Fine-grained Dynamic Adaptation of Distributed Components*

Frédéric Peschanski[1], Jean-Pierre Briot[2], and Akinori Yonezawa[1]

[1] University of Tokyo {pesch,yonezawa}@yl.is.s.u-tokyo.ac.jp
[2] Laboratoire d'Informatique de Paris 6 Jean-Pierre.Briot@lip6.fr

Abstract. Dynamic adaptability of distributed components, nowadays scarcely supported, should become a basic principle of future middleware platforms. While most related work envisage somewhat large software reconfigurations, we explore in this paper fine-grained adaptations which intervene within component boundaries. Our experiments are conducted in the framework of the Comet middleware. Dynamic adaptability is supported in Comet through distributed protocols that can be applied at runtime. These protocols may locally denote intrusive modifications which are abstracted through the notion of role. Functional roles are used to describe all-purpose adaptations. We use hook roles as wrappers around existing functionalities. Finally, filter roles interfere with the communication layer. The expressiveness of these complementary abstractions are illustrated in various examples involving non-trivial system adaptations for distributed debugging and communication flow synchronization. A preliminary but promising quantitative evaluation of our adaptation engine under real-world conditions is proposed. We also discuss the difficult but crucial issue of verifying such dynamic adaptations in terms of type, access and security contracts.

Key words: Dynamic Adaptability, Component-based Middleware, Fine-grain adaptations, Event-based Asynchronous Communications, Strong typing

1 Introduction

The *dynamic adaptability* of computer systems —their ability to be modified at runtime— represents nowadays a prominent research topic [5].

In mainstream middleware platforms such as CORBA or RMI, introducing a new distributed service (i.e. *common service*) in a running application represents a fairly complex process. Some components of the system must be updated so that they can participate to the new envisaged interactions. To do so, these components must be stopped as well as all their interacting counterparts. Then, developers may add the code needed to support the new service. Finally, the

* This work is supported by the Japan Society for the Promotion of Science (JSPS) under Research Grant #14-02748.

application must be deployed again. But in many recent systems and most especially Internet-based applications, a *continuous* mode of service delivery is expected from both providers and clients. Put in other words, the components of such systems cannot be stopped so easily. There are also cases where components may not be stopped for safety reasons.

In order to adress these issues, it is our belief that next-generation middleware environments should provide extensive support for dynamic adaptatability. However, the participation for already deployed components to new distributed services in a dynamic manner may involve intrusive functional and control-level changes. This challenges in a fundamental way the well-established middleware environments. Obviously, these mainstream infrastructures lack abstractions and mechanisms to support dynamic changes [4].

Researchers mostly investigated dynamic adaptability by means of software reconfigurations such as component replacements. We adopt in this paper a complementary point of view by focusing on finer-grained dynamic adaptations. To explore this direction, we conceived an experimental middleware platform called Comet [11]. Adaptation requirements are abstracted in Comet through the notion of *protocol*. At the operational level, such protocols can be dynamically applied on already-deployed and running components. The problem in this setting is that the components possess no knowledge about the protocols they are supposed to participate in. In consequence, we have to (1) allow the simple and *independent* description of the implied local adaptations as well as (2) ensure their consistent and efficient operationalization. We address these requirements by abstractions called *roles* which fall in three complementary categories. *Functional roles* support general-purpose adaptations. *Hook roles* are wrappers around existing functionalities that can be plugged-in dynamically. Finally, *filter role* interfere with the communication layer and support behavioral changes. Fine-grain dynamic adaptation is obtained through the runtime assignment of such roles within component boundaries. Kernel-level mechanisms must be proposed in order to verify this profound impact on the system. It is also decisive to evaluate this impact in terms of performance.

The organization of this paper is as follows. Sect. 2 presents an overview of the Comet middleware using a distributed multimedia system as an example. Through this case study, we introduce the important concepts of *component* and *event*, which are ubiquitous in our approach. The fundamental abstractions for dynamic adaptation, namely the *protocols* and *roles*, are then presented in Sect. 3. The foundations of an adaptive service for distributed debugging is presented as an illustration. We propose in Sect. 4 to evaluate our adaptation engine using a somewhat more sophisticated protocol for adaptive event flow synchronization. We then address the difficult but crucial issue of verifying these fine-grained adaptations in Sect. 5. Finally, we present an overview of related work and then conclude the discussion.

2 The Comet middleware

2.1 Principles

As for other distributed event-based systems [8], the Comet middleware is based on the *component/event* dichotomy. While components perform local computations, the data they exchange with each other are described by *typed events*. The main originality of the approach is to enforce the extraction of the coupling relation among components using two fundamental principles : structural and control-level decoupling. By *structural decoupling*, we mean that components are transparently localized and may not reference directly other components. The coupling relation is expressed using typed connections that are established at runtime. The type information attached to connections completely capture the routing semantics. While we do not discuss this feature in the present paper, we show in [12] that this added to the introduction of a proper subtyping relation results in a powerful *type-based multicast* communication model. *Control-level decoupling* between components relies on *asynchronous* communications. When a component emits an event, there is no impact on its local control-flow. As a consequence of fulfilling these two decoupling principles, it is possible to dynamically change the structure of the applications by adding/removing components and connections. This represents the proper definition of dynamic adaptability in Comet.

The original semantics of the Comet middleware are captured by constructs of the language substrate of the platform, namely Scope. The Scope language is to Comet what is, for example, Java to RMI : the language layer of the middleware environment. Scope programs are compiled to standard java source code[3]. The basic Scope features are discussed more thoroughly in [12].

2.2 Example: Distributed multimedia

The example described in this section is a simple distributed multimedia application. It consists in client components requesting multimedia streams to dedicated servers. Despite its simplicity, this example captures interestingly the client/server semantics expressed in Comet terms.

Components The following definition shows the structure of the multimedia clients we will deploy:

```
component MMClient {
  receive MMEvent; send MMRequest;
  when(MMEvent event) {
    // show the multimedia contents
    show(event); }
  void askServer(MMRequest request) {
    // send a server request
    send(request); } }
```

[3] Source-to-source Scope compilers are also available for Scheme and Common Lisp.

The `receive` declaration and the corresponding `when` construct together define a *reactive block* for type `MMEvent`[4]. Clients *react* by showing the contents of received multimedia events (`show` method). The `askServer` method is used to send a request event to a server. Note that the `send` primitive does not here references any explicit destination and as such follows our structural decoupling principle. In a similar way, we can give definition for servers:

```
component MMServer {
  receive MMRequest; send MMEvent;
  when(MMRequest request) {
    // subclasses refine this method
    doRequest(request);
  }
}
```

```
component VideoServer extends MMServer {
  receive MMRequest; send VideoEvent;
  double _frame_rate; // Frame rate field
  void doRequest(MMRequest req) {
    ... In a loop for the whole video
    send(req.sender, new VideoEvent(...)
    ... } }
```

The general structure of a multimedia server is described by the left definition. Events of type `MMRequest`, when received, denote client requests for multimedia streams. Operational servers must refine the `doRequest` method to generate the corresponding contents. A refinement for video servers is described partially on the right. Such a server will emit a serie of unicast video frames to the requesting client[5]. Of course, there are many details we do not explain here. Note however the declaration of the `_frame_rate` field, which represents as expected the current "speed" of the server; we will refer to this field later on.

Events As explained previously, events represent the data exchanged by components at runtime. These are abstracted through the definition of an *event type*. For example, we can describe video frames using the following event type definition:

```
event VideoEvent is MMEvent {
  slot _contents type DeltaImage;
  slot _serial type int;
  slot _gentime type long;
  VideoEvent(DeltaImage ic, int is, long ig) {
    _contents = ic; _serial = is; _gentime = ig;
  }
  DeltaImage getContents() { return _contents;}
  int getSerial() { return _serial; }
  long getGenTime() { return _gentime; } }
```

The type `VideoEvent` describes differential frames within video streams. In order to support different categories of multimedia contents, we take advantage of the *subtyping relation* among event types. We may for example define subtypes

[4] `Receive`/`when` declarations seems redundant but in fact address different problems: respectively type negociations among component and reaction semantics. For example, one `receive` declaration might refer to multiple `when` constructs.

[5] Events are of course not multicast from servers to clients. We use the `sender` slot of events to identify the request's origin. Since it is a relative reference, we do not break the structural decoupling rule.

for sound and voice streams, as well as synchronization events [13]. There also exists a most-generic event-type called `Event` which is a supertype of all the event types.

Instantiation and connection In order to start a distributed application from the previous definitions, we first have to deploy some client and server components. This is done using a simple `instantiate` primitive whose syntax is:

comp = **instantiate**(*ComponentType*, *location*)

Suppose for example that we have deployed two clients (`mmclient` and `mmclient2` of type `MMclient`) as well as a video server (`vserver` of type `VideoServer`). We then have to connect *dynamically* our components altogether. Analyzing the *component type* is an important preliminary for connection. This type is decomposed into an *input type* (received events) and an *output type* (emitted events). To connect the server to the client, we may use the `MMRequest` since it is emitted by the client and received by the server. This connection establishment is performed using the `connect` primitive as follows[6]:

connect(mmclient, vserver, MMRequest)

Similarly, we can connect back the server to the client for the video frame communications:

connect(vserver, mmclient, VideoEvent)

Here, the connection is granted since we used a subtype for connection: clients can interpret events of type `MMEvent` denoting more general events than just video frames. If we connect our two clients this way, we obtain the architecture depicted on Fig. 1.

2.3 Component internals

The client and server components described in the previous section are in fact higher-order components. If we look inside each of these components, we reveal an internal architecture of sub-components. These inner architectures expose very similar properties if compared to the higher-order ones: explicit coupling relation and event-based communication. They differ only in the fact that sub-components are not distributed and can communicate synchronously as well as asynchronously.

By default, the internal architecture of a component denotes an actor-like behavior with asynchronous semantics [1]. These semantics are captured by the sub-components depicted on Fig. 2. For the sake of simplicity, we do not show the input and output types of the sub-components which is always the most generic `Event` type. First, events are received and queued by the `Receive` and

[6] As we explain in [12], we also propose an inference algorithm to determine possible connection types automatically. Also note that connections are unidirectional, source and destination components are distinguished.

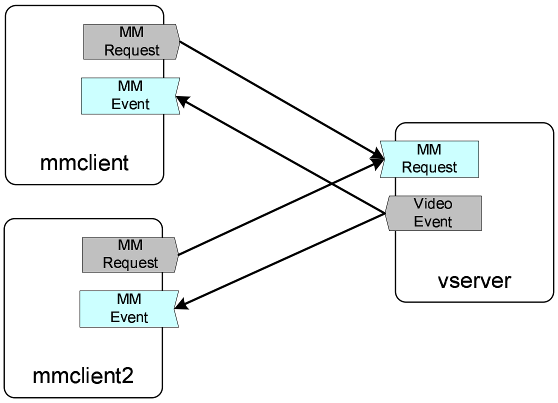


Fig. 1. Distributed multimedia architecture

InQueue sub-components. Concurrently[7], the InFetch component passes the queued events to Exec. The latter will deterministically associates the events to a (most) compatible reactive block (i.e. when construct) for execution. The Send component then handles the potential event emission requests using the implicit type-based multicast algorithm introduced previously. At that level, when an event of type t is sent, a copy of this event is emitted to all destinations compatible with type t: itself or some super-type. Of course, other sub-components can be introduced like the State sub-components which captures the interaction with the component internal state.

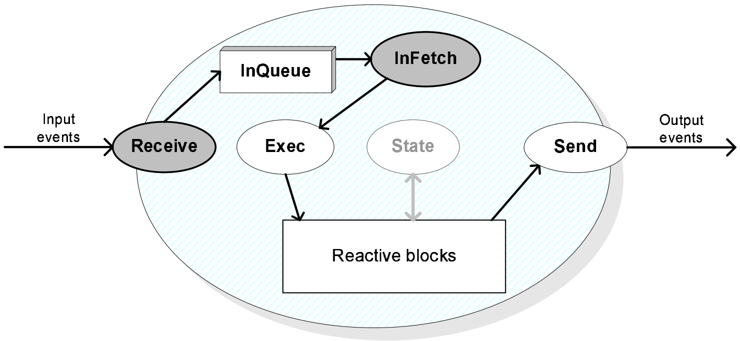


Fig. 2. Behavioral sub-components

As we explained in the previous section, the Comet middleware supports dynamic adaptability by ensuring that distributed components can be added,

[7] The concurrent or *active* components which own a thread of control are emphasized on Fig. 2. The other sub-components use synchronous communications.

removed or interconnected at runtime. By designing the internal architectures of components as if they were regular architectures, we also support the modification of the component internals at runtime by means of adding or removing sub-components. As such, a structural modification at this level will be perceived, from the outside, as a change in the way the associated distributed component is behaving. This represents the operational foundation for the fine-grained adaptations we will discuss in the remainder of the paper.

3 Dynamic adaptation in Comet

Comet components open their internal architecture to support fine-grained adaptations. However, this does not explain how such adaptations are described. In order to capture these conceptual requirements, we propose abstractions called *protocols* and *roles*.

3.1 Protocols and roles

We write protocol definitions to describe dynamic adaptations of running Comet applications. Such a definition is composed of *roles* and *functionalities*, as well as of an optional *internal state*. Functionalities describe the protocol properties that are shared among the components which will participate in it. These can be seen as methods or scripts explaining how to use the protocol. In contrast, roles describe local conventions that each participating component should follow so that the protocol can be used in practice. As a matter of fact, roles capture the essence of the protocols. They describe what will effectively change in the running system when the protocol will be applied. As hinted previously, these roles correspond to sub-components which will be plugged in dynamically within distributed component internals.

Example To illustrate these notions, we propose to design a protocol for on-the-fly inspection of component fields. The definition of this protocol is written as follows:

```
protocol ComponentInspection {
  ComponentRef _inspector; // Internally managed component
  ComponentInspection() { // Creation of the internal component
    _inspector = instantiate_local(InspectorClient);
  }
  Object inspect(ComponentRef component, String fieldname) {
    assign(component, InspectorRole);
    connect(component, _inspector, Ask);
    connect(_inspector, component, Answer);
    return _inspector.getLocalRef().inspect(fieldname); // Perform the inspection
  } }
```

The `inspect` functionality is used to remotely read the value of a component field. The component we would like to inspect is referenced through the component parameter and the name of the field to read is fieldname. This protocol manages an internal state in the form of a component that will act as a client for the inspection process. We need this component because protocols themselves are not able to receive events.

The client for inspection is defined as follows:

```
component InspectorClient {
  receive Answer; send Ask;
  Object inspect(String varname) {
    Answer ans = sendreceive(new Ask(varname));
    return ans.getValue(); } }
```

We use the event type Ask to inspect fields remotely and Answer to denote the replied value. The protocol can invoke the inspect method of this component to perform a remote inspection. The use of the sendreceive primitive allows to send an event and receive a reply in an atomic way[8]. This simplifies greatly the description of client-side computations.

The problem here is that Comet components, such as our multimedia clients and servers, do not support the inspection event types by default. We thus propose to add the necessary code at runtime; and only on the components we would like to remotely inspect. This is done by assigning them dynamically a role for inspection using the assign primitive. But for this we first have to define the inspector role as follows:

```
role InspectorRole {
  receive Ask; send Answer;
  when(Ask ask) {
    send(new Answer(outer.getFieldValue(ask.getFieldName()))); } }
```

As a sub-component, it is not surprising a role looks similar to the definition of a regular component. However, the operational identity of such role (this, as usual) cannot be considered independently from the component it is supposed to be assigned to (referenced as outer). On this example, we define a reactive block for events of type Ask which carry the name of the component's field(s) to inspect. In reply to such requests, the role sends a reply with the value resulting from the inspection. The standard method getFieldValue of the outer component is used to locally perform the inspection. Of course, this intrusive invocation must be precisely controlled. We will discuss this in Sect. 5.

In our terminology, we classify the InspectorRole as a *functional role*. The purpose of such a role is to dynamically add new functionalities —or reactive blocks— to already running components. This represents of course the most versatile form of dynamic adaptation. We will see other forms of adaptation in the following section but let us first describe the use of protocols.

To begin with, we have to create an instance of the protocol as follows:

```
ComponentInspection inspector = new ComponentInspection()
```

[8] Because we rely on multicast semantics, the sendreceive primitive supports atomic distributed *rendez-vous*. This interesting aspect is discussed in [12].

It is then possible to ask for the inspection of some component's internal state such as our previously deployed `vserver` component:

```
println(inspector.inspect(vserver, "_frameRate");
==> [float] 29.9673
```

Here, the value of the internal field `_frameRate` is inspected whereas the inspected component itself has not been tailored at the origin for such fine-grain access. Fig. 3 shows the resulting architecture after inspection.

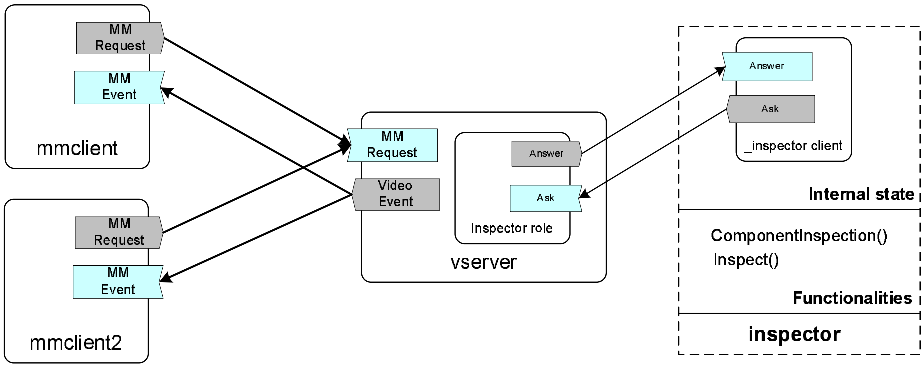


Fig. 3. Dynamic application of the inspection protocol

We can see that an instance of the inspection role has been plugged dynamically within the boundary of the inspected —or dynamically adaptated— component. On Fig. 4, we describe the generic modification involved at the level of the inner architecture.

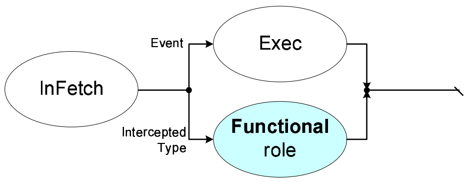


Fig. 4. Functional roles assignments

After the instantiation of the functional role as a sub-component, it is connected from the `InFetch` sub-component for the type `Ask` and also to `Send` so that events of type `Answer` can be sent. This is done as follows:

```
role = instantiate(InspectorRole);
connect(InFetch, role, Ask);
connect(role, Send, Answer);
```

We can see that only structural changes, triggered by standard primitives, are used to realize the dynamic adaptation.

3.2 Other Role Categories

In complement to functional roles, we also provide *prehook* and *posthook* roles to allow the wrapping of *existing* functionalities. And we introduce *filter* roles to be able to interfere with the communication layer of Comet. We argue that these three complementary role categories cover a very large spectrum of dynamic adaptations.

Hook roles Hooks are fine-grained wrappers that can dynamically decorate functionalities. They can be used for example to implement resource management schemes. The way these resources are managed can be modified without touching their functional usage. Another interesting use of a hook is to reflect at the global level things that are happening locally within some running component. To illustrate this, we will develop in this section a protocol for *event interception*. The idea is to inform external components (or roles) when events are received by a given component. This can be used to create replication protocols or, in our case, to complement our debugging techniques with a transparent trace protocol.

In the example of Sect. 2, the events received by the generic multimedia servers are of type `MMRequest`. Suppose that we want to trace such events and also all types that derive from a more generic `Request` type used by all server components. We thus use the latter type for the interception mechanism, defined as a role:

```
role InterceptorRole {
  prehook Request; send Request;
  before (Request request) {
    send(request); // Send a copy
  } }
```

Here, we define a prehook (`prehook/before` reactive block) for events of type `Request`. The only operation performed here is simply to send a copy of the received request to all compatible destinations (i.e. multicast send).

The corresponding fine-grained process for such assignment is depicted on Fig. 5. The hook roles, as sub-components, are plugged as wrappers around reactive blocks. This means that unlike with functional roles, the `Exec` sub-component will here also get the event and process it as usual. If compared to Corba interceptors [9], note that changes are here performed within component boundaries. Hooks may interfere with internal properties such as the component's state.

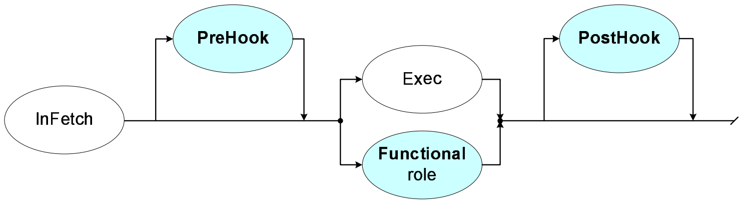


Fig. 5. Hook roles assignments

Filter roles Similarly to prehooks, filters denote computations that are done at event reception time. However, the latter may adjust the delivery semantics for the so-called filtered events. There exist various delivery semantics such as instantaneous delivery (no filter), cancellation and so on. The main use of filter roles, as shown in [12], is to apply content-based filtering protocols to complement the default type-based routing algorithm of the Comet middleware. However, one can find many other interesting uses for filter roles. In this section, we will implement a postponing scheme to support a demand-driven execution mode for components. A generic filter role for such purpose is presented below:

```
role StepFilter {
  filter Event, Step; // Filter all events and Step
  LinkedList _stepList = new LinkedList(10); int _maxStep = 10;

  Event filter (Event event) {
    _stepList.addFirst(event); // Queue the event
    if(_stepList.size()>=_maxStep-1) // Fairness condition
      return _stepList.removeLast(); // Return the oldest event
    else // Step-mode event-flow
      return null; // Cancel the current delivery
    }

  Event filter (Step s) {
    return _stepList.removeLast(); // Replaces by the oldest event
  } } }
```

In this example (and once the role has been assigned to some host component), every reception of an event (except `Step`) will be postponed for later delivery. This is performed using a bounded buffer in which we record the received events. To cancel the current delivery, the `filter` reactive block simply has to return a `null` reference. When events of type `Step` are received, then the oldest recorded event is processed instead of the step event. While simple, this algorithm completely changes the behavior of the host component which is not reactive anymore. The use of a bounded buffer (of size `_maxStep`) guarantees that the modification preserves the *liveness* property of event delivery.

Internally, the role is plugged between the `InQueue` and `InFetch` sub-components, as depicted on Fig. 6.

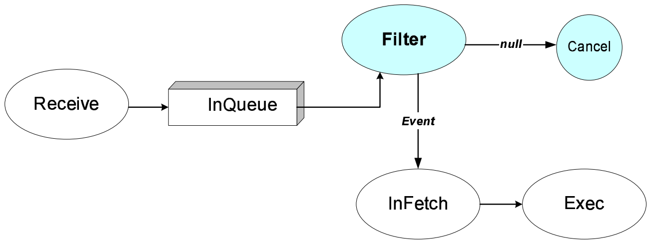


Fig. 6. Filter role assignment

3.3 Put it All Together: Dynamic Distributed Debugging

We may now summarize our mechanisms for distributed debugging. This results in a versatile protocol defined as follows:

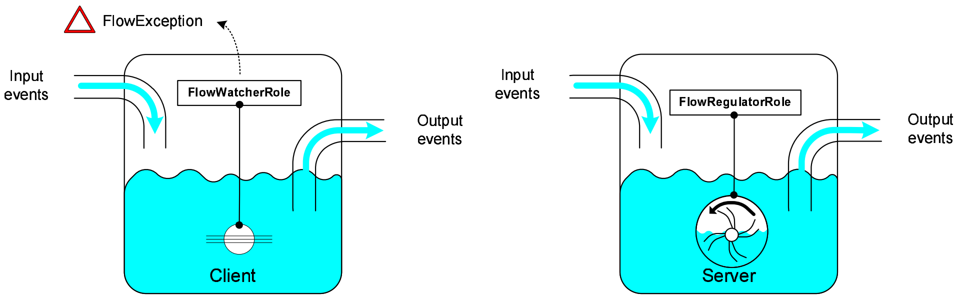
```
protocol DebugProtocol extends ComponentInspection {
  void stepMode(ComponentRef comp) { // Step-mode
    assign(comp, StepFilter); }
  void step(ComponentRef comp) { // Atomic delivery
    send(comp, new Step()); }
  void reactiveMode(ComponentRef comp) { // Standard reactive mode
    unassign(comp, StepFilter); } // Unplug the role
  void trace(ComponentRef comp, EventType type) { // Trace mode
    assign(comp, InterceptorRole);
    connect(comp, _inspector, type); } // Connection to the inherited internal component
  void unTrace(ComponentRef comp, EventType type) { // End of trace
    disconnect(_inspector, comp, type); // Disconnection
    unassign(comp, InterceptorRole); } }
```

Using this protocol, we can trace the execution of a given component (using the `trace` functionality). It is also possible to put this execution in a demand-driven mode. To do so, we first have to invoke the `stepMode` functionality. Then, every time we want to "move" one step further in the execution process, we just have to invoke `step`. The protocol above also inherits from the previous `ComponentInspection` definition. Thus, on-the-fly component inspection is also available. Of course, all the modifications needed to support the debugging facilities, which may slow down the system, can be undone using the disconnection (`disconnect`) and role removal (`unassign`) primitives.

4 Quantitative Evaluation

In the previous sections, we mainly explored the expressiveness of protocol and role constructs. We will now argue in a more quantitative way and measure the impact of their operationalization. To conduct this evaluation, we will use a protocol whose purpose is to synchronize the flow of events among components. Usually, such fine-grained flow synchronization is not needed since most transport protocols ensure some level of fairness. However, if we want to guarantee client-specific quality of service (such as minimal frame-rate for video), then user-level synchronization techniques may be required.

Protocol definition The synchronization scheme we propose is based on flow analysis and regulation mechanisms. For the analysis part, we first define a role for the detection of event flow glitches (see Fig. 7(a)).



(a) Glitch detection

(b) Flow regulation

Fig. 7. Protocol for event flow synchronization

Our criterion for flow analysis is the interval between event reception times. We say that a glitch occurs when the average interval time crosses a given threshold. The Scope definition of this role is given below:

```
role FlowWatcher {
  prehook Event; send FlowException;
  long _event_count=0; long _avg_limit=100; long _avg_time=0;
  before(Event event) {
    _event_count++;
    long current = System.currentTimeMillis();
    _total_time = _total_time + current;
    _avg_time = _total_time / _event_count;
    if(_avg_time > _avg_limit)
      send(new FlowException(_avg_time));
  } }
```

This definition implements a prehook for all the events (generic `Event` type) that are received by the host component. When the hook is performed, the total number of received events (`_event_count` variable) is incremented. Then, the local time is fetched from the operating system and added to the total execution time recored in `_total_time`. We can then compute our criterion (average execution time `_avg_time`) and test if it crosses the threshold `_avg_limit`. If so, an exception (event of type `FlowException`) is raised by the role. This exception encapsulates a proposed rate for regulation.

In order to complete the synchronization algorithm, we also need a role for flow regulation that will take decisions when flow exceptions will be raised (see Fig. 7(b)). A possible solution is to delay the event emissions from the regulated host component through the following definition:

```
role FlowRegulator {
  prehook Event; posthook Event;
  receive FlowException;
  long _start_time; long _end_time; long _rate=1000;
  before(Event event) {
    _start_time = System.currentTimeMillis();
  } after(Event event) {
    _end_time = System.currentTimeMillis();
    if(_end_time - _start_time < _rate)
      Thread.sleep(min_rate - (_end_time - _start_time));
  }
  when(FlowException except) {
    _rate = except.getRate(); // Detected anomaly
  } }
```

Here, we use another prehook to record the time when a given event has been received by the host component. Then, a corresponding posthook computes the execution time for this particular event and sees if a delay is necessary (comparison to the `_rate` variable) so that the next event won't be processed too early; that is, we perform regulation. When a flow exception is received, the event rate is updated to meet the client proposition.

A minimal definition for the whole synchronization protocol can be written as follows:

```
protocol FlowSyncProtocol {
  void sync(ComponentRef server, ComponentRef client) { // Synchronizing
    assign(client, FlowWatcher); // Client watching
    assign(server, FlowRegulator); // Server regulation
    connect(client, server, FlowException); // Protocol connection
  } }
```

The `sync` functionality applies the synchronization algorithm through the assignment of the detection and regulation roles to a couple of host components.

Evaluation The graph depicted on Fig. 8 illustrates the impact of such dynamic adaptation on our multimedia client/server application.

The vertical axis of the graph shows the immediate event handling time on the client (or detection) side. The darkened curve corresponds to the client that will be source of glitches. Another "normal" client is measured by the lighter curve.

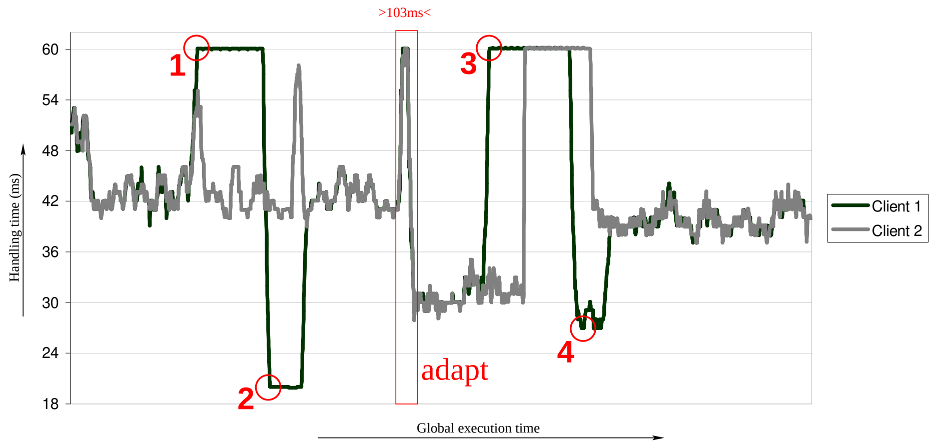


Fig. 8. Dynamic adaptation for event flow synchronization

The (1) mark shows a manually triggered glitch before system adaptation[9]. We can see that the client becomes suddenly less efficient since it cannot handle any received video event in less than 60 ms. We can also note that the second (unmodified) client is taking advantage of the situation (its performance slowly increases). At mark (2), we perform the converse operation: accelerating the instrumented client. We can see now that the second client becomes drastically less efficient. This shows that the event flow between the server and the clients is not synchronized by default.

The central part of the graph (*adapt* label) shows the impact of dynamically applying the synchronization protocol on the instrumented client and on the server. Both the clients are notably affected by this dynamic adaptation. However, we can see on table 1 that the adaptation duration itself, if observed from the client-side, is of about 100 ms and so hardly noticeable from the global point of view. As a matter of fact, this adaptation time is in the same order of magnitude as the average handling time for atomic events. Table 1 shows also that the global adaptation time has a rather limited cost (about 18 times less) if compared to the application deployment time. Mark (3) reproduces the glitch period triggered on the instrumented client. About 150 ms later (adaptation latency), we notice that the second client, albeit not itself adapted, is slowed down to obtain an approximatively equivalent shared quality of service. At mark (4), we try to accelerate our instrumented client but the system now avoids such anomaly and both clients eventually converge at the best throughput from the system's point of view: synchronization is now active.

[9] We instrumented the role and component code to support the manual generation of such "deficiency". In fact, we simply conceived a generic protocol that can be used to slow down any component.

Operation	Duration
Application deployment	9934 ms
Adaptation time (global)	541 ms
Adaptation time (client)	103 ms
Average time for event handling	42 ms

Table 1. Impact of dynamic adaptation

5 Verification contracts

As we can conclude from the preceding discussion, the fine-grained runtime modifications we support in Comet are highly intrusive and thus potentially harmful for the system's integrity. The verification of the dynamic role assignment process is in consequence a critical issue. Role/component contracts are established for verification from three different perspectives: *typing*, *internal access* and *security*.

5.1 Typing contracts

The first level of verification is built upon the versatile type system of the scope language. From a formal point of view, we saw previously that component types where composed of input/output type pairs. A *role type* is composed of such input and output types as well as pre/posthook and filtering types. A typing contract will then establish the conditions so that a role R of type $\langle R_{in}, R_{out}, R_{pre}, R_{post}, R_{filter}\rangle$ may be assigned to a host component C of type $\langle C_{in}, C_{out}\rangle$. What we have to verify is that all the types that are intercepted by the role are compatible[10] with the input types of the host component. In formal terms, we write:

$$\forall t \in R_{pre} \cup R_{post} \cup R_{filter},\ \exists t' \in C_{in},\ t' \sim t$$

The establishment of such typing contracts is performed through *static analysis* of the role and component source code. It is then generated and transmitted at assignment time as an XML document. For example, the interceptor role described in Sect. 3.2 is associated to the following contract:

```
<role type="InterceptorRole">
  <type>
    <prehook>Request</prehook>
    <out>Request</out>
  </type></role>
```

This document, when transmitted as a preamble to the role assignment, is compared to a corresponding host component type contract. In the case of our multimedia server, this contract is:

[10] We define type compatibility between types t and t' through the relation $t \sim t'$, t being a subtype of t' ($t \preceq t'$) or the converse ($t \succeq t'$).

```
<component type="MMServer">
  <type><in>MMRequest</in><out>MMEvent</out></type>
</component>
```

Intuitively, we see that both definitions match since the intercepted type is `Request` which is compatible (as a supertype) with the host input type `MMRequest`.

5.2 Access contracts

The contracts we describe in this section firstly establish the modality for sub-components —most notably the roles— to access the internals of their outer component. In order to allow or disallow a role from accessing parts of the host component internal state and methods, we define two complementary contracts. The first one is associated to the role we try to assign. This role contract must indicate which fields or methods should be accessible. For example, the `InspectorRole` described in Sect. 3.1 proposes the following access contract:

```
<role type="InspectorRole">
  <type> <in>InspectReq</in>
         <out>InspectRep</out></type>
  <access><all-fields mode="read"></access>
</role>
```

Here, we request the access to all fields for reading only. In order to satisfy this request from the host component point of view, we must allow at least access to some fields. In our example, we will only grant access for `_frameRate` which is the one we use for inspection. This is written as follows:

```
<component type="VideoServer">
  <access><field type="float" name="_frameRate" mode="read"></access>
</component>
```

As we can see, the `<field>` discriminates fields using either their type, their name or a combination of both. The access mode can be either `read`, `write` or `readwrite`. It is also possible to access/restrict method invocations and instance creations. For example, the synchronization protocol of Sect. 4 may only be plugged in if the `sleep` method of the standard class `Thread` is accessible. This is requested from the flow regulator side as follows:

```
<role type="FlowRegulator">
  <access> <method class="java.lang.Thread" name="sleep" </access>
</role>
```

By default, methods are requested/granted either as invoked from the `outer` (host component reference) or `static` contexts. On the host component side, access rights are established in the same way. The instance creations are controlled using the same scheme by considering their constructors as methods.

5.3 Security contracts

We employ low-level security rules when both type and access contracts are not enough to prevent unexpected runtime modifications. For example, it seems important to restrict the adaptability features to well-authenticated sources when potentially unsafe mechanisms are employed. In Sect. 4, the use of the `sleep` method is for example a way to slow down a system so that only an authorized person should be able to plug and control the synchronization protocol. This can be done through authentication or domain restriction. The latter would for example be expressed like this:

```
<component type="Dummy">
 <access><method class="java.lang.Thread" name="sleep"/>
  <domain ip="127.0.0.*"/></access></component>
```

Here, we only grant the access rule from domains within a range of IP addresses. In order to implement these low-level security contracts, we integrally rely on the Java security API [14].

6 Related work

Several other researchers have addressed the problem of modifying computer systems at runtime [5]. In the case of middleware environments, studies such as X-RMI [2] focus on large-grain reconfigurations. This allows existing RMI applications to be preserved while introducing dynamic reconfiguration features. Corba interceptors [9] have also been used to support runtime changes in a portable way. Reflective approaches such as [6] show that more intrusive modifications can be obtained at the price of deriving from standards (like with Comet). However, most of these propositions seem to focus on *mechanisms* for dynamic adaptation. In contrast, we think that finding *abstractions* to capture adaptation requirements should be the priority. The Drastic approach [3] seems to fit more closely this vision of dynamic adaptability. It proposes a proper frontier between the abstractions for adaptation and the operational mechanisms that support them. In this work, runtime modifications are abstracted by type contracts among distributed objects. At the operational level, these contracts are matched against physical zones that need to be temporarily frozen (using persitency) for update. Meanwhile, every components of the system outside such zones carry on their computations unaware of the runtime modification in progress. The main difference in our approach is the much finer granularity level of the changes we support. We also focus more on minimizing the impact on the system performances.

The CodA framework [7] once demonstrated that metal-level architectures could be introduced to circumvent the rigidity of most (distributed) object models. We took our inspiration from CodA to design the inner architecture of Comet components. But we (finally) found no need to convey the hardly tractable reflective concepts since both inner and outer architectures use the same fundamental concepts. This makes our work diverge from approaches such as [10] where reflection plays an essential role. Of course, the implementation of the middleware itself relies heavily on reflective mechanisms.

Actor-based languages [1] were also an important source of inspiration. The DIL approach [13], most notably, introduced similar concepts of protocol and role that we use. However, Comet is an event-based middleware relying on typed and multicast communications. This diverges in an important way from actor-based message-passing semantics. We can also note that Comet components may denote multiple internal activities (depending on the number of active sub-components) whereas actor-based languages identify the activity and actor concepts.

7 Conclusion

The Comet middleware we designed and implemented supports a development model which we argue is innovating because it is tailor-made for dynamic adaptability. From a structural point of view, runtime system reconfigurations are made (1) *possible* thanks to the extraction of the coupling relations among components and (2) *controllable* through strong typing. Relying on the same principles but within component boundaries, we were able to support dynamic adaptability at a finer granularity level. This is in our opinion the most objective contribution described in this paper.

Fine-grained adaptations are manipulated at the language level as role and protocol abstractions. We think that such domain-specific abstractions foreshadow tomorrow's *adaptive* middleware environments. The three categories of role we introduce form in our opinion a quite *expressive* model. Functional roles, for example, are particularly versatile since they support incremental and dynamic functionality enhancements. We saw how prehooks or posthooks could even change the way a particular functionality is handled. Filter roles go even further and allow incremental changes in the way events are communicated, using content-based analysis. Our expressiveness argument represents of course a more subjective contribution. But we think our example protocols are particularly illustrative in this respect. They denote less than trivial changes in a quite concise manner.

Moreover, we began to evaluate the cost of such dynamic adaptations which deeply modify the system semantics. Despite the prototype status of our implementation, these evaluations reveal promising results. Of course, this evaluation is relative to the performances of the Comet middleware itself. We are currently conducting an extensive benchmark to optimize its prototype implementation so that we may compare to industrial-strength approaches (most notably Java RMI).

In the mean time, it is very important to keep in mind that the *arbitrary* modification of a distributed system at runtime remains a mostly open research topic. Of course, type-related negotiations, if useful, are not enough to verify such intrusive changes. We discussed the complementary access and security contracts as "compensations" here. The security layer seems to be yet the only operational mechanism at our disposal to envisage the support of dynamic adaptation in today's real-world applications. But we find this quite unsatisfactory: more solid foundations are in our opinion needed to address the problem of safety in the presence of dynamic adaptations. In this perspective, we currently address the problem of describing properly the semantics of the language substrate of our middleware. We hope this would then ease the definition of properties regarding dynamic adaptations as well as their verification using, for example, theorem-proving or model checking techniques.

Acknowledgements

We would like to thank the anonymous reviewers for their useful comments. A special thanks also to Reynald Affeldt who carefully read and commented the paper.

References

1. G. Agha. *Actors : A Model of Concurrent Computation in Distributed Systems*. Series in Artificial Intelligence. MIT Press, 1986.
2. X. Chen. Extending rmi to support dynamic reconfiguration of distributed systems. In *Proceedings of ICDCS'02*. IEEE, July 2002.
3. H. Evans and P. Dickman. Zones, Contracts and Absorbing Changes: An Approach to Software Evolution. In *Proceedings of OOPSLA'99*. ACM Press, November 1999.
4. K. Geihs. Middleware challenges ahead. *IEEE Computer*, 34(6):24–31, June 2001.
5. X. Liu and H. Yang, editors. *International Symposium on Principles of Software Evolution (ISPSE 2000), Kanazawa, Japan*. IEEE computer society, November 2000.
6. J. Malenfant, M.-T. Segarra, and F. André. Dynamic adaptability: The molène experiment. In *Proceedings of Reflection 2001*, volume LNCS 2192. Springer Verlag, September 2001.
7. J. McAffer. *Metalevel Programming with Coda*. In *Proceedings of the European Conference on Object-Oriented Computing (ECOOP)*, LNCS 952, pages 190–214. Springer Verlag, August 1995.
8. R. Meyer. State of the art of distributed event models. Technical report, University of Dublin, Trinity College, 2000.
9. Object Management Group. *Common Object Request Broker Architecture (CORBA) 2.6 Specifications*. `http://www.corba.org`, 2001.
10. N. Parlavantzas, G. Coulson, M. Clarke, and G. Blair. Towards a reflective component-based middleware architecture. In *Proceedings of Workshop on Reflection and Metalevel Architectures*, June 2000.
11. F. Peschanski. A reflective middleware architecture for adaptive, component-based distributed systems. *IEEE DS Online*, 1(7), 2001.
12. F. Peschanski. A versatile event-based communication model for generic distributed interactions. In *Proceedings of DEBS'01 (ICDCS International Workshop on Distributed Event-based Systems)*. IEEE, July 2002.
13. D. C. Sturman. *Modular Specification of Interaction Policies in Distributed Computing*. PhD thesis, University of Illinois at Urbana-Champaign, May 1996.
14. Sun Microsystems. *Java Security Model*. `http://java.sun.com/security`, 2001.