Position Paper : A Generic Model for Distributed Interactions

Frédéric Peschanski
Laboratoire d'Informatique de Paris 6

Frederic.Peschanski@lip6.fr

March 24, 2000

Abstract

We propose in this position paper to model different interaction semantics using a component-based and reflective architecture called Comet. Building on a canonical asynchronous communication model, we propose to implement synchronous and future communications by the simple modification of the component behaviour, through a meta-level architecture. The Publish/Subscribe interaction style is then presented as a more complex change involving structural composition through the use of protocols.

1 Introduction

Semantics of communication models developed in research and industrial work on distributed systems are of primary importance. However, this property of predominance is often associated to immutability. For example, in the Corba Object Model [5], the components of the system communicate through synchronous invocations of methods because the initial aim of the OMG was to mimic the standard behaviour of (not distributed) objects. This synchronous behaviour is not easy to conciliate with the asynchronous nature of distributed systems. Of course, other semantics are proposed though Corba Services (event channels, publish/subscribe, transactions and so on) but they are provided using utility libraries and "side effects". These can hardly be considered as parts of the model. The research about decoupled publish/subscribe communication models [1] follows the same guidelines : the communication scheme is provided "as it is", without any relation to lower-level models. That is, adhoc solutions are often prefered to more demanding bottom-up approaches.

We propose in the present work to model different communication schemes using our Comet Architecture [6] which is briefly introduced in section 2. In order to design these communication models, we pointed out three architectural needs :

- **new linguistic constructs** : new interaction styles must be provided with clear linguistic constructs[1] so that developers can easily manage them.
- **behavioural changes** : new interaction styles may require some changes in the behaviour of the components involved.
- **structural changes** : composition of lower-level models may be used to model higher-level interaction styles.

Using slight behavioural changes to the default asynchronous behaviour (section 3.1), we can implement synchronous (section 3.2) and Future (section 3.3) communication models. We can then go further the "simple" modeling of communication scheme and implement higher level interaction styles like publish/subscribe distributed programming semantics (section 4). Section 5 finally provides a conclusion to this paper.

2 The Comet Architecture

The Comet Architecture implements components as first-level software entities. They can communicate with each other by sending asynchronous events. The type of a component explains, through the notion of connector, which kind of event it is able to receive (input connectors) or emit (output connectors). Before being able to exchange an event of a certain type, a correctly typed connection[2] must be established between the interested components. Figure 1 depicts the

[1] When speaking about linguistic constructs, both real language level constructs or library-based APIs may be considered.

[2] We introduce here a subtyping refinement to provide a flexible coupling relation.

general principles of the structural composition model proposed in Comet. A more detailed presentation can be found in [6].

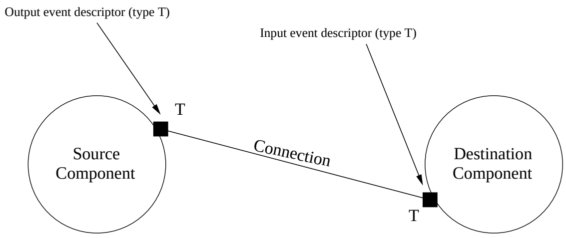


Figure 1: Structural Composition in Comet

The behaviour of a component is defined by three different and causally linked semantic levels. At the base level, developers 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 allows components to participate to new interactions (or protocols) at runtime[3].

2.1 Meta-behaviour level architecture

The meta-behaviour level, as seen on figure 2, reifies as metaobjects the functions related to the processing of the events inside a component. This extends the concepts introduced by the CodA meta-level architecture [4].

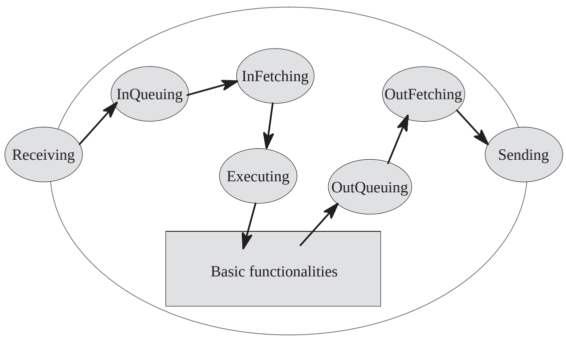


Figure 2: Meta-behaviour level architecture

First, events are received and queued (asynchronous input), then fetched from the queuing system. Finally, the corresponding functionality is executed. If event emissions are initiated at the base level, then these events are queued (asynchronous output), fetched and actually sent. This fine grain decomposition provides a large spectrum of possible specializations (synchronization schemes, multiple queues/activities and so on). By changing (statically) the behaviour model, we can build different local control strategies for different kind of components and make these components able to interact with each other. We can also reuse separately the components functionalities and their local control strategies.

2.2 Roles and Protocols

Two components can communicate with each other for a certain kind of event if they are compatible regarding this event and if a connection is established between these two components. But this notion of connection is, in the case of Comet, highly dynamic. They are established through the application of distributed protocols, using a distributed interaction language (similar to [7]). These protocols are applied (in fact, there is no other possibility) at runtime. The simplest case is when the components are correctly typed for the connections established by a specific protocol. That is, the components involved already know how to handle the events managed in the protocol. In Comet, we propose another possibility : to adapt dynamically the components so that they will be able to participate to the protocol even if they were not designed to do so. The idea is to authorize the dynamic adaptation of a component through the assignment of a role which is itself a component-like entity (without a meta-level). The role implements the functionnalities needed to support the events involved in the protocol for every concerned component. The role assignment, established by a protocol, is managed by the components at the meta-role level. This complex mechanism will be the subject of a future paper and is related to the software evolution reseach area [3].

2.3 Building new interaction styles

The Comet Architecture allows the three kind of changes we isolated in order to build new interaction styles :

- **linguistic constructs** : new interaction primitives can be provided by a component model (subtype) or a role definition. The idea is that the component model or the role provide a minimal user-level language which can be extended through specialization. In our Comet/J framework which is a Java-based instantiation of the Comet Architecture, we allow developer to add new primitives by inheriting and modifying the default component class. The new primitive corresponds to a new method in the class. Note that this kind of change is only available for system designers (as opposed to application designers).
- **behavioural changes** : the meta-behaviour level offers several[4] entry points to modify the semantics of the interaction-related control functionalities within components.

[3]In fact, at his level, we're considering generic adaptive functionalities which may not be related to interactions.

[4]There is no real limit here because new kinds of metaobject may be introduced in the model.

- **structural changes** : protocols, once applied, modify the distributed systems so that new interactions are managed. This is done, structurally speaking, mainly by the composition of connections.

The detailed technical issues involved here will be the subject of a more advanced paper, we will restrict our discussion on these general principles in the rest of the paper.

3 Communication models

The first cases we will study concern the canonical communication model implemented in Comet and the way we can modify it to manage other useful communication models. This work is similar to the ABCM/1 reduction of Now and Future Types [2].

3.1 Asynchronous interactions

The asynchronous model is canonical in our approach. We propose to structure every other model as a composition of this one.

The asynchronous behaviour is enforced in the process of emitting events as well as in the reception process. In order to make two components communicating asynchronously, we only have to create a single connection.

At the linguistic level, we only need here a way to send an event like in the following code :

```
send event
--next instruction
```

Semantically, the only important aspect here is that the `--next instruction` block will be executed as soon as the event has been communicated to the output system (OutQueuing metaobject). This process is roughly instantaneous.

The destination of the emission isn't identified within the previous code extract. This is expressed at the protocol level through the `connect` primitive :

```
connect source to dest for type
```

This way, we decoupled the protocol (or interaction) from its participant components, to allow generic protocols to be reused independantly from the components.

3.2 Synchronous interactions

Synchronous interactions represent the standard way of expressing programs in the object world. That is, when we send an event, we want to wait for the reply before continuing the execution. In order to map this synchronous behaviour on our asynchronous model, we can think about composing connections. By composing two connections, we can handle the management of both the request and the reply of the synchronous event. :

```
connect source to dest for request_type
connect dest to source for reply_type
```

At the programmer's level, we want to express that we have to wait for the result of the communication before executing the next instruction. To follow our communication-based terminology, we'll speak about mixing a send and a receive operations[5] :

```
reply = sendreceive request⁶
--next instruction
```

Before executing the `--next instruction` block, we'll first send the request event and wait for the return of the reply.

One possible variant is to simply express we want to wait for a specific event :

```
reply = receive
--next instruction
```

This will block the execution of this code sequence until an event of type compatible with reply will be received.

The composition of two connections is not enough to handle a synchronous interaction, a slight modification of the default behaviour will also be necessary. At the meta-behaviour level, we have to provide a way to suspend execution when we're waiting for an answer. This may be done at the level of the Execution metaobject that manages the base-level activities.

Next, some kind of matching rules must be managed at the level of the Input Fetching metaobject. One typical rule will be :

> *On fetching of Event of Type rtype , then reactivate the waiting activity (by communicating with the Execution metaobject).*

[5]This can be seen as an extension of the object-based notion of synchronism.

[6]In our Comet/J framework, we allow such linguistic extension in a simple object-oriented approach.

The idea is to create a new component class by inheriting from the default one.

In Java, we will write something like this :

```
class SynchronousComponent extends Component {
  Event sendreceive(Event request) {
      ...
  }
  ...
}
```

A few problems may occur here, for exemple the conflict between the regular reception of events of type *rtype* (if the component accepts such events) and the waiting activities. Different strategies can be proposed here :

1. resolving synchronous waits in priority and exclusively
2. sequentially executing the different possibilities
3. create a new activity every time a conflict is detected and treat the events concurrently

For the moment, we're thinking about making synchronous waiting loops prioritary and exclusive as the default conflit resolution rule by introducing specific synchronous event types[7] but other rules may be implemented.

3.3 Future delegation

Another basic model we should provide is the Future type of communication. Here, what we want to model is a pseudo-synchronous interaction. A request is sent as usual but instead of making the call blocking, a delegate of the reply is instantly replied so the next instructions in the code sequence may be executed without waiting for the actual reply. The delegate can then be tested to know if the reply is ready.

This model only involves a modification at the language level, the synchronous model of communication is totally reusable at the other levels.

We propose the following syntax to handle future communications :

```
delegate = future request
--next instructions
delegate.wait
reply = delegate.getreply
--handle the reply
```

Here, `--next instructions` are executed as soon as the request has been handled by the Output Queuing metaobject. Then we can decide to block the execution with a passive wait (`delegate.wait`) or we can simply check from time to time if the reply is ready (`delegate.isready?`).

4 Publish/Subscribe

The previous communication models were really simple and we can easily imagine to build them upon the canonical asynchronous model. The publish/subscribe model is more complex and so is generally provided as a basic mechanism. We propose here a way to compose multiple connections in order to implement a publish/subscribe model as a simple Comet Protocol. This will reclaim several modifications, from the linguistic level to the operational one.

[7] To avoid any interference, we can establish the following law : a synchronous event is the composition of two asynchronous events.

4.1 Linguistic issues

From the publisher point of view, we want to tell that we will publish some kind of event :

```
publish event
```

From the subscribers point of view, we simply want to describe the events that we're interested in :

```
subscribe descriptor callback
```

Where the description can be topic-based or content-based and the callback is simply a functionality to call when the matching events are received.

4.2 Publish/Subscribe protocols

At the protocol level, we can imagine at least two alternatives to implement a publish/subscribe protocol.

First, we can use a centralized scheme as depicted in Figure 3. This protocol introduces three kinds of components :

1. **Protocol Manager** : this specific component handles the whole publish/subscribe protocol.
2. **Publishers** : these are the components that generate the data. They first have to register themselves within the protocol manager, using a RegisterPublisherEvent. Then, publishers can emit events of type PublishEvent. Depending on the kind of publish/subscribe we want to handle, subtypes of PublishEvent must be provided. For example, a topic-based publishing will need the declaration of the topic concerned.
3. **Subscribers** : after receiving a published event, the protocol manager must update the concerned subscribers through an UpdateEvent. To know which subscriber might be interested by the update, it has to send a SubscribeEvent describing its needs.

Besides its simplicity, the main problem with this centralized protocol is that if the number of publishers and the number of subscribers increases, then the protocol manager becomes very sensitive regarding performance or security/safety. We can distribute the protocol manager using another protocol but regarding to performances, perhaps a better idea would be to distribute the manager "intelligence" within the several publishers involved. Each publisher will manage its own set of subscribers. The protocol manager will then only be used as a group leader. Of course, this solution is less scalable in term of connection number.

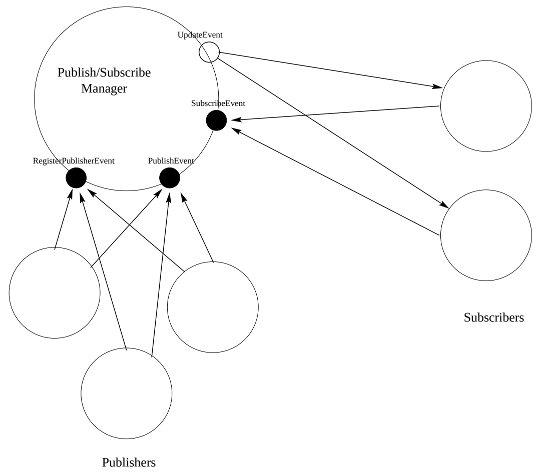


Figure 3: Centralized Publish/Subscribe Protocol

4.3 Adaptive version

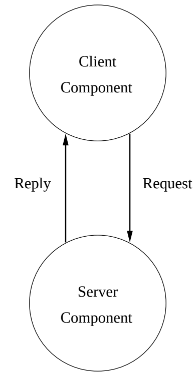
The centralized and distributed versions of our publish/subscribe protocol that we discussed above presuppose that the publishers and the subscribers already know how to handle the protocol events : PublishEvent, RegisterPublisherEvent, SubscribeEvent, UpdateEvent. This is a very strong assumption because we simply say here that the components are meant to communicate through publish/subscribe interactions. But the main idea underlying this paper is that the interaction styles are not exclusive, they can cooperate.

Let's take the example of a simple client/server application, already deployed and running. The client and the server participate in a simple client/server protocol that is built on the simple asynchronous model (cf. Figure 4(a)).

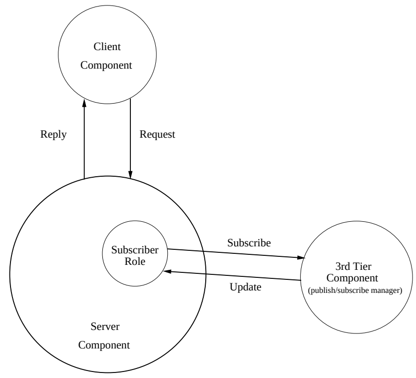
Now suppose that we want to make the server participating to a new interaction with a third tier (for example, a database system) that communicates through publish/subscribe events. The first possibility would be to stop the client/server, update the server code to make it able to participate to the publish/subscribe protocol and then deploy again the application and restart it. We propose in Comet, through the definition and dynamic assigment of roles, to modify the server dynamically so that it will be able to interact with the third tier without having to stop the application and deploy it again.

In fact, we want to make the server a subscriber for the protocol[8] . So the server has to know how to subscribe a particular kind of event (SubscribeEvent) and how to interpret the updates (UpdateEvent). This can be done by the definition of a Role that handles the SubscribeEvent and UpdateEvent types. Through reflection, we can then make this Role able to intercede with the component.

[8] We suppose that the third tier already knows how to handle this protocol as a manager/publisher.



(a) Client/Server Application



(b) Server subscription

Figure 4: Cooperation of interaction styles through protocol composition

From the system point of view (cf. Figure 4(b)), the server component will be considered as a server using simple asynchronous interactions for the client/server protocol and as a subscriber for the publish/subscribe protocol. That is, the two styles of interaction, both useful while designing the system, now cooperate at runtime, thanks to our generic approach.

Although the general principles introduced here will have to be put in practice and detailed more deeply, we already implemented other similar protocol compositions.

5 Conclusion

Our Comet architecture provides a minimal framework for designing and supporting a lot of different interaction styles. Moreover, the collaboration of different styles can be obtained by the composition of protocols and the dynamic adaptation of components. Other interaction styles are also considered in this study :

- token-based continuations : some actor languages [8] provide an interesting way to coordinate at the language level a group of entities called actors and similar to the Comet components. We are working on a Comet mapping of this interaction style.
- transaction systems : complex protocols like two-phase commit or nested transactions can be also thought as composition of simpler protocols.
- multi-agent systems interactions : higher-level interactions, involving intelligent agents and complex communication semantics can be, in our opinion, mapped on the generic model we propose.

The Comet/J framework which implements all the Comet concepts within the Java Environment will be used as the testbed platform for all our modeling experiences.

References

[1] A. Siegel B. Oki, M. Pfluegl and D. Skeen. The information bus - an architecture for extensible distributed systems. In *Fourteenth ACM Symposium on Operating System Principles*, December 1993.

[2] Akinori Yonezawa et al. *Object-Oriented Concurrent Programming*. MIT Press, 1987.

[3] Huw Evans and Peter Dickman. Zones, Contracts and Absorbing Changes: An Approach to Software Evolution. In *Proceedings of OOPSLA'99*. ACM Press, November 1999.

[4] Jeff McAffer. Metalevel Programming with Coda. In *Proceedings of ECOOP'95*, LNCS 952, pages 190–214. Springer Verlag, August 1995.

[5] Object Management Group. The Corba Architecture. `http://www.omg.org/corba`.

[6] Frédéric Peschanski. Comet : a Component-Based Reflective Architecture for Concurrent and Distributed Programming. In *First OOPSLA Workshop on Object-Oriented Reflection and Software Engineering*. Technical report, University of Milano-Bocca, Italy, November 1999.

[7] Daniel C. Sturman. *Modular Specification of Interaction Policies in Distributed Computing*. PhD thesis, University of Illinois at Urbana-Champaign, May 1996.

[8] Carlos Varela. Salsa : An Actor-Based Approach to World Wide Computing. to be published.