Separating Replication from Distributed Communication: Problems and Solutions *

Miguel Antunes[†]
Miguel.Antunes@inesc.pt

Hugo Miranda[‡]
hmiranda@di.fc.ul.pt

António Rito Silva[§]
Rito.Silva@inesc.pt

Luís Rodrigues[¶]
ler@di.fc.ul.pt

Jorge Martins[‖]
Jorge.B.Martins@inesc.pt

Abstract

Replication and distributed communication are usually tightly coupled. This code tangling forbids their independent reuse and adaptation. In this position paper the problems resulting from coupling replication with distributed communication are discussed. In addition, a solution based on separation of concerns is proposed. The abstractions for each concern are presented as well as their composition.

*This work was partially supported by Fundação para a Ciência e Tecnologia, Praxis/ C/ EEI/ 33127/ 1999 MOOSCo and Praxis/ C/ EEI/ 12202/ 1998 TOPCOM. Selected portions of this report were published in the Proceedings of the International Workshop on Distributed Dynamic Multiservice Architectures (DDMA), in conjunction with the 21st International Conference on Distributed Computing Systems (ICDCS-21) April 16-19, 2001. Phoenix, Arizona, USA.

[†]INESC/IST Technical University of Lisbon, Rua Alves Redol n°9, 1000-029 Lisboa, PORTUGAL

[‡]University of Lisbon, FCUL, LASIGE, Edifício C5, Campo Grande, Lisboa, PORTUGAL

[§]INESC/IST Technical University of Lisbon, Rua Alves Redol n°9, 1000-029 Lisboa, PORTUGAL

[¶]University of Lisbon, FCUL, LASIGE, Edifício C5, Campo Grande, Lisboa, PORTUGAL

[‖]INESC/IST Technical University of Lisbon, Rua Alves Redol n°9, 1000-029 Lisboa, PORTUGAL

1 Problem

Distributed multi-user interactive systems are an extremely relevant application area. Applications such distributed simulation, computer supported collaborative work (CSCW), multi-user games or dungeons (MUDs), and multi-user object-oriented environments (MOOs) are becoming increasingly pervasive. The MOOSCo project [10], Multi-user Object-Oriented environments with Separation of Concerns, addresses the difficulties in applying a component-based approach in a vertical and integrated manner, from analysis to implementation, to the design of this class of systems. In this project the experience of two research groups, a software engineering group and a distributed systems group, is being integrated. In particular, the composition of middleware abstractions and infrastructure communication protocols is being studied. This position paper discusses the problems associated with the composition of replication with distributed communication.

Replication and distributed communication are usually tightly coupled since replication only makes sense in the context of distributed applications. Almost accidentally, replication is mixed with distributed communication since the latter is usually used as the base of distributed applications construction. This situation results in the well-known code tangling problem [7].

Regardless of non orthogonality, it is necessary to find completely independent solutions for both concerns, such that it is possible to reuse, adapt and compose them independently. This separation of non-orthogonal concerns is a major open problem [1].

In synthesis, the following problems are addressed:

- What are the abstractions for replication and distributed communication that allows them to be independently specified from each other?
- How to deal with inter-concern dependencies such that their impact is controlled in order to keep independent the definition and evolution of each concern.

2 Solution

The proposed solution is to treat both concerns at the same level, instead of defining replication on top of distributed communication. The approach considers replication independently from distributed communication.

Abstractions for replication and distributed communication are defined in a way that each abstraction does not raise any assumption about the

other abstraction.

In order to deal with inter-concern dependencies a new abstraction is defined, a composition abstraction, that ensures the independence of each concern's specific parts. That way, the whole, the composition, is more than the sum of the parts, the composed concerns.

The independence of the concern abstractions allows them to be easily reused and adapted. Each concern has several variations that should be used depending on the application specific needs. For example, different distributed architectures can be used for distributed communication, e.g., client-server, peer-to-peer and hybrid. Both architectures have their advantages and disadvantages thus, designers should be able to choose which is more suitable for the target application. Regarding replication, different consistency criteria are needed for different objects depending of what is their replicated state, their update frequency or even their semantics. Forcing the same consistency criteria for all updates is too restrictive and may impose unnecessary overhead on the system.

So, the proposed approach requires that each concern abstraction possesses the expressive power required to support the different concerns variations. Moreover, in order to keep concerns independent, the composition abstraction should also support the variations that are part of the composition semantics. That way, non-orthogonal aspects of composition are confined to the composition abstraction and the impact of their evolution is controlled in the right place.

In the rest of this paper abstractions for distributed communication and replication are presented, Sections 3 and 4 respectively. In Section 5 it is also presented an abstraction for their composition. It is shown the expressive power of these abstractions, and how they support concern and concern composition variations. Conclusions and future work are presented in Section 6.

3 Distributed Communication

3.1 Variations

A solution for the distributed communication should consider the following variations:

- *Communication Protocol.* The solution may choose between different communication protocols. For instance, multicast or unicast. It may

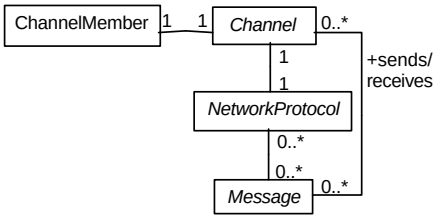

Figure 1: Distributed communication abstraction structure.

also choose between the different qualities of service. For instance, reliable and unreliable.

- *Distributed Architecture.* The solution may choose between different distributed architectures. For instance, client-server or peer-to-peer.

3.2 Structure

Figure 1 presents the structure proposed for the distributed communication concern. The abstraction has the following participants:

- `ChannelMember`. It represents a particular application that is associated with a channel. It is used to send and receive messages from the members that form a communication channel.
- `Channel`. A `Channel` instance represents a channel endpoint over which messages can be sent to all (or a subset) of the channel members and over which messages sent to channel members can be received. It is associated with a communication protocol, `NetworkProtocol`, that is used for message delivery with a particular quality of service.
- `NetworkProtocol`. It is responsible to send messages from one channel member to the remaining channel members through the network, according to a particular quality of service.
- `Message`. It represents the information exchanged between channel members through `Channel` instances.

3.3 Expressiveness

It is possible to specialize the proposed abstraction to support the different variations.

To support different communication protocols and qualities of service `NetworkProtocol` is specialized. There are two main specializations, `Unicast-Protocol` and `MulticastProtocol`, that represent, respectively, point-to-point and multicast communication protocols. These classes can be further specialized to provide different qualities of services. For instance, `ReliableUnicast` and `ReliableMulticast`.

To support different distributed architectures, the class `Channel` is specialized. `MulticastChannel` represents communication channels for multicast architectures. Note that this architecture requires a particular communication protocol, multicast protocol, supported by a specialization of `MulticastProtocol`. However, there is no commitment for a particular quality of service.

`ClientChannel` and `ServerChannel` support client-server architectures. In this distributed architecture a communication channel is represented by an instance of `ServerChannel` and multiple instances of `ClientChannel`. All messages are sent by a `ClientChannel` instance to the `ServerChannel` instance which is responsible to deliver them to the other `ClientChannel` instances. Note that this distributed architecture requires a particular communication protocol, unicast protocol, supported by a specialization of `Unicast-Protocol`.

3.4 Implementation with APPIA

Appia [1] is a communication architecture that allows different communication channels, each with its own QoS, to be integrated in a coherent multi-channel protocol stack [9]. *Appia* recognizes the need to integrate channels, allowing properties to be shared across several channels. Most of the previous distributed communication models such as the x-Kernel [6] and Ensemble [5] offer limited support for expression of *inter-channel constraints*. The work with Maestro [2] illustrates the difficulties of maintaining consistent failure detection when channels with diverse characteristics are used concurrently. To satisfy *inter-channel requirements*, *Appia* extends the abstraction provided by previous works.

The architecture of *Appia* allows the application designer to specify the protocol stack that meets her/his QoS requirements through the composition of micro-protocols. *Appia* addresses these problems by providing a stack composition model that allows to express *inter-QoS* requirements.

Figure 2 shows how *Appia* can coordinate two different objects A and B

[1] *Appia* was started in the context of the previous project, TOPCOM.

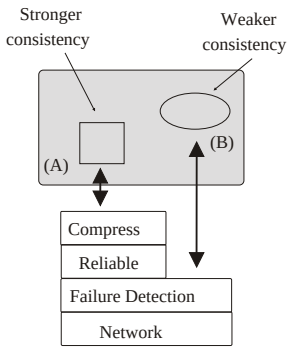

Figure 2: Two objects with different distributed consistency requirements in *Appia*

with independent consistency requirements. Both channels share a common failure detector module; this way, inconsistencies motivated by the unreliability of failure detection are avoided.

Appia handles inter-QoS requirements in a clean way: property sharing is achieved by allowing the same protocol instances to be present in the required channels. The figure presents two distinct *Appia* channels, one for object (A) and another for object (B), and behaving as such for the application programmer. Despite the flexibility of the model, developing protocols for *Appia* is not harder than for previous protocol frameworks. Depending on protocol behaviour, participation of an instance in several channels can be transparent to the implementation.

4 Replication

4.1 Variations

When developing application-specific solutions for replication the following variations should be considered:

- *Shared State*. It should be possible to choose the shared state on a per object basis. The shared state is defined identifying the object's attributes and actions that should be shared.

- *Consistency Protocols*. Different consistency should be allowed. Consistency criteria should be defined on a per object basis. Moreover, it

should be possible to define consistency criteria that apply to several objects.

- *Replication policy.* The replication policy to handle newcomers may change. In some situations a new shared space member receives all the shared objects from the same member. In other situations several members contribute for providing the new member with the shared space's objects.

4.2 Structure

Figure 3 presents the structure of the proposed solution for the replication concern. The abstraction has the following participants:

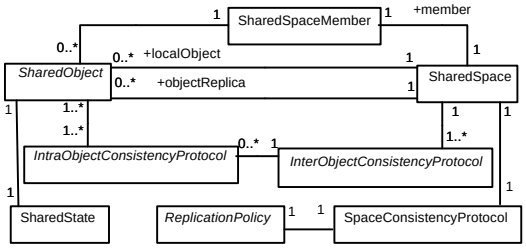

Figure 3: Replication abstraction structure.

- `SharedSpaceMember`. An application that is associated with a `SharedSpace` instance.
- `SharedSpace`. Represents a members's view for a particular shared space. It is through `SharedSpace` instances that members may access shared objects that exist in a shared space.
- `SharedObject`. An object shared in the context of a shared space.
- `SharedState`. Represents the part of a `SharedObject` instance that is actually shared.
- `IntraObjectConsistencyProtocol`. Represents a consistency protocol that enforces some consistency criteria on a `SharedObject` instance. It may be applied to one or more shared attributes and actions of the shared object.

- `InterObjectConsistencyProtocol`. Represents a consistency protocol that enforces some consistency criteria between state updates of different objects. Each aggregates several intra-object consistency protocols.

- `SpaceConsistencyProtocol`. Represents a consistency protocol that is used for maintaining consistency between shared space members regarding the number of existing shared objects. Moreover, `SharedSpace` instances use their `SpaceConsistencyProtocol` to propagate shared objects creation and destruction to other shared space members using a particular consistency criteria.

- `ReplicationPolicy`. Represents a policy to manage the creation and destruction of shared objects, whenever shared space members join and leave a shared space. Each shared space consistency protocol delegates in a `ReplicationPolicy` instance the handling of: the activation and deactivation of shared objects; space membership changes; and determining which member or members have the responsibility of sending to newcomers information about the existing shared objects.

4.3 Expressiveness

It is possible to specialize the proposed abstraction to support the different variations.

The abstraction for the replication concern allows for the application shared state to be defined on a per object basis. This allows a better control of what state changes must be propagated to other space members.

Different consistency protocols can be obtained by specializing classes `IntraObjectConsistencyProtocol` and `InterObjectConsistencyProtocol`. For instance, class `DeadReckonProtocol`, is a specialization of `IntraObjectConsistencyProtocol` that can be used to reduce the number of state updates that are propagated between space members, using a state prediction algorithm[2]. Class `CausalSpaceProtocol` defines a inter-object consistency protocol that forces causal ordering for state updates. It can be used to force the state updates of a particular set of objects to be causally ordered.

Different specializations of `ReplicationPolicy` can be defined to support different policies for handling newcomers and shared object creation

[2]DeadReckon algorithms are used in Multi-User Virtual Environments to reduce state updates of objects positions that can be predicted using the previous position and velocity of the object. Variations have been defined that apply this algorithms to any kind of data .

and destruction. For instance, class `DistributedPolicy` represents a policy where each member is responsible to send the replicas of the objects they have created to the newcomers. The classes `MasterPolicy` and `SlavePolicy` represents a policy where the responsibility of handling newcomers belongs to a single member, the master. The remaining members, the slaves, do not handle space membership changes and when they join a space, they send replicas of their objects to the master, and obtain the other member's replicas from the master.

4.4 Implementation

The implementation of consistency protocols was based on the concept of protocols layers and protocol stacks that can be found in several distributed communication platforms. Each layer supports a particular consistency protocol and protocols are composed by layering them in protocol stacks. In some aspects the structure of consistency protocols is very similar to the communication protocols supported by *Appia*. There are however some differences that try to take into account consistency protocols semantics. For instance, when initializing a consistency protocol stack, the layers determine if the state updates produced locally must be propagated through the protocol stack before being applied locally or if they can be immediately applied. For instance, if there is a layer that must force some ordering or synchronization criteria to state updates then they must be first propagated trough the stack; otherwise they can be immediately applied avoiding unnecessary delays. Also, the messages exchanged between protocol layers are state updates that can be inspected by protocols, and not simple opaque byte sequences like in communication platforms. Object protocol layers have access to the objects they are managing so that they take into account specific object semantics when forcing some consistency criteria.

Figure 4 shows a possible run-time structure of a `SharedSpace` instances and its consistency protocols. Note that the `ReplicationPolicy` class is in fact implemented as a special space protocol layer that always exist in a space control protocol.

5 Composition

5.1 Structure

The composition abstraction has the following participants:

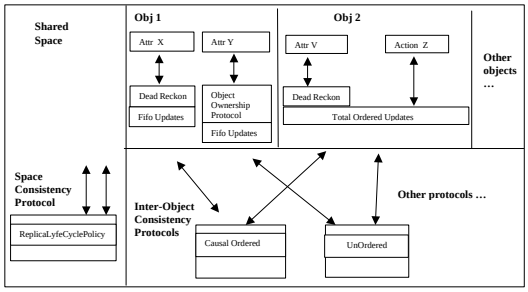

Figure 4: Run time structure of consistency protocols.

- `DistInterObjectConsistencyProtocol`. Represents a specialization of `InterObjectConsistencyProtocol` that uses a `Channel` instance to receive and propagate state updates to remote space members. It represents the composition between inter-object consistency protocol and distributed communication. Note that this composition must also be enforced for any of `InterObjectConsistencyProtocol` specializations described in section 4.3. The way the composition is supported is implementation dependent. At this level it is only important to identify what are the concern's elements that must be composed in order to obtain the intended functionality, i.e., distributed replicated objects.

- `DistSpaceConsistencyProtocol`. A specialization of `SpaceConsistency-Protocol` that implements a distributed membership protocol for managing space membership changes. Creation and destruction of replicas are propagated to remote members trough the `Channel` instance(s) associated with a `DistSpaceConsistencyProtocol`.

5.2 Expressiveness

Support for different consistency criteria is obtained by specializations of both `IntraObjectConsistencyProtocol` and `DistInterObjectConsistency-Protocol` classes, that enforce different consistency criteria to object's state updates. Also the `Channel` instances used by `DistInterObjectConsistency-Protocol` specializations must use `NetworkProtocol` instances that allow those consistency criteria to be supported over distributed communication. For instance, if a certain object requires some consistency criteria that forces ordered state updates, then it is necessary that the network protocol used

by the DistInterObjectConsistencyProtocol's channel supports at least reliable message delivery.

Different distributed architectures are obtained using different combinations of Channel and ReplicationPolicy specializations. For instance, a pure client-server architecture can be obtained by using ClientChannel and ServerChannel instances combined with SlavePolicy and MasterPolicy instances, respectively. The clients use ClientChannel instances associated with DistInterObjectConsistencyProtocol and DistSpaceConsistency-Protocol instances; and SlavePolicy associated with their DistSpace-ConsistencyProtocol instance. The server uses a ServerChannel instance to route network messages , i.e. state updates, between clients and uses MasterPolicy for managing newcomers.

5.3 Implementation

The implementation of DistInterObjectConsistencyProtocol and DistSpace-ConsistencyProtocol is supported by specializations of consistency protocol layers (see section 4.4) that have an associated Channel instance. It also uses the *Appia* implementation of Channel to perform distributed communication over different network protocols and qualities of service. The resulting composition supports distributed replicated objects, and it also maintains the support for the both concerns variations. Since the composition between distributed communication and replication is well identified and isolated, changing or extending that composition is made easier and only affects the composition code and not the correspondent concerns. For instance, decisions about associating different channels to different inter-object consistency protocols, or sharing the same channel between different consistency protocols, or even using more than one channel for the same consistency protocol can be taken at the composition level, by specializing DistInterObjectConsistencyProtocol and DistSpaceConsistency-Protocol appropriately.

There are, however, some dependencies that arise from the composition of distributed communication with replication, namely from composing consistency protocols with distributed communication. For instance, consistency protocols that enforce some ordering criteria needs that the underlying distributed communication is at least reliable. In other situations, using reliable communication may introduce unnecessary delays since the consistency protocols can deal with state updates losses, as it is the case of DeadReckon protocols. It is necessary to guarantee that the distributed communication protocols provide the necessary quality of service so that consistency

protocols can be enforced. At present, although *Appia* can manage protocol dependencies within its protocol stacks, there is no way of enforcing the dependencies between consistency protocols and *Appia*'s communication protocols.

6 Conclusions

The work presented in this paper describes a solution for supporting distributed replication that considers replication and distributed communication as two separated concerns. For each concern, a solution is proposed that supports several variations that are relevant for the domain of multi-user virtual environments. Distributed replication is obtained by composing solutions described for the replication and distributed communication concerns. This approach allowed reasoning about communication and replication at different levels of abstractions, and at the same time allowed for the composition of those concerns. Furthermore, identifying the composition points allowed for optimizations to be performed at the composition level [3, 8, 4].

Despite having defined independent abstractions for replication and distributed communication, the implementations of those abstractions were very similar, namely both the implementations of consistency protocols and communication protocols use the concepts of protocol layer and protocol stack. The main reason to use such concepts was to easily support protocol definition and composition.

One of the goals of this work is try to understand if it is possible for a platform like *Appia* to support both consistency and distributed protocols without violating the separation of concerns. Although *Appia* is focused on supporting communication protocols, it is sufficient generic to allow specializations for other domains like consistency maintenance.

The main advantage of using the same mechanisms for supporting both consistency and communication protocols, is that it may simplify the management of existing dependencies between particular consistency protocols and quality of service of the underlying communication protocols. As stated before, *Appia* already gives support for resolving dependencies between communication protocol layers. However, before using *Appia* as a general mechanism to support the definition of consistency and communication protocols, it is necessary to understand if *Appia* offers the necessary semantics to support consistency protocols as it supports communication protocols. Moreover, it is necessary to guarantee that supporting both types of protocols

will not violate the separation of concerns given by the defined abstractions.

In the future the issue of using the same mechanism to support both communication and consistency protocols will be further investigated. If necessary, changes will be made to *Appia* in order to successfully support consistency protocols definition. However, this must be done, carefully, without affecting the current ability to efficiently support communication protocols.

References

[1] Lowdewijk Bergmans and Mehmet Aksit. Composing software from multiple concerns: A model and composition anomalies, June 2000. ICSE'2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering.

[2] K. Birman, R. Friedman, and M. Hayden. The maestro group manager: A structuring tool for applications with multiple quality of service requirements. Technical report, Cornell University, Ithaca, USA, February 1997.

[3] Thomas Funkhouser. Network Topologies for Scalable Multi-User Virtual Environments. In *Proceedings of the 1996 IEEE Virtual Reality Annual International Symposium (VRAIS)*, pages 222–228, San Jose, CA, April 1996. IEEE Neural Networks Council.

[4] Chris Greenhalgh. Spatial Scope and Multicast in Large Virtual Environments. Technical Report NOTTCS-TR-96-7, Department of Computer Science, The University of Nottingham, UK., 1996.

[5] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, Computer Science Department, 1998.

[6] N. Hutchinson and L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Trans. on Software Engineering*, 17(1):64–76, January 1991.

[7] Gregor Kicsales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. Technical Report SPL97-008 P9710042, XEROX PARC, February 1997.

[8] Michael Macedonia, Michael Zyda, David Pratt, Donald Brutzman, and Paul Barham. Exploiting Reality with Multicast Groups. In *IEEE Computer Graphics and Applications*, pages 15(5):38–45, September 1995.

[9] H. Miranda and L. Rodrigues. Flexible communication support for CSCW applications. In *5th Internation Workshop on Groupware - CRIWG'99*, pages 338–342, Cancún, México, September 1999. IEEE.

[10] MOOSCo. Multi-user Object-Oriented environments with Separation of Concerns Project. MOOSCo Home Page URL: http://www.esw.inesc.pt/moosco.