Comet

A Component-based Reflective Architecture for Concurrent and Distributed Programming.

Frédéric Peschanski *

Laboratoire d'Informatique de Paris 6 (LIP6)
Frederic.Peschanski@lip6.fr

Abstract. In order to build highly flexible and dynamic distributed systems, our Comet architecture implements components whose both control and functionalities are expressed independently. Moreover, we introduce the Comet roles and protocols which authorize the dynamic modifications of both components and architectures. The Comet/J framework implements these concepts in an object-oriented approach.

1 Introduction

Today's distributed systems are increasingly complex and dynamic. This drives the needs for both decomposition and adaptability, with issues in both software engineering and system fields. Component-based software architectures [5] allow decomposition and recomposition of parts, both statically and dynamically. While preserving the neutral trait as found in object models, components avoid explicit external references within functional code (potentially strong coupling). Reflective architectures [3], on their side, provide the separation of concerns and the possible adaptation of control strategies.

Important advancements have already been achieved in these different topics but some issues need to be further addressed : interaction of different control policies, dynamic customizations of isolated or grouped entities (e.g. dynamic application of fault-tolerance protocols) and consistency checking of these adaptations. In order to investigate these issues, we propose an architectural model, named Comet, building on some existing architectures (mainly Coda [9] and DIL [11]) and providing novel features (dynamic assignment of arbitrary functionalities as well as checking both statically and dynamically the consistency of these assignments).

Similarly to Rapide [8], Comet is based on the notion of typed component, immersed in an event-based communication system. This way, we are able to target various kind of distributed systems, without making specific assumptions about the underlying computation model neither the computer language. A Java-based instantiation of Comet (named Comet/J) has been implemented and will be used for describing examples of use.

* Many thanks to Jean-Pierre Briot for reviewing the successive versions of this paper

Section 2 of this paper describes with more details the component model we are using. The associated meta-level architecture is then presented in section 3. The Comet protocols, autonomous software entities, are introduced in section 4. Section 5 establishes a panorama of related works. We finally conclude our discussion with short and longer term prospects in section 6.

2 Component model

Within Comet, components correspond to first-level software entities, basic building blocks of distributed architectures. Connections allow components to be linked together, through connectors. We propose in Comet an event-based approach to model connectors and connections. Connectors are represented by input and output event descriptors. A connection then corresponds to a link between a source component and a destination component, according to a given event type (see Fig.1).

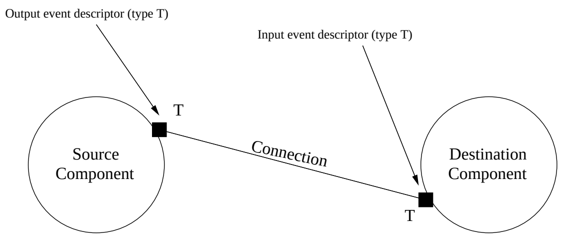


Fig. 1. The event-based model of the Comet architecture

Comet components can be represented using traditional object-oriented environments. Let's consider for example a simple Comet/J component :

```
class Server extends comet.DefaultComponent
implements ClientRequestEventReceiver,
           ServerReplyEventSender {
  void ClientRequest(ClientRequestEvent cre) {
    String result = manageRequest(cre);
    ServerReplyEvent sre = new ServerReplyEvent(result);
    sendEvent(sre);
  }
}
```

The purpose of this server component is to handle client requests (`ClientRequestEvent` events) and then generate answers (`ServerReplyEvent` events). It is important to point out the implicit nature of communications through the `sendEvent` primitive. External references are handled at the connection level (see Sect.4).

Events are coupled by a sub-typing relation allowing subsumption [1]. This property authorizes the use of an event of type T' within a context where an event of type T, super-type of T', is expected. The main purpose of the subsumption is to make more flexible the coupling relation between components.

The component typing relation, defined in an event-based point of view by the pair $(E, S)^1$, is also extended to a sub-typing relation. This way, an event of type T', sub-type of T, can be used as an input event for a component C_1 of type $(E = T \cup \tau_1, S)$. A component C_2, of type $(E, S = T'' \cup \tau_2)$, is then partially compatible with C_1 if T'' is a sub-type of T. In other terms, this means that it is possible to establish a connection associated to the type T'' between components C_2 and C_1.

3 Semantic levels

Comet components introduce three causally linked semantic levels. At the **base level**, developers can express the static functional properties of the component. Local control functionalities are (at least partially) reified at the **meta-behaviour level**. Finally, the **meta-role level** represents the adaptive functionalities layer.

3.1 Meta-behaviour level

In order to design the meta-behaviour level, we considered the set of functionalities related to the way events are processed inside a component (see Fig.2). The idea is to apply the CodA [9] approach to a component-based architecture by reifing each control functionality as a metaobject. We nevertheless propose to extend CodA by firstly allowing asynchronous emissions of events. Further specializations can be obtained by splitting the queuing/fetching phases. We describe below the role of each canonical meta-object and a set of possible specializations :

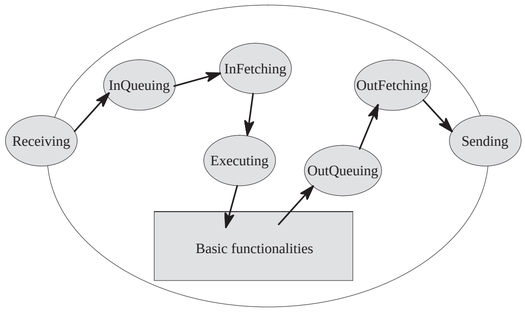


Fig. 2. Meta-behaviour level

[1] (input event types, output event types) pair.

- **Receiving** : Receiving the input events.
  Specializations : adaptations to various communication semantics.
- **InQueuing** : Queuing the received events.
  Specializations : complex input queuing system, priority queues.
- **InFetching** : Extracting the incoming events from the input queuing system
  Specializations : event-based synchronization, complex choice heuristic, management of special events (synchronous ones, atomic sequences, etc).
- **Executing** : Executing the corresponding functionality.
  Specializations : method-based synchronization schemes (guards, synchronization constraints, etc). Execution semantics : multiple activities, scheduling properties (deadlines).
- **OutQueuing** : Queuing the emitted events.
  Specializations : concurrent emissions, causal reordering system[2].
- **OutFetching** : Extracting the output events.
  Specializations : complex choice heuristic, marking time in order to satisfy a stability property of the system.
- **Sending** : Sending the output events through the communication layer.
  Specializations : adaptations to various communication semantics.

Similarly to CodA, the introduction of new meta-objects in the model is possible, depending of the systems needs. For example, the integration of a *Meta-State* meta-object, managing the internal state access, could be useful in the case of migration[3] or transactions. The composition of these various meta-objects builds up a behaviour model C. Developers can build a lot of different behaviour models by composing complementary meta-objects.

3.2 Meta-role level

The adaptive functionalities are managed by the meta-role level. Roughly, functionalities and data are combined inside a role to attribute a new property to one or several components.

A Comet role is represented by a meta-level component, very similar to a base level one, as we can see in the Comet/J example below :

```
public class RestoreServerRole extends comet.MetaRole
implements comet.RestoreServerEventReceiver,
           comet.ValidateReplicaEventSender {
  public bool checkBase() {....}
  public void RestoreServer(RestoreServerEvent rse) {
    _base.restore(rse);
    sendEvent(new ValidateReplicaEvent());
  }
}
```

[2] The loose of causal informations can result from the use of a complex input queuing system.

[3] During the migration phase, it is interesting to optimize the component availability by allowing remote access to its internal state. These accesses must then be serialized.

Once assigned, this role adds a restore functionality to a server, from a set of saved request on a replica. When the restoration process is done, validation event are sent to the concerned replica(s).

The basic mechanism involved at the meta-role level is the dynamic assignment of a role to an existing component. In order to validate this assignment, two predicates systems must be implemented. First, the V_r system allows the role to check the component properties. This mechanism is provided in Comet/J through the `MetaRole` class and its special method `checkBase`. If this method returns `false`, then the assignment is refused by the role. The complementary V_C system allows the component to check the role properties. This system is associated to the behaviour model C and managed by the meta-behaviour level. This permits for example the management of conflicting roles[4] by disallowing their composition or introducing an order relation.

4 The Comet Protocols

Assigned roles are managed by components but roles[5] are expressed within protocol definitions. Following the DIL approach [11], protocols are introduced as autonomous software entities within Comet. However, unlike DIL, the Comet protocols aren't necessarily related to interactions such as in an activation protocol whose only purpose is to assign an activation role[6] to isolated components.

The needs in term of adaptability require at the same time the possibility to modify the components dynamically but also to consider the system topology as a dynamic entity. Thus, the protocols also make possible the management of dynamic connections.

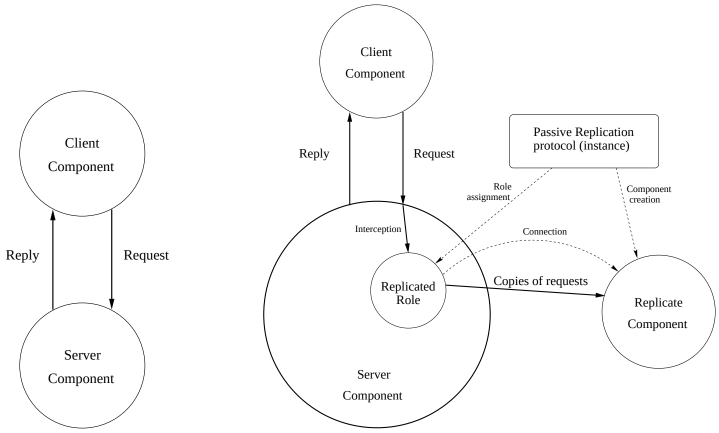
More pragmatically, a Comet protocol defines a set of protocol functionalities (e.g. adding a replica) and a set of protocol roles, which can be assigned dynamically, using the `assign` primitive (e.g. replicated role). Protocols may also maintain an optional internal state (e.g. replicas list management),

Let's take the example of a minimalist client/server application. Suppose that during some critical phases, it is necessary to dynamically extend this application with a passive replication protocol. The underlying idea of this protocol is the creation of a (or a set of) replica(s) and the dynamic assignment of a replicated role to the server. This process is depicted in figure 3.

The replication protocol implements the `ReplicatedRole` which is coded as follows within Comet/J :

```
public class ReplicatedRole extends comet.MetaRole
implements comet.ClientRequestEventPreHook,
           comet.ReplicationEventSender {
  public void BeforeClientRequest(ClientRequestEvent cre) {
    ReplicationEvent re = new ReplicationEvent(cre);
```

[4] Roles which intercept or manage the same event types.
[5] *Roles* may be associated to classes and *assigned roles* to instances
[6] Role that handles the `ActivateEvent` type



(a) Before (b) After instantiation

Fig. 3. Dynamic instantiation of a passive replication protocol

```
    sendEvent(re);
  }
}
```

This role doesn't introduce any new event but intercepts the requests received from the server[7] and sends a copy of them to the replica(s). We are now able to express in an object-oriented way, our generic passive replication protocol :

```
public class ReplicationProtocol extends comet.Protocol {
  public String createReplica(Address machine) {
    String id = upload("Replica", machine);
    instantiate(id);
    return id;
  }

  public void replicate(String id_server, String id_replica) {
    assign("ReplicatedRole", id_server);
    connect("ReplicationEvent",id_server, id_replica);
  }
}
```

Once instantiated, this protocol allows the creation of potential replica(s) (`createReplica` method). It is then possible to connect the replica(s) to some server(s) (`replicate` method). We can see, according to this example, that the protocol

[7] Using the `ClientRequestEventPreHook` interceptor, associated to the `BeforeClientRequest` method.

notion is extended, in Comet, to the deployment concerns. Thus, protocols also allow to send the code of components on remote computers (`upload`), instantiate remotely or locally new components (`instantiate`) and establish connections (`connect`). These basic protocol primitives can be extended, in the case of some mobile components models, by a migration primitive (`migrate`).

The independence feature of the protocols and roles they're managing facilitates the creation of reusable libraries. For example, the replication protocol can extend every application that provides a server component, accepting `ClientRequestEvent` events[8].

5 Related works

Comet components are, as well as actors [2], autonomous software entities which communicate asynchronously. Major differences are however notable, in particular at the level of granularity, although component models can be built on top of actors. Moreover, the protocol and role concepts we are using were introduced by the actor-based DIL framework. The Comet components are also comparable to the JavaBeans [12] but the latter do not address, or little, the issues related to concurrency and distribution. Because of such fundamental divergence, we were constrained to propose an alternative model.

The role concept has been introduced in a large number of fields. The most notable one is the Role modelling theme [10]. Because we are not focused on modelling generic software but more on implementing distributed systems, we see roles as more functional than structural entities. Anyway, further work should be done to bring closer both concepts.

Unlike other reflective middleware architectures, such as OpenCorba [7], the underlying communication system is somewhat reduced in Comet. We think that in the case of highly dynamic distributed systems, many control properties should be associated to the software entities and not delegated to the so-called middleware. This allows components with very different local control policies to cooperate. The choice of an event-based communication semantic is also closer to the asynchronous nature of distributed systems than object-based synchronous invocations.

Aspect Oriented Programming [6] identifies clearly the issues while designing complex systems. The needs in term of separation of concern are precisely presented : components to decompose a problem and implement a solution, aspects to explicit partially the control over components. By introducing some code generators instead of our applicative approach, Comet/J would be similar to an AOP framework like D [4] which does not address the issues related to dynamic adaptability.

[8] The sub-typing relation allows sub-types of `ClientRequestEvent` to be used.

6 Conclusion

Faced with both distributed systems and software engineering concerns, we propose, through our Comet architecture, an hybrid approach at once reflective and based on components. The semantic levels we introduced allow the separation of concerns between control and functionalities. This makes easier the elaboration of complex distributed systems without constraining the development model. The Comet roles and protocols also cover a large set of needs. They allow the dynamic adaptations of components and architectures, in a controlled way.

Using a preliminary implementation of this framework, we were able to design some simple distributed systems : adaptive replication, clocks synchronization, etc. We plan to validate our approach with a few application prototypes : multimedia systems, workflow-based applications and multi-agent architectures.

In a longer term, we are considering the design of some languages and tools around the Comet concepts. These tools, related to new methodological concepts, should offer an interesting framework for the design of large-scale adaptive applications, in particular within the multi-agent systems community (applications in the fields of collective robotics, crisis management, etc).

References

1. Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.
2. Gul Agha. *Actors : A Model of Concurrent Computation in Distributed Systems*. Series in Artificial Intelligence. MIT Press, 1986.
3. Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Löhr. Concurrency and Distribution in Object-Oriented Programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.
4. Cristina Videira Lopes and Gregor Kiczales. D : A Language Framework for Distributed Programming. Technical report, Xerox Palo Alto Research Center, February 1997.
5. David Garlan and Mary Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing Company, 1993.
6. Gregor Kiczales et al. Aspect-Oriented Programming. In *Proceedings of ECOOP'97*, LNCS 1241. Springer Verlag, June 1997.
7. Thomas Ledoux. OpenCorba: a Reflective Open Broker. In *Proceedings of Reflection'99*. Springer Verlag, July 1999.
8. David C. Luckham. Rapide : A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. *DIMACS Partial Order Methods Workshop*, IV, July 1996.
9. Jeff McAffer. Metalevel Programming with Coda. In *Proceedings of ECOOP'95*, LNCS 952, pages 190–214. Springer Verlag, August 1995.
10. Dirk Rhiehle and Thomas Gross. Role Model Based Framework Design and Integration. In *Proceedings of OOPSLA'98*, pages 117–133. ACM Press, 1998.
11. Daniel C. Sturman. *Modular Specification of Interaction Policies in Distributed Computing*. PhD thesis, University of Illinois at Urbana-Champaign, May 1996.
12. Sun Microsystems. *JavaBeans 1.01 specification*. `http://java.sun.com/beans`, 1998.