to be associated with the reconfiguration for the system to operate correctly. The different compositional threats are:

Unsynchronised weaving of dependent aspects. Some aspects are inherently dependent on each other; for example, decryption is dependent on encryption, and a cache may be dependent on a remote cache manager. Therefore the order in which aspects are woven is crucial: e.g., we must ensure that an assembler aspect is put in place *before* its associated fragmenter, otherwise fragmented messages may be received which cannot be handled.

Unsynchronised binding of distributed aspects. Some distributed aspect systems employ 'remote aspects' which are used by several distributed client nodes. If such an aspect, e.g. a cache manager is removed without the consent or even the awareness of its client nodes, errors can arise when clients attempt to communicate with the aspect.

Mutual exclusion of aspects. Behavioural conflicts can occur as new aspects are woven. For example adding a logging aspect into our scenario at the same join points as an encryption aspect can result in behavioural conflicts, because the system is open to read the logged, decrypted messages.

3.   The AO-OpenCom Framework

Before discussing our proposed distributed consistency framework, we briefly introduce the software composition technology that underlies our work. *AO-OpenCom* is an extension of the OpenCom component model [5] and provides a distributed AO composition service while allowing aspectual compositions to be dynamically reconfigured. An earlier version of AO-OpenCom was the subject of a prior workshop paper [16]. We revisit it here because the current version differs significantly from the earlier one in key areas.

3.1 Aspects and Aspect Composition.

Aspect composition in AO-OpenCom employs components to play the role of aspects—i.e. an aspect is simply an OpenCom component (hereafter we use the term *aspect-component* when referring to an OpenCom component that is playing the role of an aspect). Aspects are composed using so-called *AO-connectors*. These are specialised connectors that support the run-time insertion of aspect-components.

Internally, an instance of AO-OpenCom is structured as a set of per-node local instances, as illustrated in figure 2, which are combined into a multi-node AO-OpenCom distributed system. The *Distribution Framework* is a plug-in for the AO-OpenCom communication service that sends reconfiguration and management messages to every node in the system; the *ISend* interface provides a *send()* operation, while its *INotify* interface delivers received messages to the AO-OpenCom Configurator.

Turning now to the constituent components, the *Configurator* is responsible for accepting and handling reconfiguration requests from applications. It interacts with the Pointcut Evaluator and Advice Handler components on either the local node or other nodes to actually carry out the requested reconfiguration in terms of AO (re)compositions. The *Aspect Repository* holds a set of instantiable aspect-components. This is composed of a front-end proxy gateway component and a back-end database component. Finally, the *Pointcut Evaluator* evaluates pointcuts and returns a list of matching join points within the framework; and the *Aspect Handler* weaves advices at these join points in the framework.
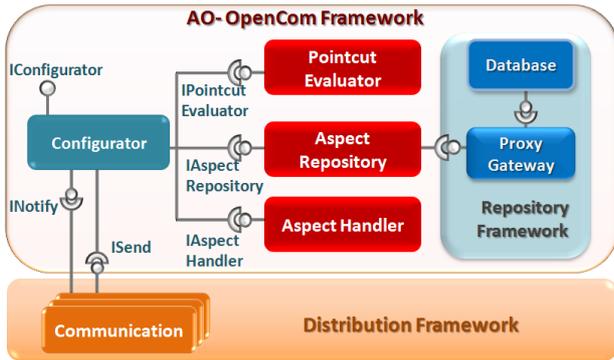

Figure 2. An AO-OpenCom per node instance

3.2 Reconfiguration in AO-OpenCom

The main API provided by an AO-OpenCom for dynamic reconfiguration takes the form of a single operation on the Configurator component:

Configurator.reconfigure(target_dcf, pc, command, aspect, scope, locus).

The *target_dcf* argument specifies which distributed system the reconfiguration should be applied to. The *pc* argument specifies a pointcut that picks out the join points at which the desired reconfiguration should occur. The *command* argument offers options— either 'add', 'remove', or 'replace' an aspect—for the action to be taken at the indentified join points. The *aspect* argument can be a direct reference to a local aspect-component, or an indirect reference to an aspect stored in an Aspect Repository, or a reference to an already-instantiated remotely-accessible singleton aspect. The *scope* argument can be either *per-instance* or *per-distributed system*. The former weaves a distinct aspect-component instance at each specified join point; the latter instantiates a single per-system instance that is connected, potentially remotely, with each specified join point. Finally, the *locus* argument describes how advices should be applied at a selected join point in terms of either *before*, *after* or *around*.

Furthermore, the Configurator is also responsible for the management of *quiescence* (i.e. it ensures that the weaving/unweaving of aspects is not carried out while affected component/aspect-components/connectors are actively processing calls). To support this, the Configurator ensures that the weaving of aspects is not carried out while the relevant connectors or other components are actively passing or processing messages or calls. To do this, it requires that all connectors and components support a basic 'quiescence' interface as follows:

$$status = quiesce(\ timeout);$$

$$status = resume();$$

Because of the strictly stylised composition supported by AO composition, achieving quiescence is a relatively straightforward task compared to non-AO composition (e.g. [8]). The *quiesce()* operation simply freezes the start of the chain of aspects attached to the AO Connector (i.e. the AO-Connectors that correspond to the advices of the woven aspects) to prevent new threads entering, and then waits for any currently executing threads to drain from the aspect chain.

To execute *Configurator.reconfigure()* the following distributed protocol is performed:

1. *Configurator.reconfigure()* is called on one of the AO-OpenCom nodes; we will refer to this node as the 'initiator'.

2. The initiator determines how the aspect is to be applied. In the case of a per-distributed system scope, it instantiates the aspect at a suitable node and sends a remote reference to this 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 name' and implicitly instruct the other members to obtain the aspect from an Aspect Repository.

3. The initiator sends a 'reconfigure' message to all the other AO-OpenCom nodes. This contains the parameters originally passed to *Configurator.reconfigure()*.

4. Upon receiving a 'reconfigure' message, each node's Pointcut Evaluator locates the target join points within its scope.

5. Each 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.

6. Each node replies to the initiator that it has completed the reconfiguration locally.

7. When all nodes have reported completion the initiator node returns control to the caller of *reconfigure()*.

An example of the use of *Configurator.reconfigure()* is given in Section 5.2.

4. The Consistency Framework

In this section we discuss our approach to the support of *consistent* dynamic reconfiguration. This is independent of the basic AO-OpenCom reconfiguration architecture discussed in the above section which handles only the basic mechanics of dynamic aspect (un)deployment. The *Consistency Framework* (COF) illustrated in Figure 3 consists of: a *System Consistency Framework*, a *Compositional Consistency Framework* and a set of 'threat aspects' which are responsible for guarding against consistency threats such as those identified in Section 2; these threat aspects are woven into the lower-level frameworks using the usual AO-OpenCom facilities.

The fundamental strategy of the COF is to guard against consistency threats by deploying 'threat aspects' *at appropriate join points within AO-OpenCom itself*. The benefit of this strategy is that threats can be handled in an incremental, selective and extensible manner where specific threat aspects can be deployed to guard against specific consistency threats. Crucially, we are using the same approach to guard against consistency as we are for 'ordinary' application-level dynamic reconfiguration: i.e. using aspect composition.

Turning now to the detail, the Consistency Configurator is responsible for managing these threat aspects and for deploying them at appropriate join points within the AO-OpenCom-based distributed system (see below).
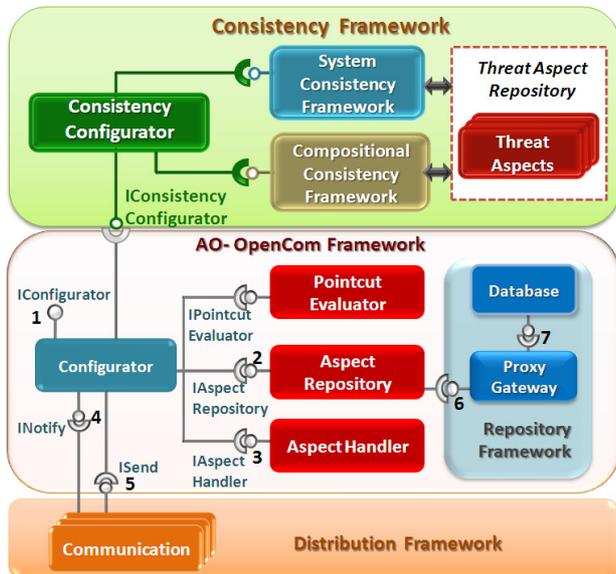

Figure 3. Applying Consistency Framework to AO-OpenCom

We now turn to a discussion of how the Consistency Configurator resolves each of the threats discussed in Section 2 by deploying appropriate threat aspects. When discussing the weaving of threat aspects, the following paragraphs refer to the numbered join points, 1-7, within the AO-OpenCom framework that are illustrated in Figure 3.

4.1 Addressing System Environment Threats

The Consistency Configurator uses the System Consistency Framework to instantiate the appropriate system environment threat aspects based on the reconfiguration needs as described in this section.

Protocol Message Disruption. To ensure that reconfiguration messages are not lost, the System Consistency Framework uses a *reliability threat aspect* and this aspect is woven at join points 4 and 5. The reliability threat aspect implements a reliability protocol atop the Distribution Framework to ensure that all messages are reliably received by each member. Because it is implemented as an aspect, this behaviour can be realised using various underlying mechanisms and can therefore be made straightforwardly applicable to a variety of implementation environments. This point is an important one and also applies to all the other threat resolution aspects to be discussed below.

In more detail, our currently-implemented reliability threat aspect is composed of an aspect with two advices and a 'message store'. The first advice is woven 'before' join point 5, and has the task of piggybacking reliability information to the message before it is sent via the *ISend* interface. The second advice is woven as a 'before' advice at join point 4 (i.e. before the message is delivered to the Configurator via *INotify*); this monitors incoming messages (and caches them in the message store), detects any losses within the transmission sequence, and requests retransmission of lost messages.

To weave the reliability threat aspect in a consistent manner (this again applies also to all the other threat resolution aspects to be discussed below) the *quiesce()* operation is first called on the connectors at join points 4 and 5 by the Consistency Configurator. Upon successfully achieving quiescence, the reliability threat aspect is woven at the front of the advice chain list (for brevity, we discuss this weaving process only for join point 5; see Figure 4); hence, it is invoked before method calls go to the Distribution Framework. Once the reliability threat aspect have been successfully woven at both join points, the *resume()* operation is called by the Consistency Configurator.
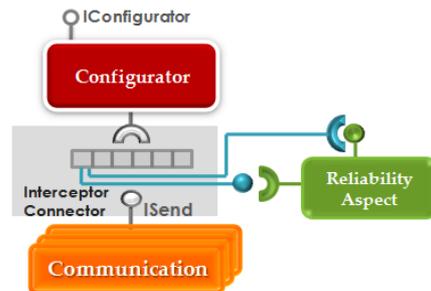

Figure 4. Weaving the reliability threat aspect at join point 5

Local Node Disruption Threat. To guard against this threat, the System Consistency Framework instantiates a *consensus threat aspect* and this aspect is woven at join points 4 and 5 to ensure that local node failures or disruptions do not compromise the consistency of the system. This aspect is 'flexible' in that it can implement any one of a range transaction protocols [7] depending on the specific requirements and deployment environment. To illustrate the operation of the advices we briefly describe our two-phase commit implementation. In this implementation, a 'before' advice woven at join point 5 takes messages before they are sent and converts them into the required sequence of messages for two-phase commit. Correspondingly, the 'around' advice at join

point 4 receives these transaction protocol messages and sends phase acknowledgements; it also communicates with the AO-OpenCom Configurator to enact or undo the local reconfiguration as appropriate.

Infrastructure Service Failures Threat. To guarantee the liveness of the infrastructure services (e.g. the Aspect Repository), the System Consistency Framework uses a *replication aspect*. This aspect is woven at join point 6 as an 'around' advice. Based on application requirements, a number of replication algorithms could be used to ensure maximum aspect availability and consistency during updates—e.g. the Coda [15] or Bayou [6] algorithms. More advanced algorithms which consider specific application and context requirements could also be used: e.g. Beloued [2].

Further, the System Consistency Framework uses a *load balancer aspect* to manage the load across the infrastructure services and this aspect is woven at join point 6 as a 'before' advice. Our current load balancer algorithm implements both the push and pull migration approaches [11]. The detailed functionality of the load balancing algorithm is beyond the scope of the paper; but, in brief, with push migration, periodic checks are made on the load of particular replicated repository loads, and as imbalances are found the load is evenly distributed from overloaded to less busy repositories. And the pull technique arranges that an idle replicated repository can transparently take tasks from a busy repository.

To prevent version conflicts in the Aspect Repository, the System Consistency Framework uses a *concurrency management aspect*. This aspect is woven as a 'before' advice at join point 7. The concurrency mechanism uses an optimistic read/write locking mechanism with priority for readers. Calls to update an aspect instance/version in the repository access the lock as a writer such that a writer can access the lock when there are no readers, while calls to retrieve aspect instances access the lock as a reader.

Simultaneous Reconfiguration Threat. To ensure that simultaneous reconfiguration requests do not interfere with one another, the System Consistency Framework uses a distributed *read/write concurrency aspect* and is woven at join point 1. This is an 'around' advice, the 'before' part being activated before the *Configurator.reconfigure()* is called. The advice then attempts to access the framework's lock set by the concurrency aspect, and blocks the call until this is obtained, at which point the reconfiguration can proceed. At this point, any reconfiguration attempts by other nodes are blocked until the present reconfiguration is complete, at which point the Configurator returns the *reconfigure()* call, and the 'after' part of the 'around' advice releases the lock.

Unauthorised Reconfiguration Threat. To prevent unauthorised nodes initiating reconfiguration, the System Consistency framework uses a series of security aspects, which are subsequently woven at join points 4 and 5. These comprise aspects that each addresses a different flavour of security threat: e.g. access control, integrity or confidentiality. The weaving order of these aspects is crucial: of the three mentioned the order would be authentication, confidentiality and then integrity.

Currently, an *authentication aspect* is woven as a 'before' advice at join point 5 such that it is called before the Distribution Framework and performs access control before allowing continuation. Then a *confidentiality aspect* encrypts the arguments of method calls as they are passed through the Distribution Framework. This is achieved by weaving an encryption advice as a 'before' advice at join point 5 and a decryption advice at join point 4, also as a 'before' advice. Finally the System Consistency Framework implements an *integrity aspect* in terms of an SSL layer between reconfigured nodes.

4.2 Addressing Compositional Threat

The Consistency Configurator uses the Compositional Consistency Framework to instantiate the appropriate compositional threat aspects based on the reconfiguration needs as described below.

Unsynchronised Weaving of dependent aspect Threat. The Compositional Consistency Framework uses a *transaction management concurrency protocol* or *coordination protocol* to preserve compositional dependencies. Each of the protocols is encapsulated as an aspect and is woven as a 'before' advice at join points 4 and 5. This process is equivalent to that used for threat 2. Here, the *Saga* transaction model [7] allows dependent aspects to be divided into a sequence of *sub-transactional aspects*, each of which manages an associated compensating sub-transaction that can be triggered to undo the effects of the committed sub-transaction aspect in case one fails.

With respect to the coordination protocol, protocol the Compositional Consistency Framework uses the NeCoMan [9] protocol which is encapsulated as an aspect and woven to provide synchronisation between the reconfigured entities.

Unsynchronised binding of distributed remote aspects. To prevent race conditions in which remote connectors attempt to communicate with remote aspects that have previously been removed, a 'before' advice is woven at join point 3. This detects when a 'remove' command is passed to the Aspect Handler, and in response weaves a *proxy caretaker aspect* this is woven in front of proxies for the removed application aspect. Then, when a remote client (connector) attempts to invoke this removed aspect, the proxy caretaker aspect is invoked instead which redirects and informs the remote connector that the referenced aspect has been removed. To avoid the connector from invoking the aspect in the future, it removes the remote aspect reference from its aspect chain when it receives the 'remove reference' message.

Mutual exclusion of Aspect(s) Threat. To ensure that conflicting aspects are not composed, the Compositional Consistency Framework uses a *semantic reasoning and resolution aspect* (e.g. [17]) and is applied at join points 1 and 4. This aspect holds application-specific rules about which mutual exclusive behaviours are allowed and not allowed when reconfiguration (both addition and removal of aspects) is performed. Using reflection, it identifies aspect(s) woven at the join point and determines if adding or removing the aspect will cause any inconsistencies. For detected conflicts an exception is raised and the reconfiguration is aborted.

4.3 Ordering of Threat Aspects

Although the threats discussed above are essentially orthogonal to one another, the order in which the corresponding aspects are composed is still important. For example, when the consensus aspect is woven at join points 4 and 5, the reconfiguration can proceed in either of the following ways: (i) if no threat aspects are deployed then the consensus aspect is then woven as a 'before' advice; or (ii) in the case where the threat 1 aspect has already been woven, the consensus aspect is woven as a 'before' advice with position 2. The decision is determined from priority ordering information attached as attributes to the individual aspects. Weaving the reliability aspect first ensures that a reliable consensus protocol is selected.

The order in which aspects woven at the same join point are invoked affects the reconfiguration semantics. This is particularly true for join points 4 and 5 at which numerous aspects are woven. Aspects being executed in the wrong order could lead to situations in which a message needing to be processed by a particular aspect has already been consumed by another.

To guard against such eventualities, the COF mandates a particular order for the weaving of the threat aspects. These are illustrated in Figures 6(a) and 6(b) which respectively illustrate the required ordering at join points 4 and 5.
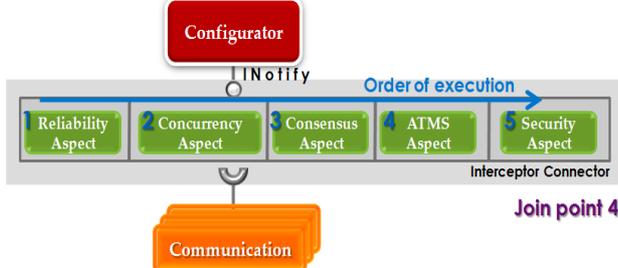

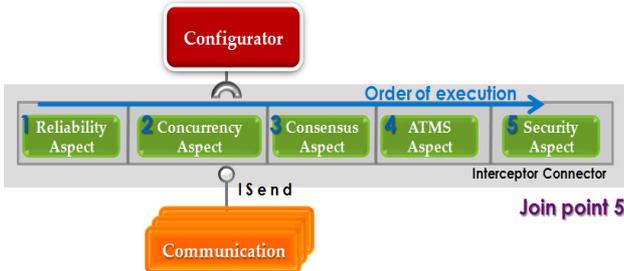**Figure 6(a).** List of threat aspects woven at join point 4

Figure 6(b). List of threat aspects woven at join point 5

5. Evaluation

We focus on two dimensions of evaluation: (i) the extent to which AO-OpenCom/COF achieves our stated goals of configurability and transparency; and (ii) the overhead of AO-OpenCom/COF in 'typical' usage scenarios.

5.1 Configurability

In Section 4.1 we have already demonstrated the *configurability* of AO-OpenCom/COF in addressing a wide range of consistency threats. Our general approach to dealing with such threats—i.e. by selectively applying threat aspects to join points in AO-OpenCom itself—is inherently highly configurable and can be changed or extended simply by applying different threat aspects. However, two potential vulnerabilities of our approach might become evident if new threat aspects are added to the set we have already identified: (i) it could become harder to keep track of the threat aspect ordering constraints discussed in Section 4.2; and (ii) there could be an increased possibility of undesirable interactions between the behaviour of the different threat aspects. The extent to which these vulnerabilities become problematic will become clearer with experience. However, we believe that the set of threat aspects we have identified is already quite comprehensive, and that many cases can be covered with the current set alone. Underlying this belief is our experience that most threats seem to reduce to a tractable number of common underlying patterns.

5.2 Transparency

Turning now to the issue of *transparency*, AO-OpenCom/COF naturally supports a *selectively transparent* approach. At one extreme, an appropriate set of threat aspects can be pre-configured at application start-up time so that the application programmer who wishes to initiate a run-time reconfiguration needs only to make the appropriate call to *Configurator.reconfigure()*. This achieves complete transparency of consistency-related mechanisms. At the other extreme, the programmer can be explicit about which threat

aspects should be put in place for each reconfiguration. In this case, COF will apply the requested threat aspects on-the-fly (if they are not already present) before proceeding to perform the requested reconfiguration. Note that this extreme is still *partially* transparent as the programmer is protected by the Consistency Configurator from the low level details of actually weaving the threat aspects.

To illustrate the partially transparent case consider a reconfiguration scenario relating to the case study in Section 2. Assume that the application programmer wants to add an MPEG4 video codec aspect to all nodes in domains 1 and 2 which already have video-codec components with an *IMPEG* interface. Further assume that domains 1 and 2 offer reliable TCP-based communications. The programmer would specify the reconfiguration request by writing code along the lines of Figure 7 (the code is simplified for presentational purposes).

Note that the required threat aspects are specified as part of the aspect specification. In this case no compositional threats are applicable, and the *protocol message disruptions* threat (T1) is not applicable either because of the availability of TCP. This leaves only the remainder of the 'system environment' threats: i.e. threats T2-T5. The *Configurator.reconfigure()* call takes the given pointcut and aspect specifications and also specifies that the specified aspect should be added, that the scope of the reconfiguration should be the entire DCF and that the weaving locus should be *before*.

```
Pointcut pc = new Pointcut( "domain1* && domain2*", "video-
              codec*", "IMPEG", "video-player*");
Aspect aspectVideo = new Aspect(MPEG4VideoCodec, "T2 T3
              T4 T5");
Configurator.reconfigure(multimedia_app, pc, add, aspectVideo,
              perDCF, before);
```

Figure 7. Reconfiguration specification

5.3 COF Overhead

The following experiment was performed on two Core Duo 2, 1.8 GHZ PCs' with 2GB RAM running Windows, and using the Java-based version of AO-OpenCom. Each measurement was repeated ten times and mean values taken to discount anomalous results. The purpose of the experiment was to evaluate the performance overhead of dynamic reconfiguration operations using AO-OpenCom and COF, We approached this by instrumenting an implementation of the application scenario described in Section 5.2, while using different threat aspect configurations from the consistency framework.

The results are shown in Figure 8 which shows the measured overhead of the following 4 cases: (i) reconfiguration without COF; (ii) reconfiguration using COF with the system consistency framework threat aspects only; (iii) COF with the compositional consistency framework threat aspects only; and (iv) COF with both the system and compositional consistency framework threat aspects.

We can see a linear increase in overhead when applying COF for compositional threat aspect while a non-linear increase of overhead for System Consistency Threat aspect used as the number of reconfigured nodes is increased. This is explained by:

- the fact that the initiator node is a bottleneck (this could in principle be alleviated by configuring AO-OpenCom with slave Configurators to increase parallelism);
- weaving of dependent aspects are treated as sub-transactions over a mixed set of nodes. The set of affected nodes having dependent causes affected nodes to dependent on each other, causing the overhead to be higher.

Overall, based on our experiments, we can conclude that the runtime overhead of COF is acceptable; with each threat aspect capable of being independently woven each threat aspect can be individually deployed based on the required reconfiguration context, thus significantly reducing the overhead compared to all threat aspects being deployed.
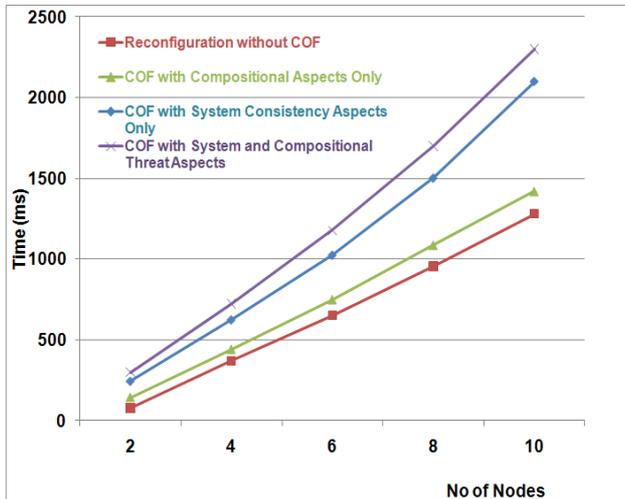

Figure 8. Overhead of reconfiguration using COF in AO-OpenCom

6. Related Work

Few AO middleware platforms have addressed the challenges of performing consistent dynamic reconfiguration. DyMac [10], and CAM/DAOP [14] are prominent examples of distributed AOP platforms that have no support for dynamic reconfiguration. Other prominent platforms such as Spring AOP [1] and FAC [13] do support reconfiguration, but do not support distribution; these systems have not needed to consider strong consistency mechanisms as reconfiguration is considerably simpler when confined to a single node.

JAC [12] is an early example of a distributed platform that supports dynamic reconfiguration. However, this support involves only the reconfiguration of advices at individual join points and provides no support for distributed consistency management.

AWED [3] supports dynamic weaving of aspects using the DJAsCo [20] distributed AOP architecture. It supports the weaving of stateful distributed aspects, and through the use of a consistency protocol ensures that whenever an aspect is woven at a specific host, mirrors are also woven at other involved hosts. However, AWED do not consider any other consistency threats as discussed in the our proposed solution.

ReflexD [18] also supports dynamic weaving/unweaving of mirrored aspects, and uses a framework to provide system-wide consistency. However, as in AWED ReflexD aspects exist only as mirrored aspects although unlike AWED, ReflexD ensures that whenever an aspect is changed the corresponding remote copies are synchronised. But again, the consistency mechanisms provided do not generalise to the extent of our proposal.

Finally, DyReS [19] is an AO middleware framework developed on top of JBOSS dynamic AOP [4] and Spring AOP [1] that provides consistent dynamic reconfiguration in a more sophisticated manner than the systems reviewed above. More specifically, DyReS uses a coordination protocol that allows aspects to be dynamically added and removed in a consistent manner by achieving

quiescence. The protocol is based on two synchronisation primitives: *wait* blocks the ongoing reconfiguration process until it gets a notify message from a specified node; and *notify* sends a synchronisation message to a specified node. Although this approach supports a degree of generality (i.e. it is portable over multiple underlying platforms), it again does not generalise to a wider set of consistency threats. For example, when deployed in a wireless network environment there is no way to address the possibility of lost or reordered synchronisation messages or other system enviroment threats as in our approach. Furthermore, compositional threats are not addressed in DyReS. Our approach is more flexible, allowing different consensus and consistency protocols to be chosen based on the required reconfiguration, the current environment, and the wide range of threats that are posed.

7. Conclusion and Future Work

In this paper we have identified a number of important threats to maintaining the consistency of distributed reconfiguration operations in AO middleware environments. We believe these threats to be representative of the type of threats that should be considered by all dynamic AOP platforms. More specifically, we have presented the AO-OpenCom platform which supports the composition and reconfiguration of distributed aspects, and an associated distributed consistency framework called COF that ensures that all of the identified threats are handled in a transparent manner. COF has the following important benefits. First, it is *simple and elegant* in that it uses aspect composition to deploy these consistency mechanisms. Second, it is *flexible and configurable* in that appropriate threat aspects can be dynamically woven and unwoven according to the types of threat and environmental conditions currently pertaining. Third, it is *inherently extensible* in that new threat aspects can be developed and woven into the system at appropriate join points as and when new threats are identified. Fourth, it achieves the maintenance of consistency with a *reasonable overhead* compared to unsafe reconfiguration.

There are several research directions that we would like to investigate in the future. First, we are currently working on performance optimisations to reduce reconfiguration overheads through the use of multiple (slave) Configurators in cases where a reconfiguration needs to be carried out on a large number of nodes. This should reduce the overheads identified in Section 5 to something closer to constant time. Second, we will investigate the potential for embedding our approach in a self-managing, autonomic environment. Finally, we plan to integrate our framework with appropriate modelling tools which can support the developer in designing, evaluating and validating complex aspect reconfigurations before they are deployed into a distributed system.

References

[1] Spring website. http://www.springframework.org/.

[2] Beloued, A., Gilliot, J.M., Segarra, M.T., Andre, F. "Dynamic data replication and consistency in mobile environments", In Proceeding of the 2nd doctoral symposium on Middleware, ACM, NY, 2005.

[3] Benavides, L., Sudholt, M., Vanderperren, W., et al., "Explicitly distributed AOP using AWED", In Proceeding 5th International Proceeding Conference on Aspect Oriented Software Development, Bonn, Germany, March 2006.

[4] Burke, B., "JBoss AOP Tutorial", 3rd Conference on Aspect Oriented Software Development, Lancaster UK, 2004.

[5] Coulson, G. Blair, G., Grace, P, Taiani, F., Joolia, A., Lee, L., Ueyama, J., Sivaharan, T., "A Generic Component Model for Building Systems Software", ACM Transactions on Computer Systems, TOCS, 2008.

[6] Demers, A., Petersen, K., Spreitzer, M., Terry, D., Theimer, M., Welch, B., "The bayou architecture: Support for data sharing among mobile users." In Proceedings IEEE Workshop on Mobile Computing, pages 2-7, 1994.

[7] Garcia, H., Salem, K., "Sagas", ACM Conference on Management of Data, 1987.

[8] Grace, P., Coulson, G., Blair, G., Porter, B., "A Distributed Architecture Meta Model for Self-Managed Middleware", In Proceeding 5th Workshop on Adaptive & Reflective Middleware, 2006.

[9] Janssens, N., Joosen, W., Verbaeten, P., "NeCoMan: middleware for safe distributed-service adaptation in programmable networks", In IEEE Distributed Systems Online, 2005.

[10] Lagaisse, B., Joosen W., "True and Transparent Distributed Composition of Aspect-Components", In Proceeding Middleware Conference, LNCS 4290, Melbourne, 2006.

[11] Minson, R., Theodoropoulos, G., "Adaptive Support of Range Queries via Push-Pull Algorithms", 21st Workshop on Principles of Advanced and Distributed Simulation, 2007.

[12] Pawlak, R., Senturier, L., Duchien, L., Florin G., "JAC: A Flexible Solution for AOP in Java". In Proceeding 3rd International Conference on Metalevel Architectures and Seperation of Crosscutting Concerns, 2001.

[13] Pessemier, N., Seinturier, L., Duchien L., Coupaye, T., "A component-based and aspect-oriented model for software evolution", International Journal of Computer Applications in Technology, Volume 31, Number 1-2, 2008.

[14] Pinto, M., Fuentes, L., Troya, J.M., "A Component And Aspect based Dynamic Platform". The Computer Journal, 2005.

[15] Satyanarayanan, M., "Coda: A highly available system for a distributed workstation environment." IEEE Trans. Computing, 39(4) pg. 447-459, 1990.

[16] Surajbali, B., Coulson, C., Greenwood, P., and Grace, P. "Augmenting reflective middleware with an aspect orientation support layer. In Proceeding 6th Workshop Adaptive and Reflective Middleware, 2007.

[17] Surajbali, B., Grace, P. and Coulson, G. 2009. A Semantic Composition Model to Preserve (Re)Configuration Consistency in Aspect Oriented Middleware. In Proc. 8th Workshop on Adaptive and Reflective Middleware. 2009.

[18] Tanter, E., Toledo, R., "A Versatile Kernel for Distributed AOP". In Proceeding International Conference on Distributed Applications and Interoperable Systems, June 2006.

[19] Truyen, E., Janssens N., Sanen, F., Joosen, W., "Support for distributed adaptations in aspect-oriented middleware". In Proceeding of the 7th International Conference on Aspect Oriented Software Development, April 2008.

[20] Vanderperren, W., Suvee, D, Wydaeghe, B., Jonckers, V., "PacoSuite and JAsCo: A visual component composition environment with advanced aspect separation features", Conference on Fundamental Approaches to Software Engineering Poland, 2003.