Configuring the Communication Middleware to Support Multi-user Object-Oriented Environments[*†]

Sandra Teixeira
U. Lisboa
steixeira@di.fc.ul.pt

Pedro Vicente
U. Lisboa
pedrofrv@di.fc.ul.pt

Alexandre Pinto
U. Lisboa
apinto@di.fc.ul.pt

Hugo Miranda
U. Lisboa
hmiranda@di.fc.ul.pt

Luis Rodrigues
U. Lisboa
ler@di.fc.ul.pt

Jorge Martins
INESC-ID
Jorge.B.Martins@inesc-id.pt

António Rito-Silva
INESC-ID
Rito.Silva@inesc-id.pt

1st September 2002

Abstract

Distributed multi-user interactive systems have a rich and complex set of requirements. A promising approach to tackle the complexity of these systems is to rely on configurable architectures that are able to support component re-utilization and composition.

The MOOSCo project, 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. To support communication among distributed components, the project uses a configurable group communication system called *Appia*. The paper discusses the role of *Appia* in the MOOSCo architecture and shows how it makes possible to derive, in a simple and elegant way, the most appropriate protocol composition depending on the objects shared by the multi-user object-oriented environment.

[*]This work was partially supported by Fundação para a Ciência e Tecnologia, POCTI/ C/ EEI/1 33127/ 1999 MOOSCo.

[†]Sections of this report have been published in the In Proceedings of the International Symposium on Distributed Objects and Applications (DOA), October 2002, Irvine (CA), USA.

1 Introduction

Distributed multi-user interactive systems represent an extremely relevant research 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) [5] are becoming increasingly pervasive. From the analysis, software engineering and system support point-of-view, these applications are extremely challenging due to its unique requirements for dependability, scalability, adaptability, usability, non-functional domains to be considered, and efficiency.

The MOOSCo project [1] [12], 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. The MOOSCo project addresses the several concerns involved in the development of MOOs, such as object interaction, awareness management, distributed communication, information sharing and so forth. A promising approach to tackle the complexity of these systems is to rely on configurable architectures that are able to support component re-utilization and composition. The MOOSCo architecture is based on component composition and addresses three abstraction layers: user models, middleware abstractions, and communication protocols. Due to the compositional characteristics of the architecture, it is possible to use middleware abstractions and communication protocols tailored to the specific user models needed in each case. To support this fine-grain level of composition, including at the level of the communication protocols, the project is relying on a framework for protocol composition and execution called *Appia* [11].

In this paper we show the advantages of using *Appia* as a composition framework for MOOs with particular emphasis on distributed communication and information sharing concerns. Using a concrete example of a virtual space with different shared objects, we show how the application designer may select a different composition of communication protocols for each attribute of a shared object and how it can enforce inter-channel constraints. Moreover, this configuration can be performed using a configuration file. This allows the application to be configured during its deployment, not only as a function of the objects being shared but also of the properties of the network infrastructure.

The paper is structured as follows. The different configuration requirements of the MOOSCo system are introduced in Section 2. Section 3 makes a brief introduction to the *Appia* system and shows how it is used in MOOSCo. The comparative performance of the resulting system is presented in 4. Section 5 presents related work. The advantages and difficulties of the approach are discussed in Section 6. Section 7 concludes the paper.

2 Configurable multi-user environments

2.1 Multi-user environments

Multi-user virtual environments, such as MOOs, support real-time interaction between several geographically distributed users. To achieve this, MOOs offer linked virtual shared spaces, usually following a *virtual room* metaphor. It is within these virtual rooms that users may share data and multimedia information (such as graphics, images and sounds).

MOOs offer the mechanisms for a user to enter or leave a virtual room, watch other users activities while they happen and interact with other users in the environment. These systems allow the creation and modification of the virtual environment by change, addition and removal of objects.

Naturally, information sharing is a central aspect in the environment implementation. There are several types of information shared by the MOO users. First of all, the users have to be aware of other users in the same virtual room and of the operations performed by these. This implies that whenever a user enters or leaves the room, all other users are informed. Secondly, users must perceive and be able to interact with the objects in the room. This implies that the adding or removing of any object must be indicated to all users in the room. Furthermore the users must be aware of objects' changes: Each object is characterized by one or more attributes (for example, position, speed, sound, etc) that can be changed independently and one or more actions (for example rotate, play, etc) that can be performed at will. Finally, the communication requirements to propagate the changes of these attributes or actions may be completely different (for example, the data propagation protocols are different from the audio propagation protocols).

Although each object attribute places different demands on the communication protocols, the dissemination of attribute information must respect a global coherence so that every user has the same perception of the environment. This means that the MOOs communication support must allow the configuration of the quality of service required for a specific attribute, but also allow the configuration of the coherence relations between different attributes.

2.2 Monolithic solutions *versus* configurable solutions

As just described, there is not a unique and best solution in the context of MOOs. Solutions should be contextual. On the other hand, the overall satisfaction of MOO requirements for consistency, adaptability, scalability, and efficiency, is not easy and may result in conflicting and inconsistent solutions. For instance, due to latency, messages might arrive in different orders at different machines. This results in a consistency problem: different users get different views of the environment.

To solve this sort of problems most MOOs resort to a given, pre-defined, non-configurable strategy. For instance, some approaches rely on a centralized server to serialize messages. Such solution does not scale to a large number of users. Other approaches use a decentralized approach, that rely on communication protocols that provide total ordering and causal ordering for messages in the system. Such approaches may perform well in local-area networks but also exhibit scalability limitations in large-scale networks due to the number of messages that may need to be exchanged. In addition, domain-specific requirements may consider different levels of consistency and even their change at runtime.

Instead of relying on fixed strategies, the MOOs design and development will profit from an approach that allows the customization of contextual solutions by the tuning and composition of predefined reusable components. Even if there are several systems providing solutions for the information sharing support, few of these systems have the required flexibility to adapt to the applications' particular requirements. This paper intends to offer a solution to a better protocol composition for the application requirements.

2.3 Configuration requirements

Each virtual environment with its particular objects places different requirements on the information sharing support system. In this section we will introduce a very simple example that shows the complexity of the configuration requirements usually found in MOO systems.

Lets consider a simple game, a MUD (Multi-User Dungeon). In this type of game, each player personifies an animal that moves in a virtual universe, searching for food. Each user is represented by an *avatar*, implemented as a shared object with three attributes: its appearance, its geographical position and the direction it is facing; and a single action: eat. The appearance is influenced by the amount of food consumed. In the following example we will consider the scenario in which there are two users that interact in a virtual shared space composed of a single virtual room with a cooling fan and a food container. The cooling fan position is fixed and pre-defined so its only relevant attribute, in terms of changing information sharing, is its rotation speed. The food container has two relevant attributes: position and number of items within. The users must approach the food container and eat one or more items. This must reflect in their avatars' appearance.

It should be pointed out that if two users try to take the last item, only one of them should succeed. This requires that some order on these concurrent actions should be established. The system must also guarantee that the order in which the attributes are changed respects causal order. For instance, if the number of items in the container diminishes an appearance change must be made because of their ingestion. On the other hand, changes to the cooling fan state are independent of the changes made to other objects.

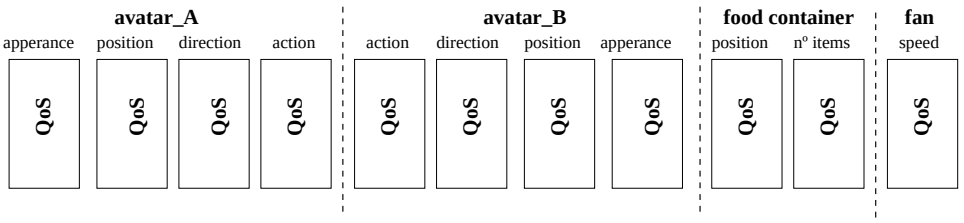

Figure 1: Independent Channels

In the following section, we describe different ways to configure the communication protocols in a setup were in each user node there is a replica of each object in the application. In this setup, users must be informed of all operation that change object attributes. In the following discussion we will use the term *channel* to describe a composition of communication protocol components. The quality of service offered by the channel depends on the protocols that compose a channel. For instance, one may build a communication channel that supports ordered and reliable point-to-point communication (typically supported by a TCP/IP protocol stack). Later in Section 3.1, we will provide a more precise definition of a channel in the context of the *Appia* system.

2.3.1 Independent communication channels

A possible protocol composition for the information sharing in the virtual room just mentioned, is to use an independent communication channel for each attribute, as depicted in Figure 1. This architecture has the advantage that allows each attribute to use a particular quality of service. For example, to disseminate cooling fan speed changes it would use a reliable communication channel without any particular order. On the other hand, to disseminate state changes to the food container it would use a total-order protocol, to enforce that all nodes see the changes in a coherent order. The disadvantage of this alternative is that it is not possible to use a shared communication component to enforce that causal order is maintained between the changes made to different attributes. For instance, it would be impossible to causally order updates to the number of items in the food contained and updates to the appearance of who has eaten them (therefore, it would be possible to observe changes in the appearance before observing its cause, the removal of a food item).

2.3.2 A single shared channel

A frequent solution to the coherence problems raised by the previous architecture is to use a single channel that is shared among all attributes, as

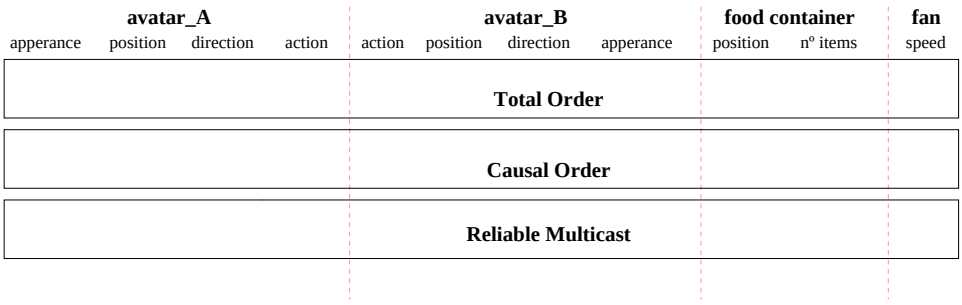

Figure 2: Global Channels

illustrated in Figure 2. This channel must satisfy the strongest order required by the shared objects. In the example it would be total-order. The drawback of this solution is that the communication among all attributes would use total-order, when only a small subset of the attributes demanded it. Since dissemination protocols that provide total-order are typically less efficient than those that only provide causal-order, or even no order at all (for the speed of the cooling fan), the use of strong total order might be an overkill, for those attributes not requiring it, leading to possible degradation of performance.

2.3.3 Shared channel with inter-channel dependencies

As just shown, neither the use of a single independent channel for each attribute, nor the use of a single channel shared among all attributes fully satisfies the requirements in our example. A solution is to create a protocol composition in which some attributes share some ordering properties, without forcing all attributes to share those properties.

In our example, all attributes would use a reliable information dissemination channel. This would be the quality of service strictly required for the dissemination of the cooling fan speed. Therefore this attribute would not use any additional ordering protocol. The remaining attributes would share a common causal order, to enforce the relations between attributes, as previously mentioned. Finally, a stronger ordering protocol, total order, would be shared among avatars and only be used for the avatars' actions. Since there is a certain predictability in the avatar's displacement it would be possible to reduce network traffic by extrapolating its movement. This could be achieved by another protocol, dead-reckoning, that does movement prediction and only propagates position updates when the forecast deviation is above a certain threshold. As this prediction potentially generates minor inconsistencies between the position of an avatar and what other users perceive, and it is a requirement that an avatar *must* be near the food container

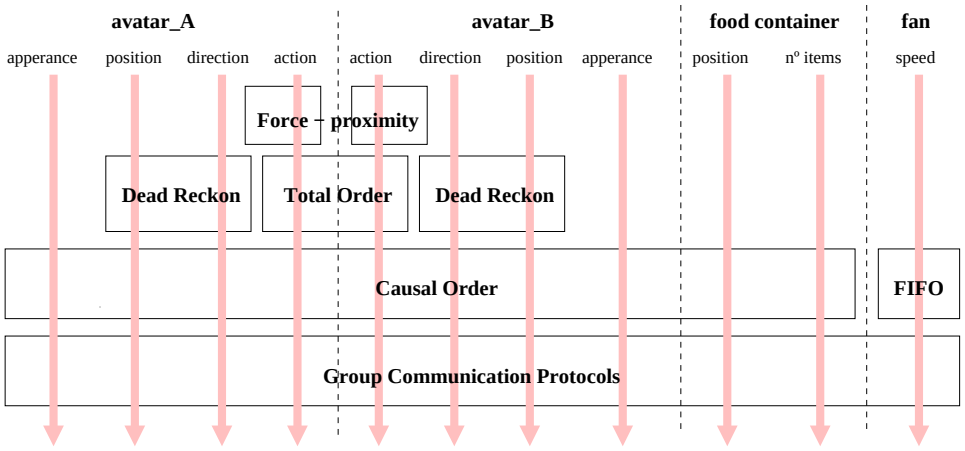

Figure 3: Shared Channels

in order to be able to eat, the avatar's actions would need a new protocol, called force-proximity, that guarantees that when an eat action is disseminated every user accurately sees the avatar near the basket. The resulting protocol composition is depicted in Figure 3.

In the following section we will describe a configurable communication system that allows the implementation of such adaptation.

3 A configurable communication system

3.1 The *Appia* protocol composition framework

Appia [11] is a protocol composition and communication framework that allows communication channels, each with its own *QoS*, to be integrated in a coherent multi-channel protocol stack. Using *Appia*, the application designer can specify the protocol stack that meets her/his QoS requirements through the composition of micro-protocols.

In the previous sections, we have identified the need for inter-channel coordination to support MOO applications. For instance, we have identified the need to preserve causal order across different channels. Similar examples have been identified by other research teams [14, 4]. A powerful feature of *Appia* is that it provides the mechanisms to express inter-channel coordination.

Stack composition in *Appia* relies on a clear separation between two related concepts: *layers* and *sessions*. We define a *layer* as the implementation of a protocol. All protocols implement the same *event interface*, which defines the types of events each layer is able to consume and produce. The format and semantics of these events is irrelevant to our exposition.

Typical examples of events are data transmission requests, indications and confirmations. Examples of layers are "datagram transport", "positive acknowledgment", "total order", "checksum", etc. Examples of relevant layers and events in the context of fault-tolerant applications can be found in [8].

An ordered set of layers (protocols) defines a quality of service. When a new quality of service is defined, *Appia* gathers the event types that each layer is able to produce and consume and uses that information for:

Performance improvement *Appia* defines event routes for each event type that will flow in a composition. Each event route keeps a reference to the protocols that are interested in receiving that event type. When an event travels a protocol composition, it will only be delivered to the protocols in his corresponding event route. We have shown elsewhere that this feature improves *Appia* performance [11].

Incoherent composition detection To be able to behave correctly, most protocols rely on the services provided by other protocols in the composition. One example of incoherent compositions are those where one service fundamental for one protocol is not provided by any other. In *Appia*, protocols can only communicate by the exchange of events. Layers are free to declare event types that must be provided by some other protocol in the composition. When creating a protocol composition, the *Appia* runtime throws an exception if it is found that one event type required by one protocol is not provided by any other.

As in *x*-kernel, we define a *session* as an instance of a layer [9]. The session maintains state that is used by the protocol code to process events. A protocol that implements ordering may keep a sequence number or a vector clock as part of the session state. In connection oriented protocols, the session also maintains information about the endpoints of the connection. Note that it is often useful to maintain several active sessions for the same layer even when they share the same endpoints: for instance, one might want to have different FIFO channels for different priority streams.

We can now provide a precise definition of a *channel* in the *Appia* system. A channel is defined as an ordered sequence of sessions. Sessions are used as a composition mechanism to implement coordination among channels. If needed, different channels may share the same session at one or more layers. The common session stores state that is shared by all channels and can implement the desired coordination. Each channel is modeled on a previously defined quality of service. Channels inherit the event routes defined by qualities of service mapping layers of the latter on sessions of the former. However, *Appia* does not require that two or more channels sharing at least one session are modeled by the same quality of service. In fact, the quality of service and the number of simultaneous channels where a session is being used may be transparent for the session. This is an innovative feature

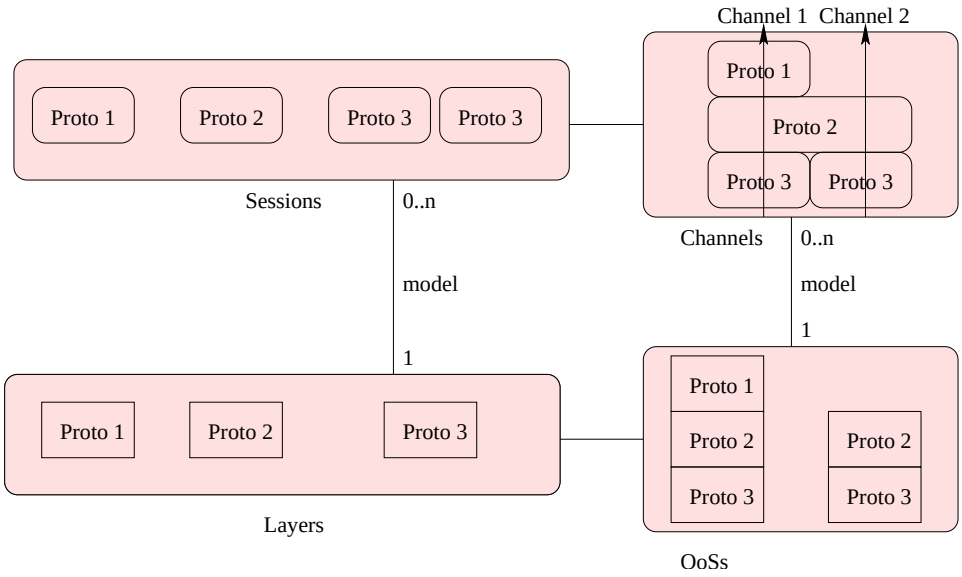

Figure 4: Relation between the basic concepts of *Appia*

of *Appia* that, to the extent of our knowledge, can not be found in other composition frameworks.

Figure 4 shows how three protocols named *Proto 1*, *Proto 2* and *Proto 3* can be used in the definition of two channels. Layers are grouped for the definition of two independent qualities of service (QoSs). Each of these QoSs can be used for the definition of any number of channels. In this example, protocols 1 and 2 have each one session created, while protocol 3 has two sessions created. Sessions are then used for defining the channels. Note that each channel uses one session for each layer declared on his corresponding QoS. However, when the channels are defined, the same session of protocol 2 is shared between both channels. Protocol 3, in turn, uses one separate session for each channel.

3.2 Defining channels with shared sessions

As noted in Section 2.3, users may share a set of objects and their respective attributes from the virtual environment. When one of the attributes is changed, its new state must be disseminated to all the processes that share the corresponding object. This dissemination must respect the consistency criteria defined by the application. With *Appia* the consistency criteria is mapped to a communication channel that offers the desired protocol composition. Therefore, for each attribute, a different communication channel is created: each channel is composed by the protocols required to provide the desired quality of service. Channels may have independent or shared sessions. When one wants to coordinate the activity of the channels at a

```
<!ELEMENT configuration  (attribute+)>
<!ATTLIST configuration object CDATA #REQUIRED>
<!ELEMENT attribute  (name, QoS)>
<!ELEMENT QoS  (micro-protocol+)>
<!ELEMENT micro-protocol  (name, sessionname?, initParameters*, mode)>
<!ELEMENT name  (#PCDATA)>
<!ELEMENT sessionname  (#PCDATA)>
<!ELEMENT initParameters  (parameter+)>
<!ELEMENT parameter  (#PCDATA)>
<!ELEMENT mode  (#PCDATA)>
```

Figure 5: System's grammar

certain protocol level, shared sessions should be used.

Applying this model to our previous example should result in the configuration depicted in Figure 3. Each attribute has its own channel to disseminate its own changes. But these channels are not independent. At the bottom of the stack, all channels share the set of protocols necessary to provide reliable group communication[1]. The channel used to disseminate the cooling fan speed only needs one more FIFO layer, whose state is preserved in a private session. All the remaining channels share a causal order session. The channels used to disseminate the position and direction of each avatar share a session of a dead-reckoning protocol. Finally, the two channels used to disseminate the eat action of the avatars also share a session of a total order protocol and each channel has its own session of a force-proximity protocol.

3.3 The configuration process

In the previous Section, we have shown how it is possible to use the *Appia* composition model to build a multi-channel protocol stack. In this section we show how the user can specify a configuration fitting her/his consistency requirements, for instance, to build the communication stack illustrated in Figure 3.

The XML language is appropriate for specifying the users' configuration. Using this language, the user can define a set of rules to build a well formed and structured file. These rules must be defined in a grammar (DTD Document Type Definition), that will be followed to define the file. Those files will be easily built and understood due to its organization.

The user defines an XML file for each object he/she wants to create. Although nothing prevents this file from being created in run-time, in our

[1]Actually, it could be possible to refine even further the example and consider the use of best-effort communication protocols in some channels but we omit such optimizations to maintain the exposition simpler.

current prototype we use exclusively specifications that are made prior to the execution. However, as we will describe, a single file can be used for each class of objects (for instance, for all avatars), thus allowing the creation of new objects in run-time.

The configuration file must follow the grammar rules that the system is ready to interpret, as presented in Figure 5. Selected portions of the XML configuration file for the avatar objects is presented in Figure 6 and the corresponding file for the fan is depicted in Figure 7. We will refer to these files to give concrete examples of some configuration features.

According to the grammar specification this file must have: the name of the object, the name of each attribute, for each attribute the declaration of the stack of protocols that implement the target consistency criteria, an identifier of the sessions to be used. Each channel is composed of a sequence of layers: the order in which they are listed corresponds to the order by which they appear in the communication stack. The configuration file for each object is parsed when the user places the object into the virtual environment to build the associated communication channel. When an object is shared, the configuration file of the channel is disseminated along with the object state, such that remote processes can create the corresponding communication channel in their local nodes.

Both protocols and sessions are identified by textual names in the configuration file. The session declaration has an attribute called `initParameters` that are used to configure the session. These parameters are passed to the session in an initialization event. For instance, in the definition of the dead-reckoning session this parameter is used to defined the threshold used to trigger an update. Each session is also characterized by an attribute called *mode* with the following meaning:

- If the session cannot be shared with other attributes, mode should be set to `localAttribute`. Such session does not need to be named as a unique name is assigned automatically in run-time using the object's name and the name of the attribute. An example of a session with this characteristic is the session of the Force-proximity protocol, which is used exclusively by the channel of the action attribute of each object.

- If a session is to be shared by more than one attribute of the same object, but not with attributes of other objects, the mode should be set to `localObject`. As before, the name of the session is automatically assigned in run-time using the name of the object. An example of such a session is the dead-reckoning protocol that controls the dissemination of the position and direction attributes of each avatar: for each object a different session is created that is shared among these two attributes.

- If a session is to be shared among different objects, the mode should be set to `shared`. In this case the user must identify the name of the

```
<configuration object="avatar">
  <attribute>
    <name>
      apperance
    </name>
    <QoS>
      <micro-protocol>
        <name>
          Causal Order
        </name>
        <sessionname>
          causal session
        </sessionname>
        <mode> shared </mode>
      </micro-protocol>
      <micro-protocol>
        <name>
          Reliable group communication
        </name>
        <sessionname>
          group session
        </sessionname>
        <mode> global </mode>
      </micro-protocol>
    </QoS>
  </attribute>

  ...

  <attribute>
    <name>
      action
    </name>
    <QoS>
      <micro-protocol>
        <name>
          Force-proximity
        </name>
        <mode> localAttribute </mode>
      </micro-protocol>
      <micro-protocol>
        <name>
          Total Order
        </name>
        <sessionname>
          total session
        </sessionname>
        <mode> shared </mode>
      </micro-protocol>
      <micro-protocol>
        <name>
          Causal Order
        </name>
        <sessionname>
          causal session
        </sessionname>
        <mode> shared </mode>
      </micro-protocol>
      <micro-protocol>
        <name>
          Reliable group communication
        </name>
        <sessionname>
          group session
        </sessionname>
        <mode> global </mode>
      </micro-protocol>
    </QoS>
  </attribute>
</configuration>
```

Figure 6: Avatar XML configuration file (partial)

```
<configuration object="fan">
  <attribute>
    <name>
      speed
    </name>
    <QoS>
      <micro-protocol>
        <name>
          FIFO Order
        </name>
        <mode>
          localAttribute
        </mode>
      </micro-protocol>
      <micro-protocol>
        <name>
          Reliable group communication
        </name>
        <sessionname>
          group session
        </sessionname>
        <mode>global</mode>
      </micro-protocol>
    </QoS>
  </attribute>
</configuration>
```

Figure 7: Fan XML configuration file

session to be shared. This allows objects from different types to share different sessions of the same protocols. In our example, the total order and causal sessions are shared sessions.

- If one knows *a priori* that, due to some global consistency criteria, all objects should share a given session, the mode should be set to `global`. In this case, the run-time ensures that no other attribute may create a new session of that protocol. In our example, all attributes share a single session of the group communication protocols.

The use of modes allows to create configuration files that can be applied to several objects of the same type. In our example, all avatars may use the configuration file of Figure 6. Thus, this sort of fine-grain configuration does not restrict the number of avatars than can be created in run-time.

4 Performance

The system has been implemented as a set of extensions and new layers to the *Appia* system [11]. A companion prototype multi-user cooperative application, that demonstrates the the operation of the system was also developed.

To illustrate the comparative performance of the different configurations, we have measured the round-trip delay associated with the propagation of updates in shared attributes. We have considered the "position" and "action" attributes, since these have quite different requirements in terms of communication protocols. We have measure the values obtained with three different configurations for the supporting communication channels:

	Independent channels	Shared channel	Shared sessions
position	68	145	74
action	71	150	147
inter-channel consistency	no	yes	yes

Figure 8: Comparative performance

i) independent channels, where each channel offers different properties but where it is impossible to enforce inter-channel constraints; *ii)* a single shared channel enforcing the strongest requirement of both attributes, in this case, total order; *iii)* channels that enforce distinct properties but where inter-channel dependencies are expressed using the notion of shared sessions.

The results are depicted in Figure 8 (all values are in milliseconds). It can be observed that the configuration that uses independent channels offers the best round-trip delays but does not allow to enforce inter-channel constraints (order across different attributes). If such constraints need to be enforced, the use of shared sessions is more beneficial than to use a single channel, since in this case, total order does not need to be used when dissemination updates to the position attribute. It can be also observed that the degradation in performance for the configuration using shared sessions is not significant when compared with the degradation of performance incurred by a solution that requires the use of a single shared channel.

5 Related work

All existing MOO systems, such as [2, 6, 13], provide support for information sharing. DIVE [6] and SPLINE [2] use a replicated database approach: all interaction is performed through the replicated database. Although they offer a clean separation between the application and the replicated database, applications have little control over the replication issues. In particular, none of these systems allow the application developer to specify customized algorithms. Furthermore, from the point of view of communications support, existing MOO systems are usually tied to a single quality of service. For instance, NPSNET [10] only uses unreliable communication while DIVE only use reliable communication. SPLINE support both reliable and unreliable with ordering for messages regarding the same object. However in some situations it could be useful to force message ordering for a particular set of objects. In all existing systems there is no support for quality of service adaptation that takes into consideration application specific requirements. As result of their monolithic structure these systems are restricted to a single user model.

In recent years, there has been a significant progress in the development

of group communication infrastructures. The latest systems offer a very impressive range of configuration facilities. For instance, Horus [15] allows communication stacks to be changed in runtime; BAST [7] allows different protocols to be selected to implement the same services under different usage patterns. Coyote [3] allows the same message to be processed by different protocols in parallel. However, these systems lack built-in mechanisms to implement inter-channel coordination. Although presenting different composition models, none of these frameworks is able to support inter-channel coordination while hiding the final composition schema from the programmer of the protocols to be shared. Previous support for these kind of features has always been limited to particular cases, typically for performance improvement. This is the case of the Horus's FAST protocol [15].

Frameworks explicitly supporting inter-channel constraints lack generality. The work of CCTL [14] uses independent communication channels which are managed by a single control channel. The quality of service of each channel must be chosen by the application programmer from one of a predefined set available at the framework. Another example is the Maestro [4] system, that illustrates the difficulties of maintaining consistent failure detection when channels with diverse characteristics are used concurrently.

6 Discussion

The main advantage of the proposed architecture is that the user is able to configure a communication stack that satisfies the consistency requirements of the set of shared objects used by the application. A specialized communication channel can be assigned to each object attribute and the different channels can be coordinated thanks to the concept of configurable shared sessions. This and the possibility to access domain-specific information from a session allowed for different protocols, not just related to communication but with general consistency required for replication, another important aspect of MOO applications. This is illustrated, for instance, by the dead-reckoning protocol whose threshold value can be configured by the user depending on the object being shared. Supporting both concerns under the same infrastructure while still maintaining the separation of concerns allows to define coherent consistency from the network-level up to application-level and yet allow for separate development and reasoning.

The composition model offered by *Appia* is a step forward with regard to previous approaches to support MOO applications. However, the current system still exhibits some limitations. A disadvantage of the configuration procedure is that it requires the use of a global name space for the sessions. When two channels need coordination, the same session name must be used in configuring the channels. Thus, in practice, the user is forced to have a global view of the configuration of all channels. The provision of *local* and

global modes mitigates this disadvantage, as a single configuration file can be used for all objects that share a given protocol composition. Therefore, the object has not to consider the configuration of channels on an object by object basis, but in terms of classes of objects.

One of the challenges raised by the model is the development of multi-channel communication protocols. Although session sharing is provided by *Appia* since its inception, several protocols developed for it do not consider this capability. This is due to two main reasons. One is concerned with programming discipline. Since the concept of shared sessions is unusual, programmers do not consider this case unless directly requested to do so. Another reason is that the complexity of coding a protocol that accepts multi-channel sessions is highly variable. Some protocols, such as FIFO protocols are very simple to implement but others are more complex.

To illustrate the difficulty of building multi-channel protocol we give the example of the total order protocol used in our MOO. The protocol is a sequencer-based protocol: a given member of the group of replicas is elected to assign sequence number to all messages exchanged in the group. In a multi-channel implementation, a single sequence of sequence numbers is used across all channels that share the same session. However, the protocol designer has to decide if it creates an additional channel just to exchange control information (such as the sequence numbers) or if it used one of the data channels and, in the later case, which one to use. Since different channels may have different properties (protocols), some introspection may be required to select the most appropriate channel.

7 Conclusions

This article presents a configurable communication architecture that satisfies the complex requirements raised by interactive multi-user applications. This architecture is based on the vertical and horizontal composition of protocols. Vertical composition enables the configuration of the protocols that compose the channels used for update dissemination. Horizontal composition enables to respect dependencies between attributes through the use of shared sessions. The configuration may be performed during the application deployment phase, without being fixed into the code. This allows the configuration to be adapted not only to the properties of the shared objects, but also to the environment in which the system is running.

Acknowledgements The authors are grateful to the anonymous reviewers for their comments on an earlier version of this paper.

References

[1] M. Antunes and A. Silva. Using separation and composition of concerns to build multiuser virtual environments. In *Proceedings of the 6th International Workshop on Groupware - CRIWG'2000*, Madeira, Portugal, 2000. IEEE.

[2] J. Barrus, R. Waters, and D. Anderson. Locales: Supporting Large Multiuser Virtual Environments. In *IEEE Computer Graphics and Applications*, pages 16(6):50–100, Nov. 1996.

[3] N. Bhatti, M. Hiltunen, R. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Transactions on Computer Systems*, 16(4):321–366, November 1998.

[4] 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.

[5] S. Evans. Building blocks of text-based virtual environments. Technical report, Computer Science University, University of Virginia, April 1993.

[6] E. Frécon and M. Stenius. Dive: A Scalable Network Architecture for Distributed Virtual Environments. In *Distributed systems Engineering Journal(Special Issue on Distributed Virtual Environments)*, number Vol. 5, No 3, pages 91–100, September 1998.

[7] B. Garbinato and R. Guerraoui. Flexible protocol composition in Bast. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS-18)*, pages 22–29, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

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

[9] 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.

[10] 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.

[11] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 707–710, Phoenix, Arizona, April 2001. IEEE.

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

[13] J. Pubrick and C. Greenhalg. Extending Locales: Awareness Management in MASSIVE-3. In *URL:http://www.crg.cs.nott.ac.uk/research/systems/MASSIVE-3*, September 1999.

[14] 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.

[15] R. van Renesse, K. Birman, and S. Maffeis. Horus: A flexible group communications system. *Communications of the ACM*, 39(4):76–83, April 1996.