Separating Replication from Distributed Communication: Problems and Solutions

Miguel Antunes[1], Hugo Miranda[2], António Rito Silva[1], Luís Rodrigues[2] and Jorge Martins[1]
[1] INESC/IST Technical University of Lisbon, Rua Alves Redol n°9, 1000-029 Lisboa, PORTUGAL
Miguel.Antunes,Rito.Silva,Jorge.B.Martins@inesc.pt
[2] LASIGE/FC University of Lisbon, Edifício C5, Campo Grande, Lisboa, PORTUGAL
hmiranda,ler@di.fc.ul.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.

1 Problem

Distributed multi-user interactive systems are an extremely relevant application area. Applications such as virtual environments, 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].

A possible solution for the code tangling problem is based on the definition of distributed communication abstractions that hide communication details and replication is built on top of distributed communication. Defining good abstractions for distributed communication may help reducing the dependency between the replication code and distributed communication code. However, even when using good abstractions, there are decisions concerning communication that are scattered through out the replication code, the replication code is biased by the distributed communication abstractions. Those decisions are about the creation and destruction of communication channels, and when to use those channels to propagate state updates to other replicas. In other words the replication is tightly coupled with distributed communication in the context of a particular composition. So, if changes are needed to how replication and distributed communication are composed then the replication code must also be changed.

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 dis-

tributed communication. The approach considers replication as completely independent 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 the inter-concern dependencies a new abstraction is defined, a composition abstraction. This composition abstraction deals with inter-concern dependencies, keeping independent the concern's specific parts. That way, the whole, the composition, is more than the sum of the parts, the composed concerns.

Concern abstraction independence 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: client-server, are simpler to manage but may compromise system scalability, peer-to-peer using multicast protocols are more scalable but are more complex to manage. Both architectures have their advantages and disadvantages. Applications should be able to choose which is more suitable. For instance, there are situations where despite potential scalability problems, client-server architectures must be chosen due to the lack of multicast support by the underlying network like the internet. 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. Achieved results are discussed in Section 6.

3 Distributed Communication

3.1 Variations

There is no single and optimal solution for the distributed communication concern. Depending on the context, on the application domain and on its requirements, different solutions are possible. The chosen solution is the result of a trade-off between the different possibilities. When treating distributed communication there are two aspects that must be addressed: communication protocol, and distributed architecture. The solution for each one of these aspects should consider different possibilities.

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

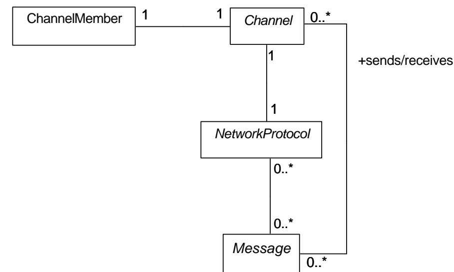

Figure 1. Distributed communication abstraction structure.

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 to send messages to 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 Collaborations

An application becomes a Channel Member by associating itself with a Channel instance.

A Channel instance delivers messages to all the members, all the members except the sender, or to a specific member. It uses its network protocol to effectively send the message.

An application leaves a channel by disassociating itself from its Channel instance.

3.4 Expressiveness

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

3.4.1 Specialization for Communication Protocols .

To support different communication protocols and qualities of service NetworkProtocol is specialized. There are two main specializations, UnicastProtocol 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.

3.4.2 Specialization for Distributed Architectures .

To support different distributed architectures Channel is specialized.

MulticastChannel represents communication channels for multicast architectures. Note that this architecture requires a particular communication protocol, multicast protocol. However, there is no commitment for a particular quality of service.

ClientChannel and ServerChannel support client-server architecures. In this distributed architecture a communication channel is represented by an instance of ServerChannel and several instances of ClientChannel. All messages are send by a ClientChannel instance to the ServerChannel instance which is responsible to deliver them. Note that this distributed architecture requires a particular communication protocol, unicast protocol.

3.5 Implementation with APPIA

3.5.1 Network Protocol refinement Modern distribution support frameworks refine the *NetworkProtocol* abstraction to encompass the composition of several micro-protocols in a single channel. Each micro-protocol provides one property to messages that are sent using it. Most of the micro-protocols rely on the properties provided by others to satisfy their own property, defining relations of dependence between them. The constraints imposed this way are named *intra-channel constraints*.

Applications may use simultaneously several channels, each supported by a possible independent network protocol. However, for some particular requirements it is possible to find dependencies between the properties expressed in different channels. For example, consider an application having concurrent messages flowing by several channels. To avoid inconsistencies derived from the unreliability of failure detectores, one might want to share the same failure detector between them. These constraints are named *inter-channel constraints*.

Each valid combination of properties provide a different Quality of Service (QoS). Channels are the instantiation of one particular QoS.

3.5.2 *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 of CCTL [12] also uses different communications channels which are managed by a single control channel. 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 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 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

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

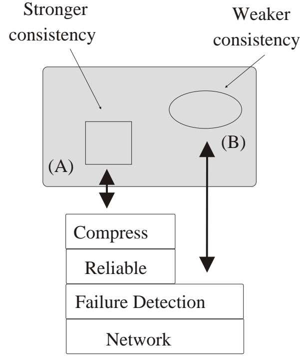

Figure 2. Two objects with different distributed consistency requirements in *Appia*

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 possibilities 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.
- *Replica Life Cycle*. Shared space life cycle may change. In some situations a new shared space receives all the shared objects from the same shared space member while, 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:

- `SharedSpaceMember`. An application that is associated with a `SharedSpace` instance.

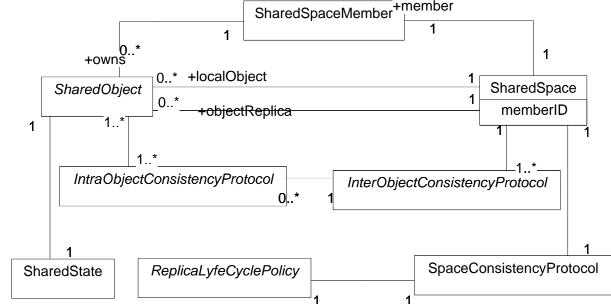

Figure 3. Replication abstraction structure.

- `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.
- `InterObjectConsistecyProtocol`. Represents a consistency protocol that affects more than one object. It is used to force some consistency criteria between state updates of different objects. Each it 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.
- `ReplicaLyfeCyclePolicy`. 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 `ReplicaLyfeCyclePolicy` policy the handling of: the activation and deactivation of shared objects; space membership changes; and determining which member or members have the responsibility of sending newcomers the information about the existing shared objects.

4.3 Collaborations

An application joins a shared space through a `SharedSpace` instance. Other shared space members will be informed that a new member has joined the space through their `SpaceConsistencyProtocol` instance. Depending on the replica life cycle policy being used, the new member will obtain object replicas from one or all of the remaining members. Leaving a shared space will cause all shared objects's local replicas to be destroyed in the member's `SharedSpace` instance.

A member may add or remove objects from a shared space. Adding and removing object from a shared space causes replicas to be created or destroyed in the other space members, respectively.

Every time a space member causes some state changes on a particular object, the state update will be propagated to other members using some consistency criteria. First the state update will be handled by a particular `IntraObjectConsistencyProtocol`. Next, the state update is handled by the associated `InterObjectConsistencyProtocol`, so that some consistency criteria between several objects can be enforced and also propagated to other shared space members.[2]. At the other members, the state update will first be handled by the inter-object consistency protocol, then by the intra-object consistency protocol and finally applied to the shared object replica.

4.4 Expressiveness

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

4.4.1 Shared State Definition .

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.

4.4.2 Specialization for Consistency Protocols .

Different consistency protocols can be obtained by specializing classes `IntraObjectConsistencyProtocol` and `InterObjectConsistencyProtocol`. For instance, class `DeadReckonProtocol`, is a specialization of `IntraObjectConsitencyProtocol` that can be used to reduce the number of state updates that are propagated between space members. By using a state prediction algorithm[3] 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.

4.4.3 Specialization for Replica Life Cycle Policy .

Different specializations of `ReplicaLyfeCyclePolicy` can be defined to support different policies for handling newcomers and shared object creation and destruction. For instance, class `DistributedPolicy` represents a policy where each member is responsible to send newcomers the replicas of the objects they have created. The classes `MasterPolicy` and `SlavePolicy` represent 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 they objects to the master, and obtain the other member's replicas from the master.

4.5 Implementation

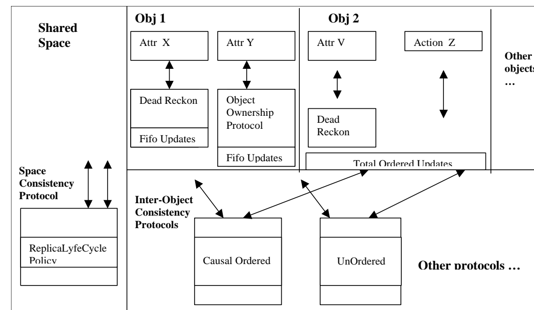

Figure 4. Run time structure of consistency protocols.

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.

[2]The propagation of state updates to other space members depends of the protocol implementation. In the presence of distribution the propagation to remote members will be supported trough the composition with distribution communication concern.

[3]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 [11].

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 `ReplicaLyfeCyclePolicy` is in fact implemented as a special space protocol layer that always exist in a space control protocol,

5 Composition

5.1 Structure

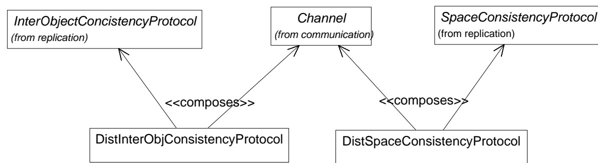

Figure 5. Structure of the replication and distributed communication composition.

- `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 4.4.2. 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 `SpaceConsistencyProtocol` 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 Collaborations

Space membership changes are received by each space member, i.e, `SharedSpace` instance, through their specialization of space consistency protocol, class `DistSpaceConsistencyProtocol`. The space membership changes are detected by listening to channel membership changes on the associated `Channel` instance.

Like in section 4.3 `DistInterObjectConsistencyProtocol` instances are used so that inter-object consistency criteria for state updates may be enforced. After processing the update each distributed inter-object protocol propagates the state updates to other members using the associated `Channel` instance to send the update information to other space members. Remote members receive state updates trough the `Channel` instances associated with their `DistInterObjectConsistencyProtocol` instances. The handling of state updates is then processed as described in section 4.3.

5.3 Expressiveness

5.3.1 Support for Different Consistency Criteria .

Support for different consistency criteria is obtained by specializations of both `IntraObjectConsistencyProtocol` and `DistInterObjectConsistencyProtocol` classes, that enforce different consistency criteria to object's state updates. Also the `Channel` instances used by `DistInterObjectConsistencyProtocol` 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.

5.3.2 Different Distributed Architectures .

Different distributed architectures are obtained using different combinations of `Channel` and `ReplicaLyfeCyclePolicy` 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 `DistSpaceConsistencyProtocol` instances; and `SlavePolicy` associated with their `DistSpaceConsistencyProtocol` instance. The server uses a `ServerChannel` instance to route network messages , i.e. state updates, between clients and uses `MasterPolicy` for managing newcomers.

5.4 Implementation

The implementation of `DistInterObjectConsistencyProtocol` and `DistSpaceConsistencyProtocol` is supported by specializations of consistency protocol layers, see section 4.5, that have an associated `Channel` instance. Figure 6 shows a shared space instance runtime structure with its consistency protocol stacks. 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 `DistSpaceConsistencyProtocol` appropriately.

Appia already gives support for different variations on the composition. *Appia*'s ability to support inter-channel constrains by allowing protocol instances to be shared between channel's allows, for instance, to have one channel per consistency protocol with different quality of service, but also different channels that share the same network layer and thus use the same communication address for message transmission. Another example is the ability to, although having a channel per consistency protocol, still support failure detection in a uniform way preventing in this way the failure of a single channel to compromise replica consistency maintenance. The failure detection would be supported by having a common protocol layer to all channels that would perform failure detection globally to all channels.

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, any consistency protocols that enforces 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.

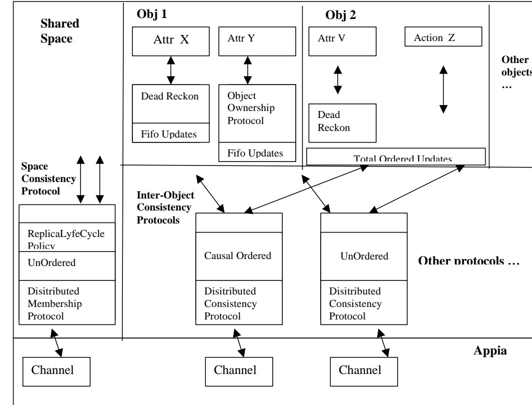

Figure 6. Run time structure of composition between replication and distributed communication concerns.

6 Discussion

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.

Despite being developed by two independent teams, the implementations of the solutions for replication and distributed communication are very similar. Although the concepts of protocol layer and protocol stack have been highly used in distributed communication platforms, they have not being used for supporting consistency protocols in the context of replication, at least in a distributed communication independent way.

Despite the similarities, there are however some differences due to semantics differences of consistency and communication protocols. Typically, consistency protocolos are at a higher level of abstraction than communication protocols. However, the mechanisms for supporting protocol composition and protocol extensibility can be the same.

One of the goals of this paper is to discuss and to try to understand if is possible for a platform like *Appia* to support both consistency and distributed protocols. Although *Appia* is focused on supporting communication protocols, it is sufficient generic to allow specializations for other do-

mains like consistency maintenance. One of the advantages 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 supports dependencies between communication protocol layers to be resolved.

A problem of using similar structures for defining consistency and communication protocols is that it becomes difficult to draw the line between what are consistency protocols and communication protocols. More precisely, can inter-object consistency protocols be represented entirely at communication level, or there are semantic differences that force them to be treated differently?

It is well known that different ordering protocols can be easily defined using platforms like *Appia* and *Appia* has also the advantage that supports *inter-channel constraints*, i.e., the ability for the same protocol layer to be shared by different channels, which can be used to support inter-object consistency constrains. But using the same protocol stack for composing consistency and communication protocols introduces other kinds of dependencies. Consistency protocols deal with state updates and may take advantages of the semantics of those state updates to force some consistency criteria. Communication protocols deal with byte sequence messages that will be sent over the network. At some point in the protocol stack it will be necessary to serialize state updates to byte sequences so that they can be handled by communication protocols. Instead of assembling consistency and communication protocols on the same stack, one could use a platform like *Appia* to support the different protocol stacks showed in figure 6: intra-object consistency protocol; inter-object consistency protocol; and communication protocol. The open question is whether it is possible to use the same structure to support semantically different protocols or customized solutions must be used for each kind of protocol due to the existent semantic differences. For instance, although ordering criteria can be supported both at the consistency and communication level, there are some differences depending if order of network message delivering or state updates is being considered. The former must be ensured only until the message is delivered to the application, the latter must be ensured until the update is applied to the application shared state, even in the presence of a multi-threaded execution model.

The problem of using the same platform to support both, consistency and communication protocols, consists in underestimating if despite their semantic differences is still possible to treat them identically. And more important, if treating them identically does not violates the separation of distributed communication and replication concerns, the original goal of this work. Basically, it is necessary to understand if the abstractions of protocol layers and protocol stacks can be used as a generic composition mechanism for composing replication and distributed communication, at least for the multiuser virtual environments domain where the ability for supporting different consistency criteria and communication quality of service is of great importance.

At the present moment the consistency and communication protocols are being considered separately and thus implemented by different although similar mechanisms. 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. Also, using different mechanisms allows for replication and communication optimizations to be treated separately. Furthermore, identifying the composition points allows for optimizations to be performed at the composition level [3, 8, 4]. The disadvantages are the management of the dependencies between consistency and communication protocols. This dependencies arise from non-orthogonality of both types of protocols.

7 Acknowledgments

This work was partially funded by Fundação para a Ciência e Tecnologia, Praxis/ C/ EEI/ 33127/ 1999 MOOSCo.

References

[1] L. Bergmans and M. 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, Feb. 1997.

[3] T. 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] C. 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, Jan. 1991.

[7] G. Kicsales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. Technical Report SPL97-008 P9710042, XEROX PARC, February 1997.

[8] M. Macedonia, M. Zyda, D. Pratt, D. Brutzman, and P. 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, Sept. 1999. IEEE.

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

[11] NetZ. Netz : Multi-player architecture for online games. NetZ Home Page URL:http://www.proksim.com/games/netz.htm.

[12] I. Rhee, S. Cheung, P. Hutto, and V. Sunderam. Group communication support for distributed collaboration systems. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 43–50, Balitmore, Maryland, USA, May 1997. IEEE.